UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2016-33
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 29/08/2016 in Luxembourg
to obtain the degree of

DOCTEUR DE L´UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

NICO NACHTIGALL

Born on 1st April 1982 in Berlin (Germany)

DOMAIN COMPLETENESS OF MODEL TRANSFORMATIONS AND SYNCHRONISATIONS

Dissertation defense committee:

Prof. Dr. Thomas Engel, Dissertation Supervisor
Professor, Université du Luxembourg, Luxembourg

Prof. Dr. Barbara König
Professor, Universität Duisburg-Essen, Germany

Prof. Dr. Ulrich Sorger, Chairman
Professor, Université du Luxembourg, Luxembourg

Dr. Frank Hermann, Deputy Chairman
Carmeq GmbH, Berlin, Germany

Dr. Raimondas Sasnauskas
SES Engineering, Luxembourg

Dr. Benjamin Braatz
Graph-IT GmbH, Cologne, Germany

To my sons Emil & Max, my wife Susann and my
grandfather †Axel

Foreword & Acknowledgements

The intrinsic question of most activities in information science, in practice or science, is *"Does a given system satisfy the requirements regarding its application?"* Commonly, requirements are expressed and accessible by means of models, mostly in a diagrammatic representation by visual models. The requirements may change over time and are often defined from different perspectives and within different domains. This implies that models may be transformed either within the same domain-specific visual modelling language or into models in another language. Furthermore, model updates may be synchronised between different models. Most types of visual models can be represented by graphs where model transformations and synchronisations are performed by graph transformations. The theory of graph transformations emerged from its origins in the late 1960s and early 1970s as a generalisation of term and tree rewriting systems to an important field in (theoretical) computer science with applications particularly in visual modelling techniques, model transformations, synchronisations and behavioural specifications of models. Its formal foundations but likewise visual notation enable both precise definitions and proofs of important properties of model transformations and synchronisations from a theoretical point of view and an intuitive approach for specifying transformations and model updates from an engineer's point of view. The recent results were presented in the EATCS monographs "Fundamentals of Algebraic Graph Transformation" (FAGT) in 2006 and its sequel "Graph and Model Transformation: General Framework and Applications" (GraMoT) in 2015. This thesis concentrates on one important property of model transformations and synchronisations, i.e., syntactical completeness. Syntactical completeness of model transformations means that given a specification for transforming models from a source modelling language into models in a target language, then all source models can be completely transformed into corresponding target models. In the same given context, syntactical completeness of model synchronisations means that all source model updates can be completely synchronised, resulting in corresponding target model updates. This work is essentially based on the GraMoT book and mainly extends its results for model transformations and synchronisations based on triple graph grammars by a new more general notion of syntactical completeness, namely domain completeness, together with corresponding verification techniques. Furthermore, the results are instantiated to the verification of the syntactical completeness of software transformations and synchronisations. The well-known transformation of UML class diagrams into relational database models and the transformation of programs of a small object-oriented programming language into class diagrams serve as running examples. The existing AGG tool is used to support the verification of the given examples in practice.

Acknowledgements

First of all, I would like to thank Frank Hermann for his great support and patience. He did a great job in guiding me and answering the big amount of questions that occured during my PhD studies. Without the help of Frank, this work would not have been possible. Furthermore, I would like to thank Benjamin Braatz, Barbara König and Claudia Ermel for their continous help, for proofreading my work and for answering many questions. Also many thanks to Thomas Engel and Raimondas Sasnauskas for supervising my thesis and for giving me the opportunity to finish this thesis.

Supported by the Fonds National de la Recherche, Luxembourg 4895603.

Fonds National de la
Recherche Luxembourg

Contents

Chapter 1

General Introduction

While this chapter gives a short and general introduction to model transformations and synchronisations, its completeness and correctness properties and the main results of this thesis, Chap. 2 gives a more in-depth introduction to model transformations, synchronisations, their properties (in particular the completeness property) and their link to the theory of graph transformations as the underlying formal framework.

1.1 Models in Information Science

Commonly, visual models are considered as being simplified versions of complex systems [Béz05] that are obtained by a process of abstraction either from mental concepts or from the properties and features of the sytem itself if it already exists during model creation. The process of abstraction requires the identification of important properties of the intended system's application and use while omitting properties and features that are considered as being irrelevant for the intended model usage. In this context, the properties that are captured by the model are requirements on the intended system's application and use. In most cases, system models are used in contexts where the system itself is still in the process of being developed, i.e., the system is (requirements are) only accessible by means of models during most stages of development. Therefore, models have an important role in information science [Fet99, Tha11] with an image characteristic in the sense that models are (simplified) images of the originals they represent with the purpose to take models as substitutes for their originals within a given temporal and situational context. The importance of models in information science is addressed by several communities with the object management group (OMG) being one of the largest global initiatives. Models are the cornerstones of their work where each complex system is rather being designed by means of visual models before being implemented [Put01]. This enables a view on problems and their potential solutions before starting with an (extensive) full implementation and therefore, reduces risks. At its extreme, the system is developed strictly by modelling and the implementation is (completely) generated from models, i.e., "the model is the code." [Tha11].

Furthermore it is generally stated that complex systems cannot be comprehended in their entirety, as, the human capacity is limited and therefore, abstractions are necessary that allow us to manage complexity [Qua98] by focusing on the relevant details only. In model-driven software and systems development (MDD), models usually exhibit the following characteristics [Sel03]. The model must be (1) easy to understand for the intended group of model users, (2) accurate, i.e., the model only reflects real properties of the system, (3) predictive, i.e., the model must be a reflection of important, non-obvious properties of a system under development, (4) inexpensive,

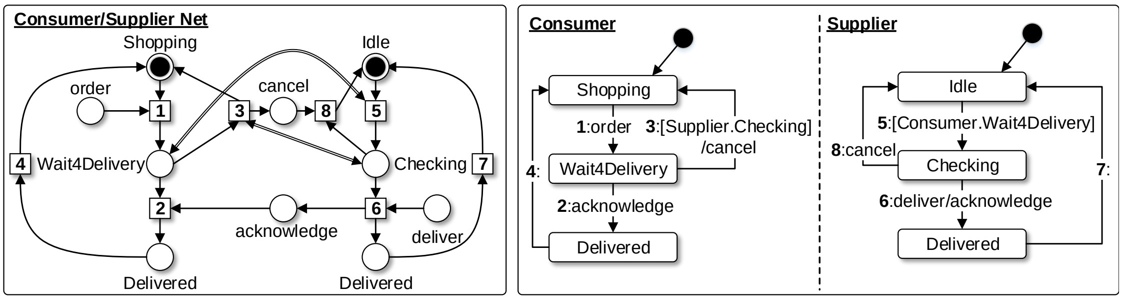


Figure 1.1: Models in Information Science - Place/Transition System (left) & Statechart (right)

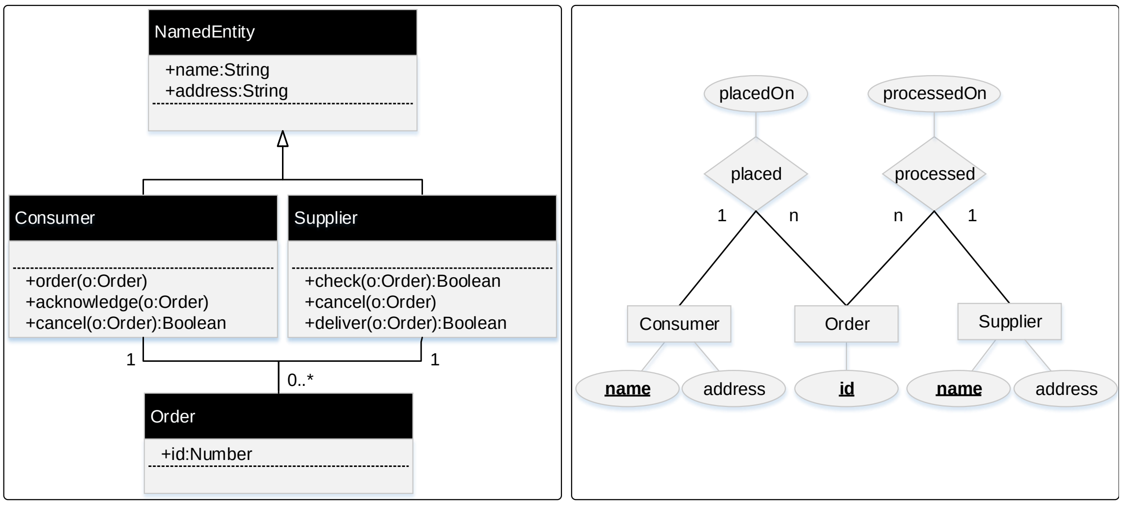


Figure 1.2: Models in Information Science - UML Class Diagram (left) & Entity-Relationship Diagram (right)

i.e., the creation and analysis of a model must be cheaper than the implementation of the system and its analysis itself.

Although most models share a common set of properties which make them identifiable as such, Herbert Stachowiak [Sta73] and Bernd Mahr [Mah09, Mah10] each developed a comprehensive theory of model-being. Instead of identifying an object as a model based on a definition with a fixed set of properties, the model-being of an object is rather the result of a context-specific judgement. An object O is interpreted as a model M of something A with the purpose for something B and therefore, O carries a cargo (some type of information) from A to B. The model object O is produced based on (a subset of) the properties of original object A. Therefore, these properties must be identifiable in an observation on O. Analogously, purpose B is realised based on (a subset of) the properties of model object O. Therefore, these properties must be identifiable in an observation on the resulting object B. Similarly, [MFBC12] advocates that model-being depends on the existence of a representation relation between model object and its original where a model is considered as the representation of its original.

Figs. 1.1 and 1.2 depict different types of system models from different perspectives and within different problem domains that are used in MDD [WHR14, BBG05a, BBG05b] for communication between experts of the same domain and across different domains. Fig. 1.1 depicts models of the dynamic behaviour of a given (concurrent) purchase order system that may be used to implement the behaviour in the form of operations in computer programs. In contrast to Fig. 1.1, Fig. 1.2 depicts models that represent the static organisation of concepts (data) and their interrelationships in the system, i.e., the ontology for the domain of discourse (the purchase order scenario). These models may be used to implement static aspects of the system, e.g., class definitions in an object-oriented programming language or relational database structures that both may serve as the data basis for the operations in computer programs.

Fig. 1.1 (right) illustrates a UML statechart [UML15, RJB04]. UML statecharts are originated from classical hierarchical Harel statecharts [Har87, HKP05] which extend the formalism of finite-state machines [DR15] mostly by hierarchically nested states and orthogonal regions to enhance the readability and scalability of the models. The statechart consists of (system) states (boxes) and transitions between states (arrows). Each transition describes some system behaviour by the change from one system state to another state. Furthermore, the statechart is separated into two orthogonal (concurrently operating) regions as marked by a dashed line. In each region, the initial state is marked by a black node. Each transition has an inscription of the form: $[event]['['condition']'][\,'/'action]$ where [] is meant to be optional. A transition is enabled if the source state of the transition is the current state, the *event* occured and the *condition* can be evaluated to true. If a transition fires, then the current state changes to the target state of the transition and the *action* is performed. The following behaviour of a consumer-supplier system is modelled in Fig. 1.1 (right): The consumer and supplier start with Shopping and Idle, respectively. After Shopping, the consumer may decide to order and consequently, waits for delivery afterwards (state Wait4Delivery). If the consumer is waiting for delivery, the supplier immediately pass over to Checking the order. As long as the supplier is Checking and the consumer is waiting, the consumer may cancel the order at any time which directly causes a cancellation on the supplier side. After cancellation both sides return to their initial states. After Checking, the supplier may decide to deliver the ordered items while asking the consumer to acknowledge the receipt at the same time. Subsequently, the consumer acknowledges, both transition into state Delivered and may return to their initial states at any time.

Fig. 1.1 (left) depicts a Petri net, more precisely a place/transition net with initial marking [NMN^{+}92, Rei85], which captures the same dynamic behaviour from Fig. 1.1 (right). The marking is given by a (black) token distribution which defines the system state. By firing an enabled transition (boxes) the state can be changed. A transition is enabled, if each input place (circles with outgoing edges to the corresponding transition) contains a token. When a transition is fired, all tokens from the input places are removed and for each output place (circles with incoming edges from the corresponding transition) a token is added. Therefore, for each transition in Fig. 1.1 (right) there is a transition defined in Fig. 1.1 (left) and the initial states (each state and event) in the statechart are (is) modelled by an initial marking (a place) in the net.

While the statechart delivers a readable model of the behaviour, the Petri net may be used for formal analysis of the behaviour as supported by a variety of existing tools.

Fig. 1.2 (left) presents a UML class diagram [UML15, RJB04]. The diagram designates classes Consumer, Supplier and Order as central concepts of the system. Consumers and Suppliers are named entities (cf. inheritance relationship with class NamedEntity) and therefore, they have a name and address both of type String. Furthermore, a Consumer has the following operations (possible behaviour): A Consumer may: 1. place an Order o, 2. acknowledge the receipt of an Order o, or 3. cancel an Order o with Boolean return value. By Fig. 1.1, a cancellation is only possible if the Supplier is checking the order as indicated by a Boolean return value which returns true in case of success and otherwise false. Similarly, a Supplier may: 1. check an Order o, 2. cancel an Order o, or 3. deliver an Order o. Furthermore by the interrelationships as given in the diagram: 1. Each Consumer may have placed an arbitrary number of Orders, each

Order identified by an id, 2. each Supplier may have processed an arbitrary number of Orders, and 3. each Order was placed (processed) by exactly one Consumer (Supplier).

Fig. 1.2 (right) presents an entity-relationship diagram containing the same concepts and interrelationships from the class diagram. However, note that the operations (suggestions of possible behaviour) of the entities are missing, the interrelationships placed and processed become explicit figures with additional attributes placedOn and processedOn instead of simple arrows and attributes name and id are particularly marked as unique instance identifiers (primary keys).

Therefore, the class diagram is rather used to generate the base frame of a program in an object-oriented programming language while the entity-relationship diagram is used to create the structure of a relational database that may be used by the program.

1.2 Models, Transformations & Synchronisations

Sec. 1.1 already highlighted the importance of models in information science especially in the model-driven development of software and systems in general. It was stated and supported by different concrete examples that models are created and intended to be used in the context of different problem domains where they capture requirements on the intended system's application and use from different perspectives. The requirements may change over time, need to be refined or may be inferred from requirements that are formulated in other domains. This implies that models may be transformed either within the same domain-specific modelling language (DSL) or into models in another language. Furthermore, model updates may be synchronised between interrelated models and different domains.

For example, the net in Fig. 1.1 (left) can be completely inferred from the statechart (right). A corresponding model transformation from models that are formulated in the DSL of UML statecharts into models that are formulated in the DSL of place/transition nets is given in [EEPT06]. Vice versa, UML statecharts cannot be completely inferred from place/transition nets in any cases, since, the statecharts may contain information (e.g., the explicit separation into concurrently operating sub-systems (orthogonal regions)) that are not available in (flat) nets. However, a transformation from place/transition nets into statecharts can be used in order to obtain initial statechart models from existing nets and where the statecharts are refined at a later step, e.g., by adding orthogonal regions. While a transformation from UML statecharts into place/transition nets may be used for simulating the behaviour of the model based on the net [HS04], the reverse transformation may be used in order to obtain a more readable model in the form of a statechart from the net.

The same is true for the examples in Fig. 1.2. UML class diagrams and entity-relationship diagrams share a common set of knowledge while containing exclusive information at the same time. While both share the knowledge of domain entitites (Consumer, Supplier and Order) and their interrelationships, class diagrams may additionally capture operations for each entity and entity-relationship diagrams may make the interrelationships more explicit as well as may mark specific attributes as unique instance identifiers (primary keys). However, analogously to the place/transition net-statechart scenario, it is common practice to derive entity-relationship diagrams from class diagrams and vice versa in order to obtain initial models in the one domain from existing models in the other domain. This transformation is also referred to as the object-relational mapping between the concepts in object-oriented programming languages and concepts in relational database systems.

By this means, each class diagram may be in relation with a corresponding entity-relationship diagram. Therefore, a model update on a class diagram need to be synchronised with its entity-relationship diagram and vice versa. For example, by changing the attribute of entity Order in

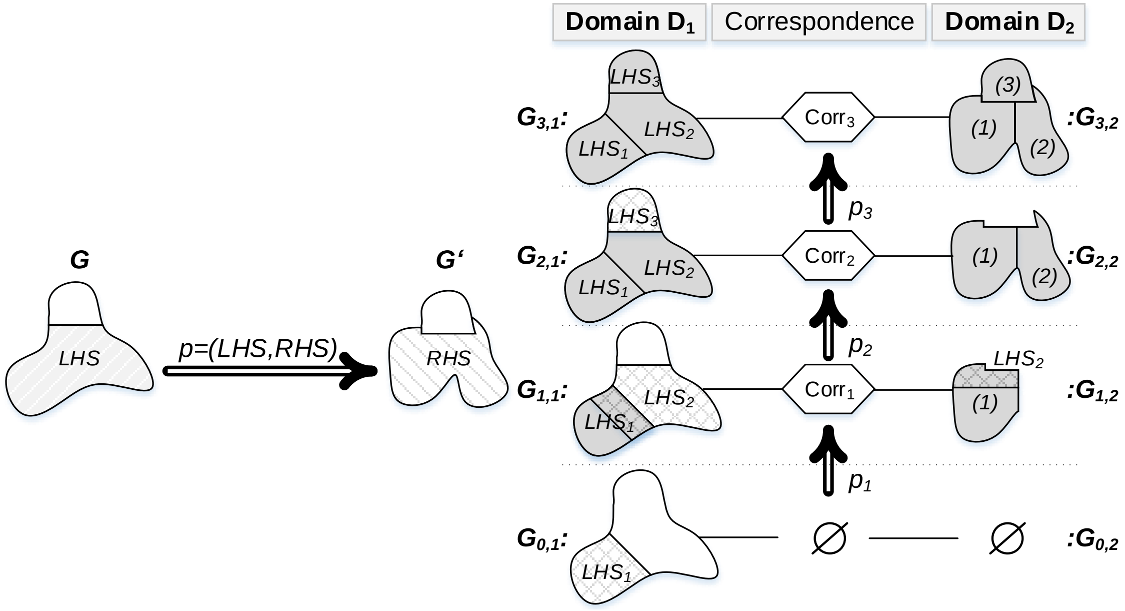


Figure 1.3: Rule-Based Graph Transformation Step (left) & Model Transformation Sequence (right)

Fig. 1.2 (left) from id to uid, in order to emphasize the uniqueness of the identifier, this change must lead to a corresponding update in Fig. 1.2 (right).

Note that especially if we think of model-being as a result of a judgement (cf. Sec. 1.1), "Everything is a Model" [Béz05] or more precisely, everything can be construed as a model, be it a mental concept or a physical entity. In this thesis, we focus on visual models in the form given in Figs. 1.1 and 1.2, their transformation and synchronisation. The typical example of model-to-model transformations, the transformation from UML class diagrams to relational database models (CD2RDBM) [EEGH15], serves as the running example throughout all chapters. In Sec. 2.1, we review general basic notions of model transformations and synchronisations in more detail.

1.3 Graph Transformation for Model Transformations & Model Synchronisations

In this thesis, the theory of graph transformations is used as the consistent formal framework for defining models, model updates and model transformations as well as for performing model transformations and synchronisations [EEPT06, EEGH15]. Therefore, models are represented by graphs. We assume that most types of visual models as given in Sec. 1.1 can be represented by graphs. Essentially, a graph consists of a set of nodes and a set of edges between nodes. A model update defines which part of a graph is deleted and which part is added by the update. A model transformation is defined in terms of a set of graph transformation rules. A transformation rule (or production) $p = (LHS, RHS)$ contains a left-hand side (LHS) and a right-hand side (RHS). When applying a rule p to a graph G, then G is transformed in the sense that LHS is replaced by RHS in G leading to a new graph G', denoted by $G \overset{p}{\Longrightarrow} G'$ (cf. Fig. 1.3 (left)). Therefore, if LHS is a subgraph of RHS (i.e., p only creates elements and therefore is non-deleting) then $RHS \setminus LHS$ is mainly added to G while LHS is preserved by the rule application. Otherwise, if intersection

$LHS \cap RHS \neq LHS$, then $LHS \setminus RHS$ is deleted in G, $LHS \cap RHS$ is preserved and $RHS \setminus LHS$ is added to G. Note that this is an intuitive approach to graph transformations via set-theoretic operations $\setminus$ and $\cap$ on nodes and edges of graphs while technically, transformations are defined based on the category-theoretic concept of pushouts (cf. Sec. 2.2.4).

Commonly, model transformations from models in source domain D_1 (e.g., UML class diagrams) into models in target domain D_2 (e.g., entity-relationship diagrams) are defined based on a set of non-deleting graph transformation rules and performed by graph transformations, i.e., by applying transformation rules successively (so called model transformation sequences) as illustrated in Fig. 1.3 (right). The source (input) model (graph) $G_{0,1}$ of a transformation sequence in source domain D_1 is traversed step-wise in n steps and target (output) model (graph) $G_{n,2}$ in target domain D_2 is constructed in parallel as follows:

1. At each step i ($i \in \{1 \ldots n\}$), a rule $p_i = (LHS_i, RHS_i)$ is applied to graph $(G_{i-1,1} \leftarrow Corr_{i-1} \rightarrow G_{i-1,2})$ leading to a new graph $(G_{i,1} \leftarrow Corr_i \rightarrow G_{i,2})$ where $G_{i,1}$ represents the graph part in domain D_1, $G_{i,2}$ the graph part in domain D_2 and $Corr_i$ is the correspondence that relates elements from $G_{i,1}$ with elements from $G_{i,2}$. - The model transformation sequence in Fig. 1.3 (right) consists of three steps ($n = 3$).

2. At the first step, only source graph $G_{0,1}$ is given as input to the transformation while $Corr_0$ and $G_{0,2}$ are empty (- nothing is transformed yet). Therefore, by applying rule p_1, LHS_1 in $G_{0,1}$ is marked as transformed in $G_{1,1}$ (by dark grey colouring) and $Corr_1$ with dark grey area (1) are added (i.e., LHS_1 in $G_{0,1}$ is transformed into $G_{1,2} = (1)$).

3. Analogously, in the following steps i, LHS_i in $(G_{i-1,1} \leftarrow Corr_{i-1} \rightarrow G_{i-1,2})$ is transformed into (i) in $G_{i,2}$. Note that LHS_i may overlap with already transformed elements. For example in Step 2 in Fig. 1.3 (right), LHS_2 overlaps with LHS_1 in domain D_1 and with (1) in domain D_2 and the result of Step 2 is that $LHS_1 + LHS_2$ in D_1 together are transformed into $(1) + (2)$ in D_2. Step 3 in Fig. 1.3 (right) shows the case where LHS_i in D_1 is transformed one-to-one without changing its structure into (i) in D_2. This is especially true for model refactorings where domains $D_1 = D_2$, most steps are identical transformations and only a small part of the input graph is changed and therefore transformed into different structures.

4. The transformation sequence terminates and is complete, if the input graph $G_{0,1}$ is completely transformed, i.e., if $G_{0,1}$ is completely coloured (or marked) with dark grey such as is the case after Step 3 in Fig. 1.3 (right).

Thus, the rules for the model transformation are non-deleting, since, the source model is not transformed in-place but is only traversed step-wise and is rather preserved by the transformation while the target model is constructed in parallel.

Model synchronisations are performed based on model transformations w.r.t. a given model update.

In the literature, a variety of different graph transformation approaches is discussed with each having its own replacement mechanism for LHS by RHS in rule applications [Roz97, EEKR99, EKMR99]. In this thesis, we use the algebraic approach to graph transformations [EEPT06] which was extended to model transformations and synchronisations in [EEGH15]. In Sec. 2.2, we review basic notions of algebraic graph transformation. In Sec. 2.3, we review basic technical notions and concepts of model transformations and synchronisations based on graph transformations.

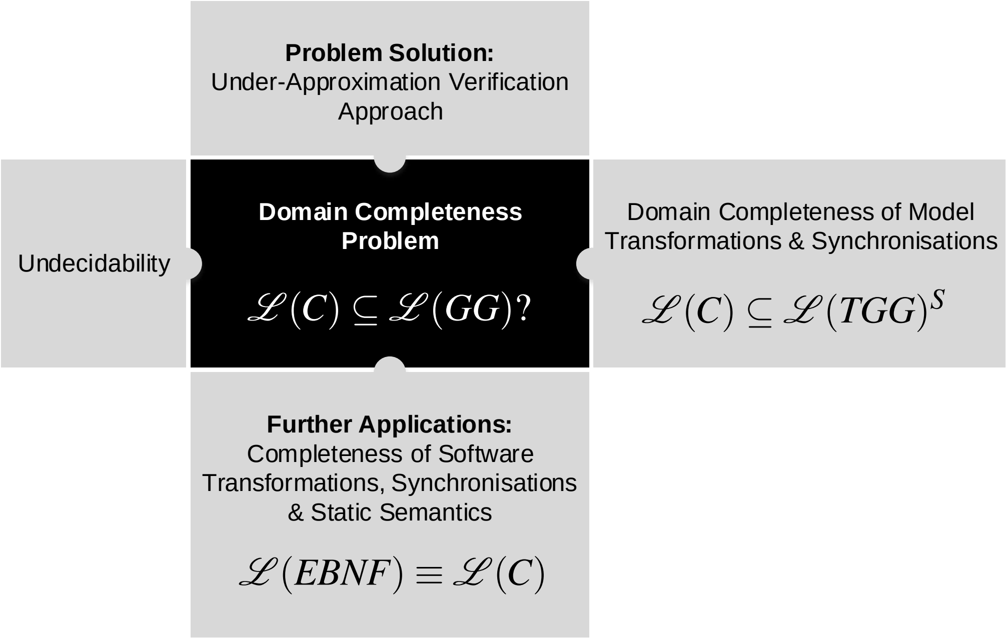

Problem Solution:
Under-Approximation Verification Approach
Undecidability
Domain Completeness Problem
$\mathscr{L}(C) \subseteq \mathscr{L}(GG)?$
Domain Completeness of Model Transformations & Synchronisations
$\mathscr{L}(C) \subseteq \mathscr{L}(TGG)^S$
Further Applications:
Completeness of Software Transformations, Synchronisations & Static Semantics
$\mathscr{L}(EBNF) \equiv \mathscr{L}(C)$

Figure 1.4: Main Results of this Thesis

1.4 Organisation of this Thesis & Main Results

Fig. 1.4 presents the main results of this thesis. The domain completeness problem is introduced as the core problem statement. Generally, we assume that the set of allowed models (graphs) in a given domain of discourse D, i.e., the visual domain-specific modelling language (DSL) of D, is defined by a set of (domain) graph constraints. A graph constraint formulates structural restrictions on graphs. Given a set of constraints C, then $\mathscr{L}(C)$ is the set of all graphs that satisfy the restrictions which are formulated by the domain constraints in C. Therefore, $\mathscr{L}(C)$ is the set of allowed models (graphs) in the given domain D, i.e., $\mathscr{L}(C)$ is the DSL of D. On the other hand, a set of graph transformation rules together with a start graph form a graph grammar GG. With $\mathscr{L}(GG)$ we denote the language of graphs that can be created by rule applications starting at the start graph. The domain completeness problem is defined as follows: Does it hold that $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$? Therefore, can all graphs that satisfy the domain constraints in C be created from the start graph in GG by successively applying the rules of graph grammar GG? - OR - Is the set of domain constraints C as restrictive as or more restrictive than grammar GG, respectively? - OR - Is DSL $\mathscr{L}(C)$ completely covered by the graph grammar GG?

It turns out that the domain completeness problem is undecidable in general, i.e., we cannot find a complete computable solution to this problem. Therefore, we propose an under-approximation solution to the domain completeness problem in order to verify the language inclusion $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$ of domain completeness approximately. Basically, the solution consists of a set of conditions that are sufficient but not necessary and need to be verified. Therefore, the solution may lead to false negatives but not to false positives, i.e., if the conditions are fulfilled, then it is ensured that the language inclusion holds. If the conditions are not fulfilled, then the language inclusion may hold or not hold and additional checks may be necessary in order to obtain (partial) truth.

Furthermore, we developed three main applications of the solution (cf. Fig. 1.4). The domain completeness problem of model transformations is stated as follows: Can all allowed models in the given domain D (all graphs in $\mathscr{L}(C)$) be transformed based on a given triple graph grammar *TGG*? - OR - Does it hold that $\mathscr{L}(C) \subseteq \mathscr{L}(TGG)^S$ for forward model transformations or $\mathscr{L}(C) \subseteq \mathscr{L}(TGG)^T$ for backward model transformations? Therefore, we have instantiated our solution to the approximate verification of the language inclusion $\mathscr{L}(C) \subseteq \mathscr{L}(TGG)^S$ or $\mathscr{L}(C) \subseteq \mathscr{L}(TGG)^T$, respectively. Thus, the language inclusion problem is stated in the context of a triple graph grammar *TGG* [ALS15, SK08, Sch94] instead of a flat graph grammar *GG* where the *TGG* specifies the model transformation. The domain completeness problem of model synchronisations is stated analogously: Can all allowed model updates in the given domain be synchronised? We show that this problem can be verified based on the results of model transformations. The third application of the solution is to verify the completeness of software transformations and synchronisations:

1. Can all programs (their abstract syntax trees (ASTs)) written in a given programming language be transformed?

2. Can all program updates (updates of their abstract syntax trees (ASTs)) be synchronised?

Programs written in a given programming language are represented by their abstract syntax models (ASTs) and transformed by performing model transformations of ASTs based on a given *TGG*. Commonly, programming languages are defined by a context-free word grammar. Therefore the third application of the solution mainly focuses on solving the problem to close the gap between the word grammar world and graph world by presenting a mapping from a context-free grammar in Extended-Backus-Naur Form (EBNF) notation into a set of graph constaints C such that the language of ASTs over the EBNF $\mathscr{L}(EBNF)$ is equivalent (isomorphic) to $\mathscr{L}(C)$, i.e., $\mathscr{L}(EBNF) \equiv \mathscr{L}(C)$.

Fig. 1.5 presents the peer-reviewed publications for the period of this thesis and their assignment to the corresponding chapters. Publications that are not assigned to a chapter contain no direct contribution to this thesis.

After we have introduced the main results, we refer briefly to their chapters.

1.4.1 Chapter 2: Model Transformations, Model Synchronisations & Formal Framework

While Chap. 1 gives a general introduction to the topic, in Chap. 2, we give a more in-depth introduction to model transformations and synchronisations. We recall different classes of transformations and synchronisations, i.e.:

1. In-place transformations vs. “external” transformations that preserve the source models,

2. horizontal vs. vertical transformations,

3. endogeneous vs. exogeneous transformations, and

4. model (text)-to-model (text) transformations.

Moreover, we clarify how meta-modelling is linked to model transformations and synchronisations.

We recall basic definitions and results of the theory of algebraic graph transformation. In particular this includes the category $\mathbf{AGraphs}_{ATGI}$ of typed attributed graphs with node type inheritance, their rule-based transformation, graph grammars, (nested) graph conditions and constraints and $\mathscr{M}$-adhesive transformation systems that generalise the concepts from attributed

Title	Authors	Chapter
Towards the Propagation of Model Updates along different Views in Multi-View Models [GNE$^+$16]	Susann Gottmann, Nico Nachtigall, Claudia Ermel, Frank Hermann, Thomas Engel	-
Triple Graph Grammars in the Large for Translating Satellite Procedures [HGN$^+$14]	Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, Claudia Ermel	Sec. 5.1
Solving the FIXML2Code-case Study with HenshinTGG [HNB$^+$14]	Frank Hermann, Nico Nachtigall, Benjamin Braatz, Thomas Engel, Susann Gottmann	-
Towards Domain Completeness for Model Transformations Based on Triple Graph Grammars [NHBE14]	Nico Nachtigall, Frank Hermann, Benjamin Braatz, Thomas Engel	Chap. 3
On an Automated Translation of Satellite Procedures Using Triple Graph Grammars [HGN$^+$13]	Frank Hermann, Susann Gottmann, Nico Nachtigall, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel	Sec. 5.1
Correctness and Completeness of Generalised Concurrent Model Synchronisation Based on Triple Graph Grammars [GHN$^+$13]	Susann Gottmann, Frank Hermann, Nico Nachtigall, Benjamin Braatz, Claudia Ermel, Hartmut Ehrig, Thomas Engel	Sec. 4.2
Symbolic Execution of Satellite Control Procedures in Graph-Transformation-Based EMF Ecosystems [NBE13]	Nico Nachtigall, Benjamin Braatz, Thomas Engel	Sec. 5.4
Transformation Systems with Incremental Negative Application Conditions [CHH$^+$12]	Andrea Corradini, Reiko Heckel, Frank Hermann, Susann Gottmann, Nico Nachtigall	-

Figure 1.5: List of Publications

graphs to a variety of different types of graphs (and other structures). Note that the main results have been developed with a focus to be applied in $\mathbf{AGraphs}_{ATGI}$. Furthermore, we review the notions of triple graphs, triple graph grammars (TGGs) and model transformations / synchronisations based on TGGs. Finally, we give an overview of important properties of model transformations and synchronisations with a focus to the completeness property.

1.4.2 Chapter 3: Domain Completeness

Chap. 3 introduces the domain completenes problem, shows its undecidability and presents an approximative solution to this problem. It is shown how to verify the domain completeness problem. Moreover, limitations of the verification approach are discussed and the concept of recursive graph constraints is introduced in order to allow an application of the approach in the context of infinite domain constraints. For example, infinite graph constraints are used when describing regular paths in graphs. Two methods are presented for deriving finite graph constraints from infinite graph constraints such that the verification process can terminate. Finally, we consider the verification of domain completeness under restrictions of the domain type graph. This reflects the situation where only a subset of all constituents of the given domain is subjected to the verification.

1.4.3 Chapter 4: Domain Completeness of Model Transformations & Model Synchronisations

In Chap. 4, we instantiate the results from Chap. 3 to the verification of domain completeness of model transformations and model synchronisations. We reformulate the completeness problem in the context of transformations and distinguish between forward and backward domain completeness of transformations and synchronisations.

1.4.4 Chapter 5: Further Applications

Chap. 5 presents further applications of the previous verification results for domain completeness. In particular, this includes the instantiation of the results to the verification of the completeness of software transformations and synchronisations. Furthermore, it is shown to which extend the concept of recursive graph constraints from Chap. 3 can be used for the verification of completeness in the context of expressing static semantics in models.

1.4.5 Chapter 6: Conclusion, Related & Future Work

In Chap. 6 we conclude and discuss related work and aspects of future work.

Chapter 2

Model Transformations, Model Synchronisations & Formal Framework

2.1 General Introduction to Model Transformations & Synchronisations

The initial introduction to model transformations and synchronisations from Sec. 1.2 is extended to a more detailed view in the following Sects. 2.1.1 and 2.1.2.

2.1.1 Model Transformation

Fig. 2.1 illustrates different types of model transformations. In general, a model transformation $MT\colon \mathscr{L}(D_1) \Rrightarrow \mathscr{L}(D_2)$ is a relation that maps (transforms) models $M \in \mathscr{L}(D_1)$ in a domain-specific modelling language (DSL) $\mathscr{L}(D_1)$ in source domain D_1 (in)to models $M' \in \mathscr{L}(D_2)$ in DSL $\mathscr{L}(D_2)$ in target domain D_2 [MG06]. Therefore, $(M, M') \in MT$ means that model M expressed in modelling language $\mathscr{L}(D_1)$ is transformed into model M' expressed in modelling language $\mathscr{L}(D_2)$ via model transformation MT. The model transformation is exogeneous if lanuage $\mathscr{L}(D_1)$ differs from langauge $\mathscr{L}(D_2)$ ($\mathscr{L}(D_1) \neq \mathscr{L}(D_2)$) and is endogeneous otherwise ($\mathscr{L}(D_1) = \mathscr{L}(D_2)$). Moreover, the model transformation is horizontal if the models in $\mathscr{L}(D_1)$ and the models in $\mathscr{L}(D_2)$ are representations of their originals at the same layer of abstraction. Otherwise, if the layer of abstraction of $\mathscr{L}(D_1)$ differs from the layer of abstraction of $\mathscr{L}(D_2)$, then the model transformation is vertical. According to [MG06], the following examples highlight the different types of transformations:

1. **Endogenous & horizontal:** Model refactoring or simplification, i.e., changing or simplifying the internal structure of a model in order to obtain a better readability, reusability, modularity or adaptability but without changing the meaning or behaviour of the model itself,

2. **Endogenous & vertical:** Model refinement (abstraction), e.g., by addding (removing) more concrete (platform-specific) details to (from) a model while staying in the same modelling language,

3. **Exogenous & horizontal:** Language migration, i.e., migration of models from one DSL to another modelling language but by staying at the same layer of abstraction, and

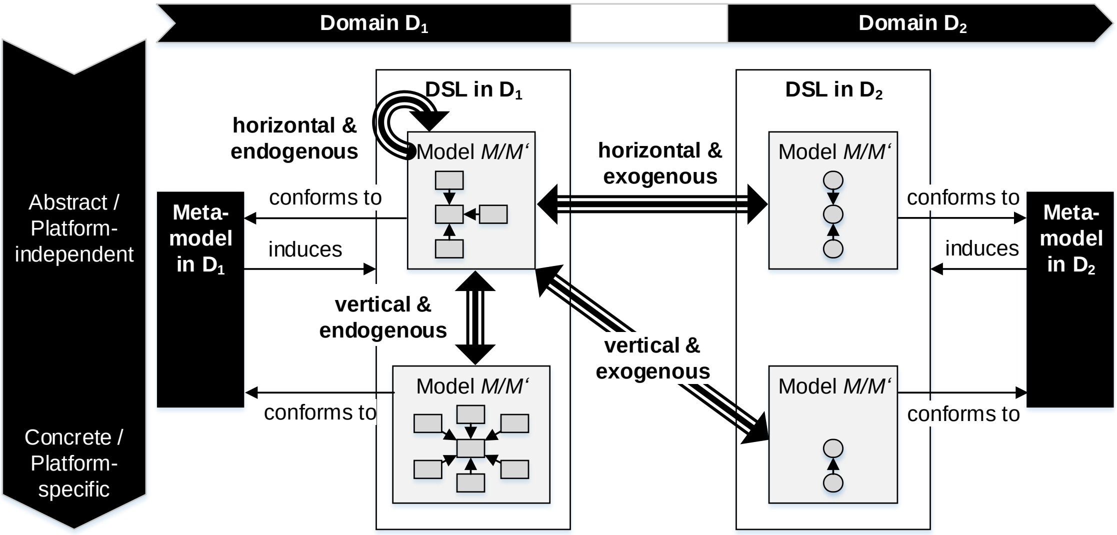


Figure 2.1: Classification of Model Transformations (Adaption from [EEGH15])

4. **Exogenous & vertical:** Code (model) generation, i.e., transforming visual models expressed in a more abstract (platform-independent) DSL into abstract syntax trees over a platform-specific programming language that are serialised to source code at a later step (or vice versa), e.g., the transformation from UML class diagrams into Java source code [CH06].

For example, the model transformations from UML statecharts into place/transition nets and from UML class diagrams into entity-relationship diagrams or vice versa in Sec. 1.2 are exogenous transformations. If we neglect details that are only prevailing in one domain, e.g., orthogonal regions in statecharts or operations in class diagrams, then these transformations may be considered as horizontal. Otherwise, they are rather vertical transformations, e.g., the additional knowledge of operations of classes in class diagrams together with their visibilities to system developers (+ for public, - for private etc., cf. Fig. 1.2) gives a more concrete and platform-specific view on the system compared to entity-relationship diagrams in this aspect. Transforming class diagrams by adding operations to classes in the diagrams is rather a model refinement, i.e., an endogenous and vertical model transformation. In contrast, a simple renaming of class or attribute names is an endogenous and horizontal transformation. In the literature, endogenous model transformations are associated with model rephrasing and exogenous transformations are associated with model translations.

Furthermore, DSLs $\mathscr{L}(D_1)$ and $\mathscr{L}(D_2)$ are each defined by a meta-model in the corresponding domain, i.e., $\mathscr{L}(D_1)$ ($\mathscr{L}(D_2)$) is induced by a meta-model in domain D_1 (D_2) (cf. Fig. 2.1). The meta-model defines the concepts of the domain and their interrelationships that can be used in models. In turn, $\mathscr{L}(D_1)$ ($\mathscr{L}(D_2)$) is the set of all models that conform to the meta-model in D_1 (D_2), i.e., the models only contain the concepts and interrelationships of the meta-model as model elements. For example, we consider the DSL of entity-relationship diagrams in the domain where we want to reflect over the different entities of a system and their interrelationships. The modelling language is defined by the meta-model in Fig. 2.2. Entity-relationship diagrams may contain several Entities and directed Relationships between them, i.e., each relationship is directed from an entity to an entity. Moreover, entities and relationships may have several Attributes that may be primary keys (isPrimary). By inheritance, each entity, relationship and attribute is a NamedModelElement and therefore, has a name of type String. Note that the class

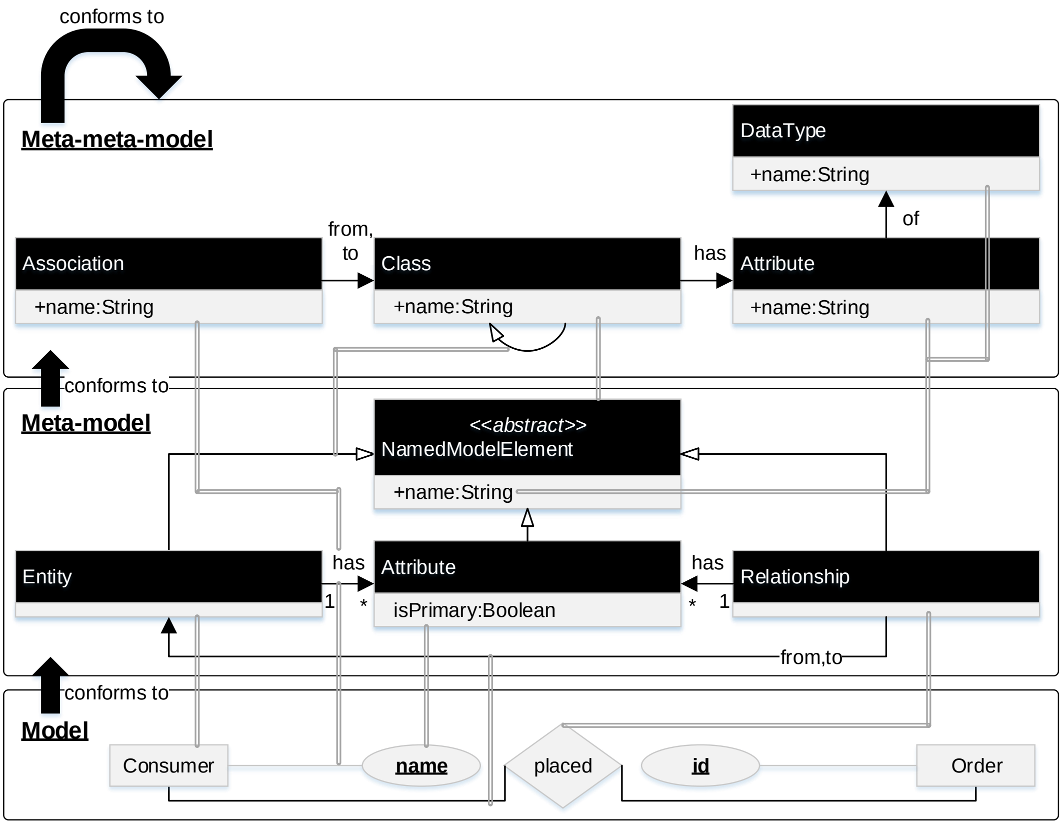


Figure 2.2: Different Layers of Meta-Modelling

NamedModelElement is declared as << abstract >>, i.e., entity-relationship diagrams are not allowed to directly contain nodes of type NamedModelElement but only in the form of entities, relationships and attributes. Therefore, the language of entity-relationship diagrams is given by all diagrams (models) that conform to the meta-model. For example, the model in Fig. 2.2 conforms to the meta-model. Consumer and Order are Entities whereas placed is a Relationship from Consumer to Order. Entity Consumer (Order) has a name (id) as Attribute which is marked as primary key (isPrimary=**true**).

This view can also be applied to the meta-model where the language of meta-models is defined by a meta-meta-model. Most meta-models can be expressed by means of the concepts of class diagrams that are given in excerpts by the meta-meta-model in Fig. 2.2. Each meta-model may contain several Classes and Associations from classes to classes. Furthermore, each class may have several Attributes of some DataType and with some name of type String. Between two classes there may be an inheritance relationship. For example, the meta-model in Fig. 2.2 conforms to the meta-meta-model. Entity, Attribute and Relationship are Classes that inherit from class NamedModelElement. From class Entity to class Attribute there is an Association with name has. The same is true for class Relationship and Attribute. The class Attribute has an Attribute with name isPrimary of DataType Boolean. Analogously, class NamedModelElement has an attribute of name name and of type String. In turn, the language of meta-meta-models can be defined by a meta^3-model such that the meta-meta-models conform to the meta^3-model etc.. However, in order to have a closure over the "conforms to" relation, OMG proposes a three-layer hierarchy of meta-modelling, i.e., models, meta-models and meta-meta model, where the meta-meta-model is defined by (conforms to) itself (cf. Fig. 6 in [HSGP13]). Note that for the

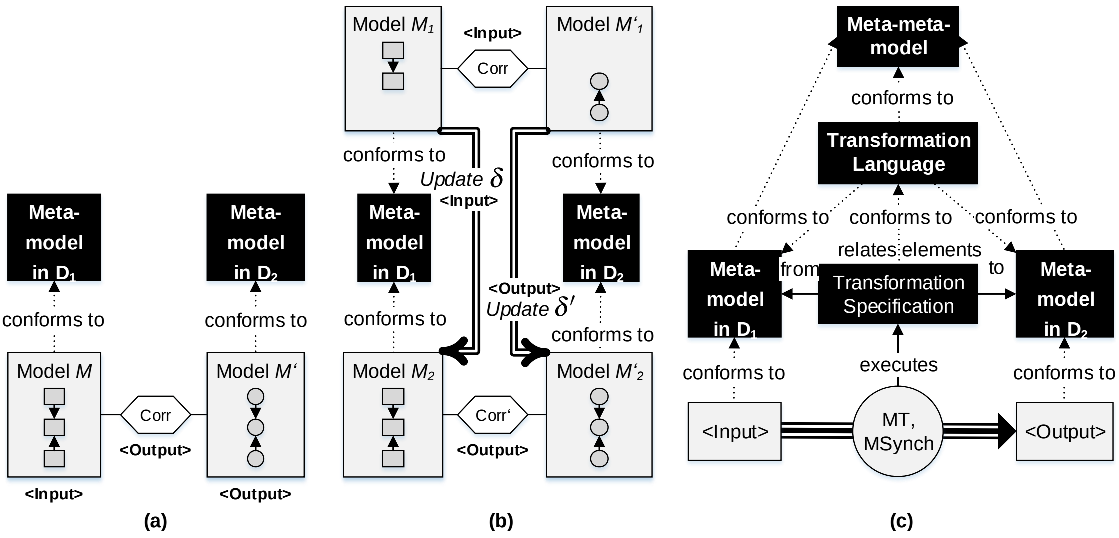


Figure 2.3: Setting of Model Transformations (a), Model Synchronisations (b) & their Execution (c) (Adaption from [EEGH15, LAD$^+$14])

language of class diagrams, the meta-meta-model may directly serve as the meta-model which leads to only two layers of meta-modelling.

Fig. 2.3 (c) illustrates the execution of a model transformation that takes a source model $M \in \mathcal{L}(D_1)$ over the meta-model in source domain D_1 as input and outputs a target model $M' \in \mathcal{L}(D_2)$ over the meta-model in target domain D_2 together with a correspondence (Corr) which relates elements from M to M' as depicted in Fig. 2.3 (a). The model transformation (MT) is performed by executing the model transformation specification. The specification is defined based on the two meta-models in domains D_1 and D_2 and defines which model elements in D_1 should be transformed to which model elements in D_2 by relating (mapping) model elements from D_1 to D_2. The transformation specification is expressed in (conforms to) a transformation language. The transformation language contains (allows to express) the set of all conceivable mappings of model elements from D_1 to D_2. Therefore, the transformation language conforms to the meta-models in domains D_1 and D_2 which in turn conform to a common meta-meta-model (cf. Fig. 2.2), i.e., the transformation language indirectly conforms to this meta-meta-model. Although, both meta-models may conform to different meta-meta-models in general, the assumption of a common meta-meta-model of class diagrams is valid as already discussed.

Different model transformation approaches and tools [HLH$^+$13, RHM$^+$14] do exist. Beside model transformation by-example [KLR$^+$12] where the transformation specification is generated from a given source model, target model and their correspondence, different types of transformation languages exist for explicitly specifying the transformation [GdLK$^+$13], e.g., QVT [QVT15], ATL [JABK08], triple graph grammars (TGGs) [ALS15] and ETL [KPP08]. Different types of transformation languages may allow transformation specifications in textual or visual form and in declarative or operational manner [Men13]. While declarative transformation specifications focus on the specific mapping of model elements between domains, operational specifications additionally provide the concrete steps how the target model is derived from a source model, e.g., by providing the concrete order of traversing the model elements of the source model. Furthermore, we distinguish between in-place model transformations and "external" transformations where the source model is preserved by the transformation while the target model is constructed in parallel. Moreover, model-to-model transformations are distinguished from model-to-text, text-to-model and text-to-text transformations. However, note that any of these types of transformations can be simulated by "external" model-to-model transformations where the parallel target model is taken as output of an in-place transformation and text is parsed to (serialised from) a model before (after) executing the model-to-model transformation. Therefore, we focus on "external" model-to-model transformations based on TGGs that allow visual, declarative transformation specifications. We review basic concepts and notions in Sects. 2.3.1 and 2.3.2. Beside the model-to-model transformation CD2RDBM from UML class diagrams to relational database models in 2.3.2, we present software transformations, i.e., text-to-(text)model transformations, from source code to visual models (or source code again) in Sec. 5.1.

2.1.2 Model Synchronisation

Basically, model synchronisations $MSynch\colon \Delta_{D_1} \to \Delta_{D_2}$ are performed based on model transformations where model updates $\delta \in \Delta_{D_1}$ in domain D_1 are propagated (mapped) to (model updates $\delta' \in \Delta_{D_2}$ in) domain D_2 by performing model transformations. Therefore, the classification of model transformations from Sec. 2.1.1 can also be applied to model synchronisations, i.e., model synchronisations may be 1. endogenous or exogenous, 2. horizontal or vertical, 3. based on textual or visual transformation specifications, 4. based on declarative or operational transformation specifications, and 5. propagations of updates from model-to-model, model-to-text, text-to-model or text-to-text.

According to Fig. 2.3 (b), a model update in some domain D relates model $M_1 \in \mathscr{L}(D)$ from before the update with model $M_2 \in \mathscr{L}(D)$ from after performing the update and therefore, the update documents the changes to M_1 which have led to M_2. Thus, a model update relates models that are expressed in the same DSL $\mathscr{L}(D)$, i.e., models M_1 and M_2 conform to the same meta-model in domain D. Fig. 2.3 (c) illustrates the execution of a model synchronisation that takes a model update δ in source domain D_1 from model M_1 to model M_2 as input together with a correspondence (Corr) which interrelates model M_1 with model M'_1 in target domain D_2 to which the update should be propagated. The execution outputs an update δ' in target domain D_2 from model M'_1 to model M'_2 together with a correspondence (Corr') which relates model M_2 with model M'_2. The model synchronisation (MSynch) is performed based on executing the model transformation specification.

According to Sec. 2.1.1, we focus on model-to-model synchronisations based on TGGs that allow visual, declarative transformation specifications. We review basic concepts and notions in Sects. 2.3.1 and 2.3.3. Beside the model-to-model propagation of model updates from UML class diagrams to relational database models in Sec. 2.3.3, we present software synchronisations, i.e., text-to-model(text) propagations of updates from source code to visual models (or source code again) in Sec. 5.2.

2.2 Graph Transformation

We recall basic definitions and results of the theory of algebraic graph transformation from [EEPT06, EEGH15]. In particular this includes the category $\mathbf{AGraphs}_{ATGI}$ of typed attributed graphs with node type inheritance from [GLEO12, EEGH15] in Sec. 2.2.1, their transformation and the generalisation to $\mathscr{M}$-adhesive transformation systems from [EEGH15] in Sec. 2.2.4. Furthermore, we review the notions of (nested) graph conditions and constraints as well as their interpretation via AC-schemata from [HP09, EEGH15] in Sec. 2.2.3. Moreover, in Prop. 2.1 we show that the direct interpretation of type strict conditions coincides with their interpretation via AC-schemata when restricting to $\mathscr{M}$-matches. For the general case, in Prop. 2.2 we show that

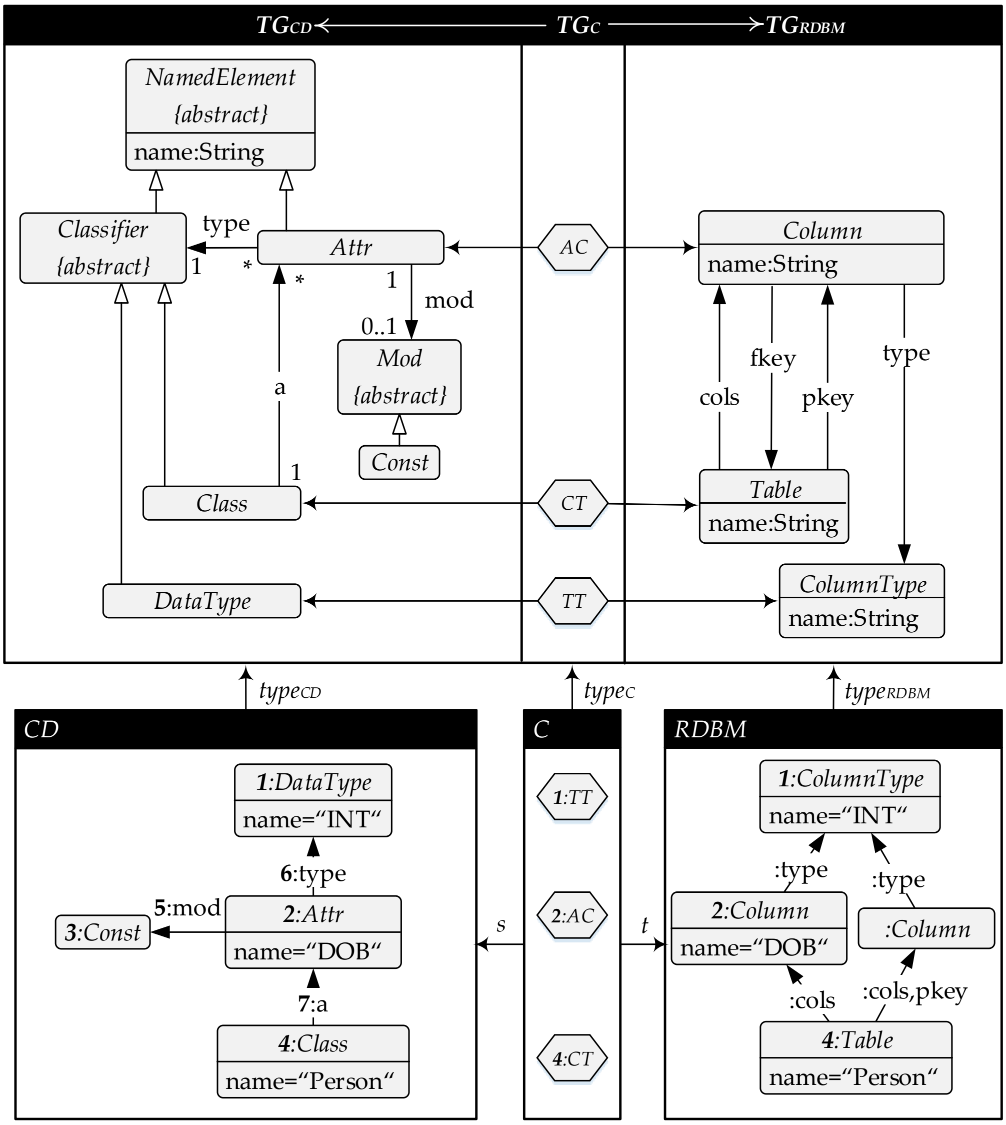


Figure 2.4: Attributed Triple Type Graph ($TG_{CD} \leftarrow TG_C \rightarrow TG_{RDBM}$) (top) & Typed Attributed Triple Graph ($CD \xleftarrow{s} C \xrightarrow{t} RDBM$) (bottom)

the standard satisfiability of an AC-schema coincides with the $\mathcal{O}$-satisfiability of the underlying condition. This allows an interpretation of conditions via $\mathcal{O}$-matches and $\mathcal{O}$-satisfiability from a user point of view while an interpretation via the standard satisfiability of AC-schemata with $\mathcal{M}$-matches is used to prove technical results.

2.2.1 Graphs, Typed & Attributed Graphs

We assume that (visual) models are represented by graphs (cf. Sec. 1.3). A plain graph consists of nodes (vertices) and edges between nodes. An edge links a source node with a target node. The presented notion of graphs allows parallel edges between two nodes and edge loops. A morphism $f\colon G_1 \to G_2$ from a graph G_1 to a graph G_2 is a mapping from nodes and edges in G_1 to nodes and edges in G_2 such that the structure of G_1 is preserved, i.e, the source (target) node of each edge e in G_1 is mapped to the source (target) node of edge $f(e)$ in G_2.

Definition 2.1 (Graph and Graph Morphism (Def. 2.1 in [EEGH15])) A *graph* $G = (V_G, E_G, s_G\colon E_G \to V_G, t_G\colon E_G \to V_G)$ consists of a set of nodes V_G, a set of edges E_G and two functions s_G, t_G that map the source node (via s_G) and target node (via t_G) to each edge. A *graph morphism* $f\colon G_1 \to G_2$ from G_1 to G_2 with $f = (f_V, f_E)$ consists of two functions $f_V\colon V_{G_1} \to V_{G_2}, f_E\colon E_{G_1} \to E_{G_2}$ such that $s_{G_2} \circ f_E = f_V \circ s_{G_1}$ and $t_{G_2} \circ f_E = f_V \circ t_{G_1}$. △

A typed graph is a plain graph G together with a graph morphism from G to a distinguished type graph TG that defines the typing of each node and edge in G - We say that G is typed over TG. Therefore, a type graph is part of the meta-model for a given domain and defines the concepts (node types) for the domain and their interrelationships (edge types) that can be used in graphs as models in the domain (cf. Sec. 2.1.1 and Fig. 2.2). A typed graph morphism between two typed graphs is a graph morphism that additionally preserves the typing.

Definition 2.2 (Typed Graph and Typed Graph Morphism (Def. 2.2 in [EEGH15])) Given a distinguished graph TG as the type graph. A *typed graph* $G^{\mathrm{T}} = (G, type_G\colon G \to TG)$ *over* TG is given by a graph G and a graph morphism $type_G$ from G to TG. A *typed graph morphism* $f\colon G_1^{\mathrm{T}} \to G_2^{\mathrm{T}}$ is a graph morphism $f\colon G_1 \to G_2$ such that $type_{G_2} \circ f = type_{G_1}$. △

$$\begin{tikzcd}
G_1 \arrow[rr, "f"] \arrow[dr, "type_{G_1}"'] & (=) & G_2 \arrow[dl, "type_{G_2}"] \\
& TG &
\end{tikzcd}$$

Attributed graphs are defined based on the notion of E-graphs that allow the attribution of edges and nodes. The set of possible attribute values is defined by an algebra. For an introduction to algebraic signatures and algebras we refer to [EEPT06].

Definition 2.3 (Attributed Graph and Attributed Graph Morphism (Def. 2.4 in [EEGH15])) An *E-graph* $G^{\mathrm{E}} = (V_G^G, V_D^G, E_G^G, E_{NA}^G, E_{EA}^G, (s_i^G, t_i^G)_{i \in \{G,NA,EA\}})$ with graph nodes V_G^G, data nodes V_D^G, graph edges E_G^G, node attribute edges E_{NA}^G, edge attribute edges E_{EA}^G and source and target functions s_i^G, t_i^G with signatures as defined on the right. An *E-graph morphism* $f\colon G_1^{\mathrm{E}} \to G_2^{\mathrm{E}}$ with $f = ((f_{V_i}\colon V_i^{G_1} \to V_i^{G_2})_{i \in \{G,D\}}, (f_{E_j}\colon E_j^{G_1} \to E_j^{G_2})_{j \in \{G,NA,EA\}})$ is a pair of tuples of functions f_{V_i} for mapping nodes and functions f_{E_j} for mapping edges from G_1^{E} to G_2^{E} such that f commutes with all source and target functions. Given a data signature $DSIG = (S, OP)$, then an *attributed graph over DSIG* is given by $G = (G^{\mathrm{E}}, D_G)$ with G^{E} being an E-graph and D_G being a $DSIG$-algebra such that $\dot{\cup}_{s \in S}(D_{G,s}) = V_D^G$. Given attributed graphs G_1 and G_2 over common $DSIG$, then an *attributed graph morphism* $f\colon G_1 \to G_2$ from G_1 to G_2 with $f = (f_G, f_D)$ is a pair of an E-graph morphism $f_G\colon G_1^{\mathrm{E}} \to G_2^{\mathrm{E}}$ and an algebra homomorphism $f_D\colon D_{G_1} \to D_{G_2}$ such that $f_{G,V_D}(x) = f_{D,s}(x)$ for all $x \in D_{G_1,s}, s \in S$. With $(V_G^G, (E_X^G)_{X \in \{G,NA,EA\}})$ we refer to the (structural) graph part of attributed graph G and distinguish it from its data part (V_D^G, D_G). With $f_S = (f_{G,V_G}, f_{G,E_G}, f_{G,E_{NA}}, f_{G,E_{EA}})$ we refer to the (structural) graph part of attributed graph morphism f and distinguish it from its data part f_D. △

$$\begin{tikzcd}
& E_G^G \arrow[rr, bend left, "s_G^G"] \arrow[rr, bend right, "t_G^G"'] & & V_G^G & \\
E_{EA}^G \arrow[ur, "s_{EA}^G"] \arrow[rr, "t_{EA}^G"] & & V_D^G & & E_{NA}^G \arrow[ul, "s_{NA}^G"'] \arrow[ll, "t_{NA}^G"']
\end{tikzcd}$$

Typed and attributed graphs (morphisms) are combined to typed attributed graphs (morphisms).

Definition 2.4 (Typed Attributed Graph and Morphism (Def. 2.5 in [EEGH15])) Given a distinguished attributed graph $ATG = (TG, Z)$ as attributed type graph with Z being the final $DSIG$-algebra. The final $DSIG$-algebra contains exactly one element in each carrier set. A *typed attributed graph* $G^{\mathrm{T}} = (G, type_G\colon G \to ATG)$ *over* ATG is given by an attributed graph G over

DSIG and an attributed graph morphism $type_G$ from G to ATG. Given typed attributed graphs G_1^{T} and G_2^{T} over common ATG, then a *typed attributed graph morphism* $f\colon G_1^{\mathrm{T}} \to G_2^{\mathrm{T}}$ is an attributed graph morphism $f\colon G_1 \to G_2$ such that $type_{G_2} \circ f = type_{G_1}$. △

Remark 2.1 (Typed Attributed Graphs with Node Type Inheritance) *Typed attributed graphs and morphisms are extended to typed attributed graphs and morphisms with node type inheritance. Therefore, the type graph is extended with an additional inheritance relation between nodes that defines which nodes in the type graph inherit from which other nodes. Furthermore, nodes in the type graph can be explicitly marked as abstract, i.e., abstract nodes cannot be directly used as concrete types for nodes in graphs that are typed over this type graph. A node* A *that inherits from a node* B *shares all node attributes as well as incoming and outgoing edges of* B *with* B. *If node* A *inherits from node* B, *then* A *is called the sub-node (sub-type) of* B *whereas* B *is called the super-node (super-type) of* A. *A morphism between two typed attributed graphs G_1 and G_2 with node type inheritance may refine the types of nodes from G_1 to G_2 by mapping nodes in G_1 of super-type* B *to nodes in G_2 of sub-type* A. *Basically beside abstract nodes, typed graphs with node type inheritance do not lead to additional expressiveness in comparison to graphs without node type inheritance, since, the type graph with inheritance can most widely be flattened to a type graph without inheritance information by duplicating edges and attributes from super-nodes to sub-nodes (cf. Def. 6 in [GLEO12]). However, type graphs with inheritance allow a more compact notation in comparison to their flattened versions (cf. Figs. 1 and 4 in [GLEO12]). We allow inheritance in type graphs and refer to [GLEO12] for technical details.* △

Example 2.1 (Attributed Type Graph & Typed Attributed Graph) *Fig. 2.4 depicts attributed type graphs TG_{CD} and TG_{RDBM} for class diagrams (CD) and relational database models (RDBM). Both type graphs are part of the meta-models for the domains of CDs and RDBMs. Class diagrams may contain several* Classes*, each with a set of class* Attr*ibutes that are assigned via* a *edges to the class. Furthermore, each attribute is* typed by a Classifier. *By node type inheritance, classifiers are classes or other* DataTypes. *Furthermore by node type inheritance, each class, attribute and data type is a* NamedElement *and therefore has a specific* name *of type* String. *Moreover, attributes may have a* Modifier Constant *which declares that the attribute value does not change. Note that nodes* NamedElements, Classifiers *and* Modifiers *are marked as* abstract *and therefore, they cannot be directly used as concrete types in class diagrams but their sub-nodes (sub-types).*

Relational database models may contain several Tables. *A table has a* name *of type* String *and may have several* Columns *assigned via* cols *edges where one column may contain the primary keys of the rows of the table (edge* pkey*) or columns may refer to other tables by foreign keys (*fkey*). Furthermore, each column has a specific* name *of type* String *and is of a specific* type. *Analogously, each column type has a* name *of type* String.

The multiplicity constraints on the edges in type graph TG_{CD} are expressed by graph constraints in Sec. 2.2.3 and Ex. 2.3 and complete the meta-model for the domain of UML class diagrams.

Typed attributed graph CD (RDBM) is a class diagram (relational database model) typed over type graph TG_{CD} (TG_{RDBM}) via morphism $type_{CD}$ ($type_{RDBM}$). The class diagram CD contains a class of name “Person″ *together with a class attributes* “DOB″ *(date of birth) of type* “INT″. *Furthermore, attribute* “DOB″ *is equipped with modifier* Const *(constant). For typing, node* 2 *in CD is mapped to node* Attr *in TG_{CD} along morphism $type_{CD}$. All other nodes, edges and node attributes in CD are mapped analogously. The formal notation of graph CD is given below:*

$CD = (\underline{CD}, type_{\underline{CD}} \colon \underline{CD} \to TG_{CD})$ *where* $\underline{CD} = (\underline{CD}^E = (V_G^{\underline{CD}}, V_D^{\underline{CD}}, E_G^{\underline{CD}}, E_{NA}^{\underline{CD}}, E_{EA}^{\underline{CD}}, (s_i^{\underline{CD}}, t_i^{\underline{CD}})_{i \in \{G,NA,EA\}}), D_{\underline{CD}})$ *with*
$V_G^{\underline{CD}} = \{1,2,3,4\}$,
$V_D^{\underline{CD}} = D_{\underline{CD},String}$,
$E_G^{\underline{CD}} = \{5,6,7\}$,
$E_{NA}^{\underline{CD}} = \{a,b,c\}$,
$E_{EA}^{\underline{CD}} = \{\}$,
$s_G^{\underline{CD}} = (5 \mapsto 2, 6 \mapsto 2, 7 \mapsto 4)$,
$t_G^{\underline{CD}} = (5 \mapsto 3, 6 \mapsto 1, 7 \mapsto 2)$,
$s_{NA}^{\underline{CD}} = (a \mapsto 1, b \mapsto 2, c \mapsto 4)$,
$t_{NA}^{\underline{CD}} = (a \mapsto \text{“}INT'', b \mapsto \text{“}DOB'', c \mapsto \text{“}Person'')$,
$s_{EA}^{\underline{CD}} = t_{EA}^{\underline{CD}} = \varnothing$,
$D_{\underline{CD}} = (D_{\underline{CD},String} = \{w^* \mid w \in \{a..z, A..Z\}\}, OP_{D_{\underline{CD}}} = \varnothing)$ *being a* $DSIG$*-algebra for algebraic data signature* $DSIG = (S = \{String\}, OP = \varnothing)$, *and*
$type_{\underline{CD}} = (type_{\underline{CD},G} = (type_{\underline{CD},V_G}, type_{\underline{CD},V_D}, type_{\underline{CD},E_G}, type_{\underline{CD},E_{NA}}, type_{\underline{CD},E_{EA}}), type_{\underline{CD},D} \colon D_{\underline{CD}} \to D_{TG_{CD}})$ *being the type morphism with*
$type_{\underline{CD},V_G} = (1 \mapsto DataType, 2 \mapsto Attr, 3 \mapsto Const, 4 \mapsto Class)$,
$type_{\underline{CD},V_D}(x) = type_{\underline{CD},D,String}(x), \forall x \in D_{\underline{CD},String}$,
$type_{\underline{CD},E_G} = (5 \mapsto mod, 6 \mapsto type, 7 \mapsto a)$,
$type_{\underline{CD},E_{NA}} = (a,b,c \mapsto name)$,
$type_{\underline{CD},E_{EA}} = \varnothing$, *and*
$type_{\underline{CD},D,String}(x) = String, \forall x \in D_{\underline{CD},String}$ *for final* $DSIG$*-algebra* $D_{TG_{CD}} = (D_{TG_{CD},String} = \{String\}, OP_{D_{TG_{CD}}} = \varnothing)$ *of attributed type graph* TG_{CD}.

The relational database model RDBM contains a corresponding Table, Column *and* ColumnType *for each* Class, Attr*ibute and* DataType *of class diagram CD. The formal notation of typed attributed graph RDBM is analogously to CD.* △

Visual Notation As depicted in Fig. 2.4, nodes (edges) in typed graphs are visualised by x : y with x being the name of the node (edge) and y being the type of the node (edge). The mapping of nodes and edges along morphisms correspond to their naming in visual notation, e.g., in visualisations of graph conditions, graph transformation rules, triple graphs and model updates (cf. Sects. 2.2.3, 2.2.4, 2.3.1 and 2.3.3). We omit node and edge names in visualisations if they are irrelevant and write : y instead of x : y. Note that in formal notation of attributed graphs, node (and edge) attributes are edges and attribute values are nodes. However, in visual notation we write attr = x for attribute attr with value x directly in the corresponding node (or edge) shape. For an explicit visualisation of attributed graphs in E-graph notation we refer to Ex. 8.5 in [EEPT06]. Although technically, a node or edge may have the same attribute several times (with possibly different attribute values), usually in practice, each node and edge has each attribute at most once such that the mapping of attributes and their values along morphisms is clear and is not explicitly visualised.

2.2.2 $\mathscr{M}$-adhesive Categories

In the following, we review the notion of $\mathscr{M}$-adhesive categories as a generalisation of the category of typed attributed graphs. For a short introduction to category theory we refer to [EEPT06] and for a more detailed view we refer to [EM90, AHS90].

Remark 2.2 (Basic Notions of Category Theory & Category $\mathbf{AGraphs}_{ATGI}$) A category $\mathbf{C} = (Ob_{\mathbf{C}}, Mor_{\mathbf{C}}, \circ, id)$ *is defined by a class* $Ob_{\mathbf{C}}$ *of objects, a set* $Mor_{\mathbf{C}}$ *of morphisms* $f\colon A \to B$ *between objects* $A,B \in Ob_{\mathbf{C}}$*, for all objects* $A,B,C \in Ob_{\mathbf{C}}$ *and morphisms* $f\colon A \to B, g\colon B \to C \in Mor_{\mathbf{C}}$ *a composition* $g \circ f \in Mor$*, and for each object* $A \in Ob_{\mathbf{C}}$ *an identity morphism* $id_A\colon A \to A \in Mor_{\mathbf{C}}$ *such that "Associativity:" For all objects* $A,B,C,D \in Ob_{\mathbf{C}}$ *and morphisms* $f\colon A \to B, g\colon B \to C, h\colon C \to D \in Mor_{\mathbf{C}}$ *it holds that* $(h \circ g) \circ f = h \circ (g \circ f)$*, and "Identity:" For all objects* $A,B \in Ob_{\mathbf{C}}$ *and morphisms* $f\colon A \to B \in Mor_{\mathbf{C}}$ *it holds that* $f \circ id_A = f$ *and* $id_B \circ f = f$ *(cf. Def. A.1 in [EEPT06]). A functor* $F\colon \mathbf{C} \to \mathbf{D}$ *is a mapping from objects and morphisms of category* $\mathbf{C}$ *to objects and morphisms of category* $\mathbf{D}$ *which is compatible with composition and the identities (cf. A.6 in [EEPT06]). Inclusions* $i\colon A \to B$ *are morphisms with* $i(A) = A$*. With* mono- epi- and iso-morphisms *we denote special types of morphisms in categories* $\mathbf{C}$*. Intuitively, an isomorphism is a morphism between two objects of the same structure that additionally preserves this structure. According to Def. A.9 in [EEPT06], morphism* $f\colon A \to B \in Mor_{\mathbf{C}}$ *is an isomorphism, if there exists an inverse morphism* $f^{-1}\colon B \to A \in Mor_{\mathbf{C}}$ *such that* $f^{-1} \circ f = id_A$ *and* $f \circ f^{-1} = id_B$*. In this context, the inverse morphism* f^{-1} *is unique and also an isomorphism (cf. Rem. A.10 in [EEPT06]) and the composition* $i_2 \circ i_1$ *of two isomorphisms* i_1, i_2 *is again an isomorphism. We write* $G \cong G'$ *and say that* G *is isomorphic to* G'*, if there exists an isomorphism* $i\colon G \to G'$*. According to Def. A.12 in [EEPT06], mono- and epi-morphisms are defined as follows. A morphism* $h\colon B \to C \in Mor_{\mathbf{C}}$ *is a monomorphism, if for all morphisms* $f,g\colon A \to B \in Mor_{\mathbf{C}}$ *it holds that* $h \circ f = h \circ g$ *implies* $f = g$*. Conversely, a morphism* $f\colon A \to B \in Mor_{\mathbf{C}}$ *is an epimorphism, if for all morphisms* $g,h\colon B \to C \in Mor_{\mathbf{C}}$ *it holds that* $g \circ f = h \circ f$ *implies* $g = h$*. Note that an isomorphism is both an epi- and monomorphism but a morphism that is an epi- and monomorphism must not be an isomorphism in general, since, the inverse morphism may not exist. According to Def. A.16 in [EEGH15], a morphism pair* $(f_1\colon A_1 \to B, f_2\colon A_2 \to B)$ *is jointly epimorphic, if for all morphisms* $g,h\colon B \to C$ *it holds that* $g \circ f_i = h \circ f_i$ *for* $i = 1,2$ *implies* $g = h$*. Given the category* $\mathbf{AGraphs}_{ATGI}$ *with all typed attributed graphs over attributed type graph ATGI with node type inheritance as objects and all typed attributed graph morphisms between them as morphisms where furthermore, the identities are given by the componentwise identities on nodes, edges, attributes and algebras and the composition* $g \circ f$ *is given by* $g(f(x))$ *componentwise for all nodes, edges, attributes and elements* x *of carrier sets in the corresponding algebra (cf. Sec. 2.2.1 and Def. 2.4). Then, the monomorphisms are exactly those morphisms that are componentwise injective, the epimorphisms are exactly those morphisms that are componentwise surjective and isomorphisms are exactly those morphisms that are componentwise bijective (i.e., both injective and surjective) (cf. Fact 2.15 in [EEPT06]). The jointly epimorphic pairs of morphisms are exactly those pairs that are together surjective. Basic constructions in categories are pushouts and pullbacks. A pushout* $B +_A C$ *is the gluing of objects* B,C *via common sub-object* A*. In* $\mathbf{AGraphs}_{ATGI}$*, if* $A = \varnothing$ *is the empty graph* $\varnothing$*, then* $B +_A C$ *is the componentwise disjoint union of graphs* B *and* C*. In contrast, a pullback is the intersection of objects* B *and* C *via common object* D*.*

$$\begin{tikzcd}
X \arrow[dr, "x"] \arrow[drr, "h_X"] \arrow[ddr, "k_X"'] & & & \\
& A \arrow[r, "f"] \arrow[d, "g"] \arrow[dr, phantom, "(1)"] & B \arrow[d, "g'"] \arrow[ddr, "h_Y"] & \\
& C \arrow[r, "f'"] \arrow[drr, "k_Y"'] & D \arrow[dr, "y"] & \\
& & & Y
\end{tikzcd}$$

A pushout (PO) (1) or (f',g') over morphisms (f,g), *written* $B+_A C$, *is defined by 1. a pushout object* D, *and 2. morphisms* f',g' *with* $f' \circ g = g' \circ f$, *such that the following universal property is fulfilled: for all objects* Y *and morphisms* h_Y, k_Y *with* $k_Y \circ g = h_Y \circ f$, *there is a unique morphism* $y\colon D \to Y$ *such that* $y \circ g' = h_Y$ *and* $y \circ f' = k_Y$ *(cf. Def. A.17 in [EEPT06]). Note that the pushout object is unique up to isomorphism (cf. Rem. A.18 in [EEPT06]) and furthermore,* (f',g') *is jointly epimorphic. According to Def. A.20 in [EEPT06], for pushout* (1), (f',g) is called the pushout complement over (f,g'). *The "smallest" pushout (1) with* $f, f' \in \mathscr{M}$ *for a given morphism* g' *is called initial pushout for* g'*. In categories of graphs, the initial pushout (1) is the smallest pushout for* g' *in the sense that boundary graph* A *only contains those graph elements of* B *that are necessary to glue* B *and context graph* C *via common* A *to* D*. For technical details we refer to Def. 4.23, item 4 in [EEGH15]. The pushout complement over* (h_X, g') *exists if and only if for the initial pushout (1) for* g'*, there is a morphism* $b^*\colon A \to X$ *such that* $h_X \circ b^* = f$ *(cf. Thm. 6.4 in [EEPT06]).* A pullback (PB) (1) or (f,g) over morphisms (f',g') *is defined by 1. a pullback object* A*, and 2. morphisms* f,g *with* $g' \circ f = f' \circ g$*, such that the following universal property is fulfilled: for all objects* X *and morphisms* h_X, k_X *with* $f' \circ k_X = g' \circ h_X$*, there is a unique morphism* $x\colon X \to A$ *such that* $f \circ x = h_X$ *and* $g \circ x = k_X$ *(cf. Def. A.22 in [EEPT06]). With diagram* (1) *commutes we mean that* $g' \circ f = f' \circ g$*. With* $(1)+(2)$ *we denote the composition of two adjacent diagrams.* △

$\mathscr{M}$-adhesive categories are defined based on the following properties. For $\mathscr{M}$-morphisms m, we write $m \in \mathscr{M}$ and say that m is in $\mathscr{M}$. For several $\mathscr{M}$-morphisms $m_1, \ldots, m_n$ we write $m_1, \ldots, m_n \in \mathscr{M}$.

Definition 2.5 (PO-PB compatibility (Def. 4.2 & Rem. 4.3 in [EEGH15])) A morphism class $\mathscr{M}$ in a category $\mathbf{C}$ is called *PO-PB compatible* if

1. $\mathscr{M}$ is a class of monomorphisms, contains all identities (and isomorphisms), is closed under composition ($\mathscr{M}$-composition), i.e., $(f\colon A \to B \in \mathscr{M}, g\colon B \to C \in \mathscr{M} \implies g \circ f \in \mathscr{M})$, and is closed under decomposition ($\mathscr{M}$-decomposition), i.e., $g \circ f \in \mathscr{M}, g \in \mathscr{M}$ implies $f \in \mathscr{M}$.

2. $\mathbf{C}$ has pushouts and pullbacks along $\mathscr{M}$-morphisms (i.e., if $f \in \mathscr{M}$ or $g \in \mathscr{M}$ ($f' \in \mathscr{M}$ or $g' \in \mathscr{M}$), then there is a pushout (pullback) (1)), and $\mathscr{M}$-morphisms are closed under pushouts and pullbacks (i.e., $\mathscr{M}$-morphisms are preserved by pushouts and pullbacks - for pushout (pullback) (1), if $f \in \mathscr{M}$ ($f' \in \mathscr{M}$), then $f' \in \mathscr{M}$ ($f \in \mathscr{M}$)).

△

For $\mathscr{M}$-van Kampen squares, we refer to Def. 4.1 in [EEGH15].

Definition 2.6 ($\mathscr{M}$-adhesive Category (Def. 4.4 in [EEGH15])) A category $\mathbf{C}$ with a PO-PB compatible morphism class $\mathscr{M}$ is called an *$\mathscr{M}$-adhesive category* $(\mathbf{C}, \mathscr{M})$ if pushouts in $\mathbf{C}$ along $\mathscr{M}$-morphisms are $\mathscr{M}$-van Kampen squares. △

In addition to the properties in Defs. 2.5 and 2.6, the following basic HLR properties hold for $\mathscr{M}$-adhesive categories (we only list those basic HLR properties that are used in proofs of results of this thesis - For a complete list we refer to Def. 4.21 in [EEGH15]).

Definition 2.7 (Basic HLR properties (Thm. 4.22 in [EEGH15])) Given an $\mathcal{M}$-adhesive category $(\mathbf{C},\mathcal{M})$, then the following properties are valid:

$$\begin{tikzcd}
A \arrow[r, "k"] \arrow[d, "l"'] \arrow[dr, phantom, "(1)"] & B \arrow[r, "r"] \arrow[d, "s"] \arrow[dr, phantom, "(2)"] & E \arrow[d, "v"] \\
C \arrow[r, "u"'] & D \arrow[r, "w"'] & F
\end{tikzcd}$$

- Pushouts along $\mathcal{M}$-morphisms are pullbacks, i.e., given pushout (1) with $k \in \mathcal{M}$, then (1) is also a pullback.
- $\mathcal{M}$-pushout-pullback decomposition, i.e., given commuting (1) and (2) where $(1)+(2)$ is a pushout, (2) is a pullback, $w \in \mathcal{M}$, and ($l \in \mathcal{M}$ or $k \in \mathcal{M}$), then (1) and (2) are pushouts and also pullbacks.
- Uniqueness of pushout complements, i.e., given morphisms $k \in \mathcal{M}$ and s, then there is, up to isomorphism, at most one C with morphisms l,u such that (1) is a pushout. △

Remark 2.3 ($\mathcal{M}$- and $\mathcal{O}$-Morphisms in the $\mathcal{M}$-adhesive Category $(\mathbf{AGraphs}_{ATGI},\mathcal{M})$) *According to Thm. 6 in [GLEO12], category $(\mathbf{AGraphs}_{ATGI},\mathcal{M})$ is $\mathcal{M}$-adhesive with $\mathcal{M}$ being the class that consists of all typed attributed graph morphisms $f\colon G^T \to H^T$ that are componentwise injective, type strict (i.e., $type_H \circ f = type_G$) and where f_D is an isomorphism. Since $\mathcal{M}$-morphisms are type strict, they cannot refine the types of nodes from super- to sub-types along a node type inheritance relation (cf. Sec. 2.2.1 and Rem. 2.1). For a morphism $f\colon G \to H \in \mathcal{M}$, we say that G occurs in H or G is a sub-graph of H or H covers G. According to Def. 12 in [GLEO12, HEGO10] and Def. 7.3 in [EEGH15], $\mathcal{O}$-morphisms in $\mathbf{AGraphs}_{ATGI}$ are all typed attributed graph morphisms f that are almost injective, i.e., that are componentwise injective except perhaps for the mapping f_D of the data nodes as possible attribute values. In the context of graph transformations in Sec. 2.2.4, rules should be applied along $\mathcal{O}$-match morphisms that do not identify structures of graphs, but which may identify attribute expressions to identical values. The same situation arises for matches and the satisfaction of graph conditions and constraints in Sec. 2.2.3 and Def. 2.12. Therefore, $\mathcal{O}$ is a distinguished class of match morphisms. According to [EEGH15], the underlying categories $(\mathbf{Graphs},\mathcal{M})$ and $(\mathbf{Graphs}_{TG},\mathcal{M})$ of plain and typed graphs over type graph TG with(out) node type inheritance with $\mathcal{M}$ being the class of all (type strict) monomorphisms (i.e., componentwise injective morphisms) are also $\mathcal{M}$-adhesive.* △

Remark 2.4 ((Strict) $\mathcal{M}$-decomposition) *Note that $\mathcal{M}$-adhesive category $(\mathbf{AGraphs}_{ATGI},\mathcal{M})$ has $\mathcal{M}$-decompositions by definition Def. 2.6. However, $(\mathbf{AGraphs}_{ATGI},\mathcal{M})$ does not have strict $\mathcal{M}$-decompositions ($g \circ f \in \mathcal{M}$ implies $f \in \mathcal{M}$), since, f may not be an isomorphism on the data part f_D (cf. Rem. 2.3). Categories $(\mathbf{Graphs},\mathcal{M})$ and $(\mathbf{Graphs}_{TG},\mathcal{M})$ have strict $\mathcal{M}$-decompositions.* △

Usually, formal results are applied in the context of finitary $\mathcal{M}$-adhesive categories $(\mathbf{C}_{fin},\mathcal{M}_{fin})$ where the objects $Ob_{\mathbf{C}}$ and morphisms $Mor_{\mathbf{C}}$ of an $\mathcal{M}$-adhesive category $(\mathbf{C},\mathcal{M})$ are restricted to finite objects $Ob_{\mathbf{C}_{fin}} \subseteq Ob_{\mathbf{C}}$ with finitely many $\mathcal{M}$-subobjects and morphisms $Mor_{\mathbf{C}_{fin}} \subseteq Mor_{\mathbf{C}}$ between them. For example, the finitary $\mathcal{M}$-adhesive category $(\mathbf{AGraphs}_{ATGI,fin},\mathcal{M}_{fin})$ contains all typed attributed graphs G over type graph $ATGI$ that are finite in the sense that the graph part of G is finite (i.e., the sets of graph nodes, edges and attributes are finite) while type graph $ATGI$ and the data part of G may be infinite (i.e., the algebra of G and the sets of data nodes may be infinite) (cf. Thm. 4.47 in [EEGH15]). Moreover, class $\mathcal{M}_{fin}$ is the finitary restriction of class $\mathcal{M}$ according to the finitary restriction of morphisms from $\mathbf{AGraphs}_{ATGI}$ to $\mathbf{AGraphs}_{ATGI,fin}$. Finitary $\mathcal{M}$-adhesive categories have the additional HLR property of unique (extremal) $\mathcal{E}$-$\mathcal{M}$ factorisations for all morphisms (cf Prop. 3 in [BEGG10] for uniqueness & Prop. 4 in [BEGG10] or Thm. 4.42 in [EEGH15] for the existence of $\mathscr{E}$-$\mathscr{M}$ factorisations). $\mathscr{E}$-$\mathscr{M}$ factorisations are used for the definition of AC-schemata of graph conditions in Sec. 2.2.3 and Def. 2.18. For a class $\mathscr{E}$ of morphisms, an $\mathscr{E}$-$\mathscr{M}$ factorisation $m \circ e$ of a morphism f is a decomposition of f into morphisms $e \in \mathscr{E}, m \in \mathscr{M}$ such that $m \circ e = f$. Note that category $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ does not have $\mathscr{E}$-$\mathscr{M}$-factorisations in general for class $\mathscr{E}$ of all epimorphisms, since, for factorisations $m \circ e$, $\mathscr{M}$-morphisms in $\mathbf{AGraphs}_{ATGI}$ are isomorphisms on the data part m_D and therefore, e_D is not necessarily an epimorphism on the data part implying further that e is not necessarily in $\mathscr{E}$. Therefore, we review $\mathscr{E}$-$\mathscr{M}$ factorisations based on class $\mathscr{E}$ of all extremal morphisms w.r.t. $\mathscr{M}$. In $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, extremal $\mathscr{E}$-morphisms e w.r.t $\mathscr{M}$ are epimorphisms on the graph part e_S but not necessarily epimorphisms on the data part e_D. If $m \circ e$ is the $\mathscr{E}$-$\mathscr{M}$ factorisation of a morphism f in $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ which refines types along the type inheritance relation of type graph $ATGI$, then all refinements are shifted to morphism e, since, $\mathscr{M}$-morphism m is type strict according to Rem. 2.3. In the underlying $\mathscr{M}$-adhesive categories $(\mathbf{Graphs}, \mathscr{M})$ and $(\mathbf{Graphs}_{TG}, \mathscr{M})$ of plain and typed graphs over type graph TG, class $\mathscr{E}$ contains all epimorphisms and class $\mathscr{M}$ all monomorphisms and therefore, the (extremal) $\mathscr{E}$-$\mathscr{M}$ factorisation corresponds to the well-known epi-mono factorisation of morphisms. In finitary $\mathscr{M}$-adhesive categories, the extremal $\mathscr{E}$-$\mathscr{M}$ factorisation of a morphism $f\colon A \to B$ can be performed by constructing decompositions $m \circ e, e \in \mathscr{E}, m \in \mathscr{M}, m \circ e = f$ for all $\mathscr{M}$-subobjects $[m]$ of B and stepwise pullbacks of them as shown by Prop. 4 in [BEGG10].

Definition 2.8 (Finitary $\mathscr{M}$-adhesive Category & $\mathscr{M}$-subobject (Defs. 4.29 & 4.30 in [EEGH15])) *An $\mathscr{M}$-subobject of an object G* in an $\mathscr{M}$-adhesive category $(\mathbf{C}, \mathscr{M})$ is an equivalence class $[a\colon A \to G \in \mathscr{M}]$ of $\mathscr{M}$-morphisms with codomain G over equivalence relation $\sim := \{(a_1\colon A_1 \to G, a_2\colon A_2 \to G) \mid a_1, a_2 \in Mor_{\mathbf{C}}, \exists$ isomorphism $i\colon A_1 \to A_2 \in Mor_{\mathbf{C}}$ such that $a_2 \circ i = a_1\}$. Object G is finite if it has finitely many $\mathscr{M}$-subobjects. *An $\mathscr{M}$-adhesive category $(\mathbf{C}, \mathscr{M})$ is called finitary* if each object $G \in Ob_{\mathbf{C}}$ is finite. △

Definition 2.9 ((Extremal) $\mathscr{E}$-$\mathscr{M}$ Factorisation (Def. 4.34 in [EEGH15])) Given an $\mathscr{M}$-adhesive category $(\mathbf{C}, \mathscr{M})$, the class $\mathscr{E}$ of all *extremal morphisms w.r.t. $\mathscr{M}$* is defined by $\mathscr{E} := \{e \in Mor_{\mathbf{C}} \mid \forall m, g \in Mor_{\mathbf{C}}, m \circ g = e. m \in \mathscr{M} \Rightarrow m$ is an isomorphism$\}$. For a morphism $f \in Mor_{\mathbf{C}}$, an *(extremal) $\mathscr{E}$-$\mathscr{M}$ factorisation of f* is given by morphisms $e \in \mathscr{E}$ and $m \in \mathscr{M}$ such that $m \circ e = f$. △

Remark 2.5 (Uniqueness of Extremal $\mathscr{E}$-$\mathscr{M}$ Factorisation) *According to Fact 4.38 in [EEGH15], in $\mathscr{M}$-adhesive categories $(\mathbf{C}, \mathscr{M})$, extremal $\mathscr{E}$-$\mathscr{M}$ factorisations are unique up to isomorphism.* △

Remark 2.6 (Finitary $\mathscr{M}$-adhesive Categories, Existence of $\mathscr{E}$-$\mathscr{M}$ Factorisations & Initial Pushouts) *According to Rem. 2.3 and Thms. 4.42 & 4.47 in [EEGH15], category $(\mathbf{AGraphs}_{ATGI,fin}, \mathscr{M}_{fin})$ is finitary $\mathscr{M}$-adhesive and has a unique extremal $\mathscr{E}$-$\mathscr{M}$ factorisation with $\mathscr{E}$ being the class of all extremal morphisms w.r.t. $\mathscr{M}$ and furthermore, the category has initial pushouts. Consequently, the underlying categories $(\mathbf{Graphs}, \mathscr{M})$ and $(\mathbf{Graphs}_{TG}, \mathscr{M})$ (their finitary restrictions $(\mathbf{Graphs}_{fin}, \mathscr{M}_{fin})$ and $(\mathbf{Graphs}_{TG,fin}, \mathscr{M}_{fin})$) as well as $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ have a unique extremal $\mathscr{E}$-$\mathscr{M}$ factorisation (and are finitary $\mathscr{M}$-adhesive).* △

The definition of the satisfaction of graph constraints in Sec. 2.2.3 and Def. 2.15 relies on the notions of initial objects and initial morphisms.

Definition 2.10 (($\mathscr{M}$)-Initial Object (Defs. A.28 & 4.25 in [EEGH15])) In a category $\mathbf{C}$, an object I is called *initial* if for each object $G \in Ob_{\mathbf{C}}$ there exists a unique initial morphism $i_G \colon I \to G$. An initial object I in $\mathscr{M}$-adhesive category $(\mathbf{C},\mathscr{M})$ is called *$\mathscr{M}$-initial* if for each object $G \in Ob_{\mathbf{C}}$ the unique initial morphism $i_G \colon I \to G$ is in $\mathscr{M}$. △

Remark 2.7 (($\mathscr{M}$)-Initial Objects in Categories of Graphs) *In* $(\mathbf{Graphs},\mathscr{M})$ *and* $(\mathbf{Graphs}_{TG},\mathscr{M})$*, the initial and $\mathscr{M}$-initial object is the empty graph $\varnothing$ with the empty morphism $i_G \colon \varnothing \to G \in \mathscr{M}$ as unique initial morphism for each graph G. In category* $(\mathbf{AGraphs}_{ATGI},\mathscr{M})$*, object $((\varnothing,T_{DSIG}),type)$ is initial with $\varnothing$ being the empty graph, $type$ being the empty type morphism and T_{DSIG} being the DSIG-term algebra. In the following, we simply write $\varnothing$ for initial object $((\varnothing,T_{DSIG}),type)$. For a graph G, the unique initial morphism $i_G \colon ((\varnothing,T_{DSIG}),type) \to G$ is given by the empty morphism $i_{G,S}$ on the graph part and the unique* eval *morphism $i_{G,D}$ on the data part that evaluates the terms of T_{DSIG} to data values in DSIG-algebra D_G. However,* $(\mathbf{AGraphs}_{ATGI},\mathscr{M})$ *does not have an $\mathscr{M}$-initial object, since, according to Rem. 2.3 the initial $\mathscr{M}$-morphisms i_G are isomorphisms on the data part $i_{G,D}$ which does not hold for all graphs G in* $\mathbf{AGraphs}_{ATGI}$. △

For the results of Sects. 3.4 and 3.5 we assume $\mathscr{M}$-adhesive categories with effective pushouts.

Definition 2.11 (Effective Pushout (Def. 4.23 in [EEGH15])) Given a pullback (f,g) over (h_Y,k_Y) and a pushout (1) with all morphisms being $\mathscr{M}$-morphisms, then also the induced morphism $y \colon D \to Y$ is in $\mathscr{M}$. We say that (1) is the effective pushout over (h_Y,k_Y). △

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & B \arrow[d, "g'"] \arrow[ddr, "h_Y"] & \\
C \arrow[r, "f'"] \arrow[drr, "k_Y"'] & D \arrow[dr, "y"] & \\
 & & Y
\end{tikzcd}$$

Remark 2.8 (Effective Pushouts in Category $(\mathbf{AGraphs}_{ATGI},\mathscr{M})$) *According to Rem. 4.24 in [EEGH15], $\mathscr{M}$-adhesive categories may not have effective pushouts in general, but* $(\mathbf{AGraphs}_{ATGI},\mathscr{M})$ *has effective pushouts (cf. Rem 5.57 in [EEGH15]).* △

General Assumption In the following, we assume the context of $\mathscr{M}$-adhesive categories for definitions and results if not made explicit, especially when speaking of $\mathscr{M}$-morphisms.

2.2.3 Graph Conditions & Constraints

Formally, we define graph constraints via the notion of (nested) graph conditions according to [HP09]. Nested graph conditions provide the concepts for both, graph constraints and application conditions for graph transformation rules. Conditions are called constraints in the context of graphs where conditions may globally restrict the structure of graphs and are called application conditions in the context of rule definitions where conditions may restrict the application of rules.

Definition 2.12 ((Nested) Condition and Satisfaction (Def. 2.7 & 2.8 in [EEGH15])) *A (nested) condition ac_P* over a premise object P is inductively defined by:

- **true** is a condition over P.
- For every morphism $(a \colon P \to C)$ and condition ac_C over C, $\exists(a, ac_C)$ is a condition over P. Object C is called the conclusion w.r.t. premise object P.
- Boolean formulae over conditions, i.e., $\neg ac_P$, $\vee_{i\in I}(ac_{P,i})$, $\wedge_{i\in I}(ac_{P,i})$, are conditions over P for conditions ac_P and $ac_{P,i}, (i \in I)$ over P for some index set I.

A morphism $p\colon P \to G$ *satisfies a condition* ac_P *over* P (written $p \models ac_P$), if $p \in \mathcal{O}$ and:

- $ac_P = \mathbf{true}$, or
- $ac_P = \exists(a\colon P \to C, ac_C)$, $\exists\, q\colon C \to G \in \mathcal{M}$ with $q \circ a = p$ and $q \models ac_C$, or
- $ac_P = \neg ac'_P$ and $\neg(p \models ac'_P)$, or
- $ac_P = \wedge_{i\in I}(ac_{P,i})$ and for all $i \in I$ it holds that $p \models ac_{P,i}$, or
- $ac_P = \vee_{i\in I}(ac_{P,i})$ and there is an $i \in I$ with $p \models ac_{P,i}$.

Two conditions ac_P and ac'_P over P are equivalent, written $ac_P \equiv ac'_P$, if for all morphisms $p\colon P \to G$ it is true that $p \models ac_P$ if and only if $p \models ac'_P$. In contrast to the standard satisfiability of conditions via $\models$, $\models_{\mathcal{O}}$ defines the satisfiability of conditions with $q \in \mathcal{O}$ instead of $q \in \mathcal{M}$ ($\mathcal{O}$-satisfiability). For a nested condition ac_P, the number of nestings is given by the largest number of sequenced morphisms in ac_P. If the number of nestings of ac_P is zero or one, then ac_P is called a plain condition . According to Sec. 2.2.1 and Def. 2.2, for a given type graph *TG*, we say that a condition ac_P is typed over *TG*, if all objects (graphs) in ac_P are typed over *TG*. Consequently, a set of conditions C is typed over *TG*, if all $ac \in C$ are typed over *TG*. A condition ac_P is finite, if the index set I of every conjunction $\wedge_{i\in I}$ and disjunction $\vee_{i\in I}$ in ac_P is finite [HP09]. △

Remark 2.9 (Conditions – Abbreviations) *Although not being explicitly defined in Def. 2.12, we use the following abbreviations for conditions:* ***false*** $:= \neg\mathbf{true}$, $ac_P \Rightarrow ac'_P := ac'_P \vee \neg ac_P$, *and* $ac_P \Leftrightarrow ac'_P := (ac_P \Rightarrow ac'_P) \wedge (ac'_P \Rightarrow ac_P)$. △

Definition 2.13 (Positive Condition (Def 2.4 in [SEM+12])) *A condition* ac_P *over* P *is positive* if it does not contain negations $\neg$, i.e., ac_P is built up only by: 1. **true**, 2. $\exists(a, ac_C)$, and 3. $\vee_{i\in I}(\wedge_{i\in I})(ac_{P,i})$ with $I \neq \varnothing$. △

Definition 2.14 ($\mathcal{M}$-normal form (Def. 5 in [HP09])) A condition ac_P is in $\mathcal{M}$*-normal form*, if for all sub-conditions $\exists(a, ac)$ of ac_P, morphism a is in $\mathcal{M}$. △

Graph constraints are conditions that are extended to conditions over the initial object I when evaluating their satisfaction by graphs (cf. Def. 5.10 in [EEGH15]). In accordance with [SEM+12], we distinguish between initial and general satisfaction of graph constraints. While initial satisfaction refers to the existential satisfaction, general satisfaction refers to the universal satisfaction of constraints. Thus, a graph G initially satisfies a constraint ac_P if premise P occurs in G such that ac_P holds. In contrast, a graph G generally satisfies a constraint ac_P if for all occurrences of premise P in G condition ac_P holds.

Definition 2.15 (Constraint and Satisfaction) Let ac_P be a condition over P. *An object* G *initially satisfies a constraint* ac_P, written $G \overset{I}{\models} ac_P$, if the initial morphism $i_G\colon I \to G$ satisfies condition $ac_I = \exists(i_P\colon I \to P, ac_P)$ over initial object I and initial morphism i_P. *An object* G *generally satisfies a constraint* ac_P, written $G \models ac_P$, if the initial morphism $i_G\colon I \to G$ satisfies condition $ac_I = \neg\exists(i_P\colon I \to P, \neg ac_P)$ over initial object I and initial morphism i_P. *An object* G *initially (generally) satisfies a set of constraints* C, written $G \overset{I}{\models} C$ ($G \models C$) if $G \overset{I}{\models} ac$ ($G \models ac$) for all $ac \in C$. △

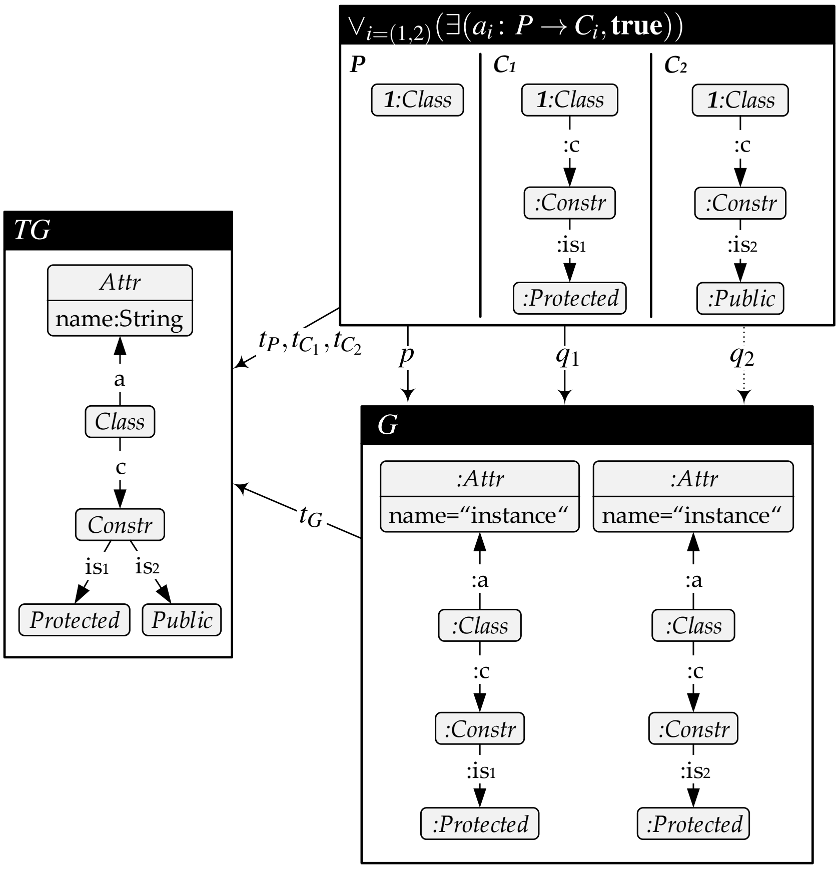


Figure 2.5: Graph Constraints and their Satisfaction by Graphs

Remark 2.10 (Initial & General Satisfaction of Constraints) *Note that for general satisfaction, condition $\forall(i_P\colon I \to P, ac_P)$ is equivalently expressed by $\neg\exists(i_P\colon I \to P, \neg ac_P)$. This allows us to speak of positive conditions ac_P in the sense of Def. 2.13 in view of their general satisfaction by graphs in Sec. 3.5 while the conditions that are actually used for evaluating their satisfaction are not positive. Note that by the uniqueness of initial morphisms, for conditions ac_P over P we have that:*

1. *$G \overset{I}{\models} ac_P \Leftrightarrow \exists p\colon P \to G \in \mathcal{M}$ such that $p \models ac_P$, and*
2. *$G \models ac_P \Leftrightarrow \forall p\colon P \to G \in \mathcal{M}$ it holds that $p \models ac_P$.* △

Example 2.2 (Graph Constraint and Satisfaction) *In reference to the type graph TG_{CD} of UML class diagrams in Sec. 2.2.1 and Fig. 2.4, Fig. 2.5 depicts a modified type graph TG, condition $ac_P = \bigvee_{i=(1,2)}(\exists(a\colon P \to C_i, \mathbf{true}))$ over P and graph G both typed over TG via morphisms $(t_x\colon x \to TG)_{x\in\{P,C_1,C_2,G\}}$. According to type graph TG, class diagrams may contain several* Classes, *each class may have several* Attr*ibutes each with a specific* name *of type* String *and each class has a* Constr*uctor with visibility* Protected *or* Public. *Constraint ac_P claims that each class has a constructor with visibility protected or public. Graph G both initially and generally satisfies constraint ac_P, since, morphism $p\colon P \to G \in \mathcal{M}$ can be mapped to the left or right class in G such that there exists $q_1\colon C_1 \to G \in \mathcal{M}$ with $q_1 \circ a = p$ and $q_1 \models$ **true**. However, there does not exist $q_2\colon C_2 \to G \in \mathcal{M}$ such that $q_2 \circ a = p$. Thus, for constraint $ac'_P = \bigwedge_{i=(1,2)}(\exists(a\colon P \to C_i, \mathbf{true}))$,*

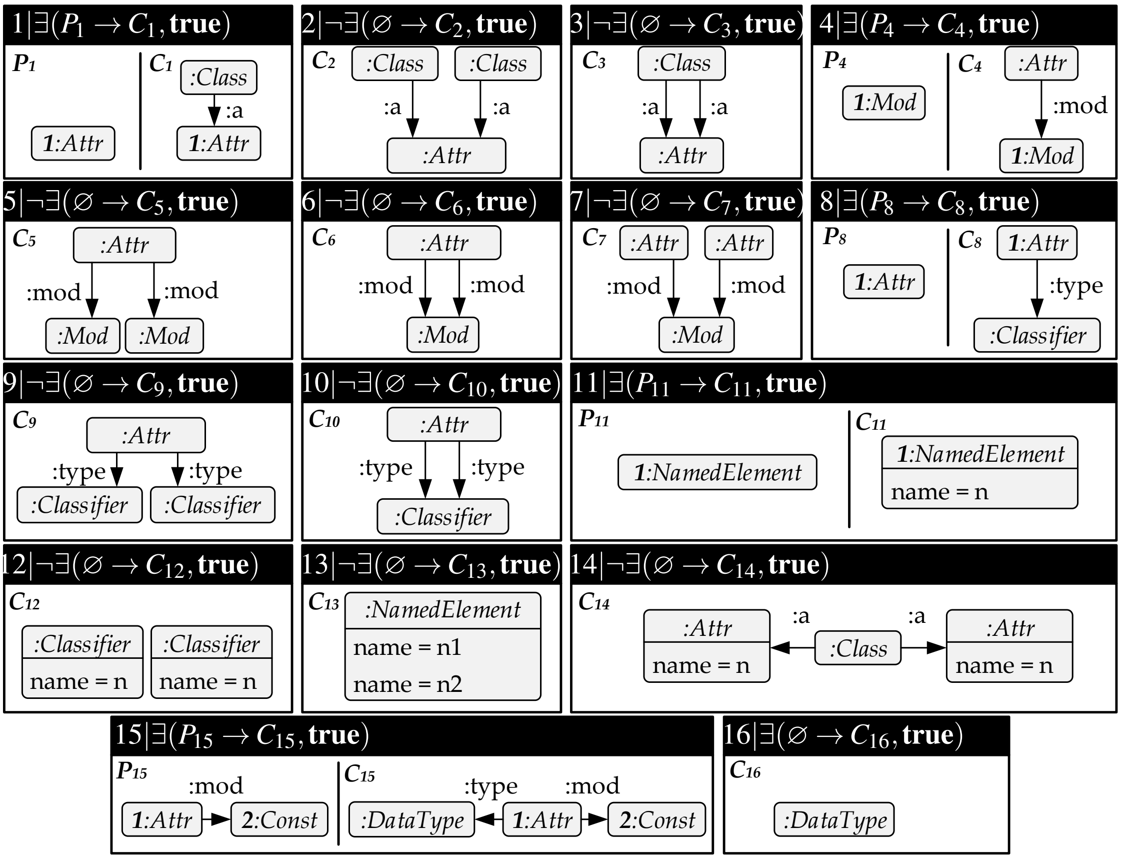


Figure 2.6: Graph Constraints for UML Class Diagrams

G satisfies ac'_P neither initially nor generally. △

Visual Notation According to Sec. 2.2.1, the mapping of nodes, edges and attributes along morphisms correspond to their naming in visual notation. For example in Fig. 2.5, node : Class in P is mapped to node : Class in C_1 (C_2) along a_1 (a_2) as indicated by its name 1.

Example 2.3 (Graph Constraints for UML Class Diagrams) *Fig. 2.6 depicts the graph constraints for UML class diagrams over type graph TG_{CD} from Sec. 2.2.1 and Fig. 2.4 which are used for verifying domain completeness in Sec. 3.2. All constraints in Fig. 2.6 are designated for general satisfaction. According to Sec. 2.2.1 and Ex. 2.1, this includes the multiplicity constraints in TG_{CD} which complete the meta-model of UML class diagrams: 1. Constraint* 1 *claims that each* Attribute *is assigned to a* Class *- in more detail - each attribute is assigned to exactly one class as refined by constraint* 2*, 2. Constraint* 4 *claims that each* Modifier *is the modifier of some attribute - in more detail - each modifier is the modifier of exactly one attribute as refined by constraint* 7*, 3. Analogously, constraint* 8 *claims that each attribute has some* type *- in more detail - each attribute has exactly one type as refined by constraint* 9*, and 4. Constraint* 5 *claims that each attribute has zero or one modifier. Beside the multiplicity constraints, the structure of class diagrams is additionally restricted by the following constraints: 1. Constraints* 3*,* 6 *and* 10 *forbid duplicate edges between two nodes, 2. Constraints* 11 *and* 13 *claim that each* NamedElement *has exactly one* name*, 3. Constraint* 12 *claims that different* Classifiers *must have different names, 4. Constraint* 14 *claims that different attributes of the same class must have dif-*

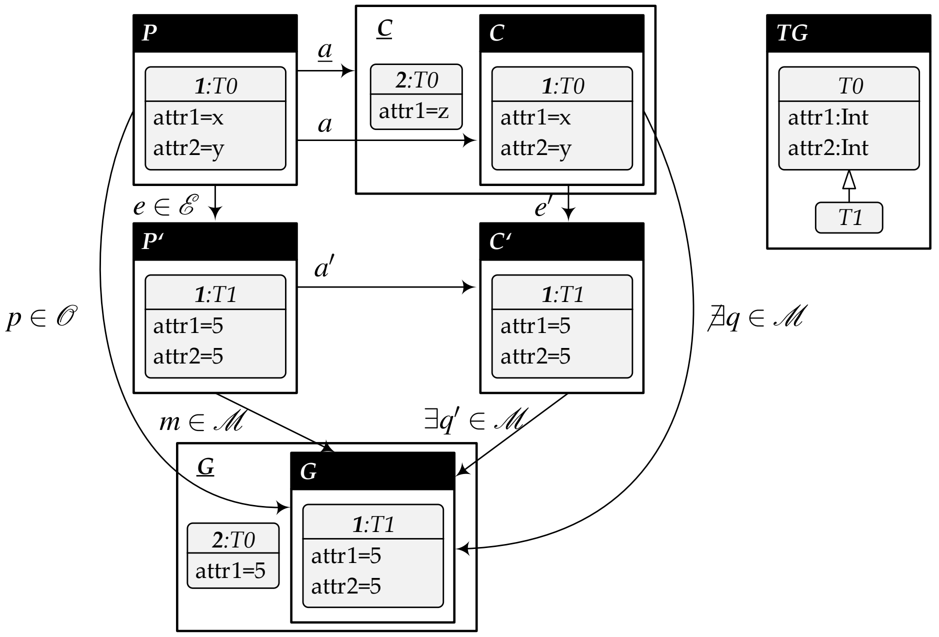


Figure 2.7: Non-Satisfiability and Instance of Condition $ac_P = \exists(a\colon P \to C \in \mathcal{M}, \textbf{true})$ along $\mathcal{O}$-matches (left) & Type Graph TG (right)

ferent names, 5. Constraint 15 *claims that each* Constant *attribute is of type* DataType *and not of type* Class*, and 6. Constraint* 16 *claims that there exists at least one* DataType. *Note that according to Sec. 2.2.1 and Rem. 2.1, abstract node types* Mod, Classifier *and* NamedElement *are forbidden to be directly used in graphs like P_4. However, instead of covering abstract types via the formal definition of graphs with node type inheritance, we assume dedicated constraints of the form* $\vee_{s \in S}\exists(1:\mathsf{t} \to 1:\mathsf{s}, \textit{\textbf{true}})$ *for each abstract type* $\mathsf{t} \in \{\mathsf{Mod}, \mathsf{Classifier}, \mathsf{NamedElement}\}$ *and all non-abstract sub-types S of* t *but that are not explicitly depicted in Fig. 2.6.* △

According to Def. 2.12, match morphisms p are in $\mathcal{O}$ but morphisms q are in $\mathcal{M}$ for standard satisfiability of conditions. Therefore according to Sec. 2.2.2 and Rem. 2.3 in $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$, match morphisms p may be non-injective on the data part p_D and may refine node types along a node type inheritance relation whereas morphisms q are isomorphisms on the data part q_D and are type strict. More specifically, in category $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$, conditions ac_P are often attributed via the $DSIG$-term algebra $T_{DSIG}(X)$ over variables X whereas graphs G are attributed via a concrete $DSIG$-algebra D_G. In most cases $T_{DSIG}(X)$ is not isomorphic to D_G, since, different terms in $T_{DSIG}(X)$ may be evaluated to the same concrete value in D_G, and therefore, $q \in \mathcal{M}$ does not exist. Thus, a condition ac_P in $\mathcal{M}$-normal form may be non-satisfiable by p although the condition seems to be a tautology as shown in Fig. 2.7 where p identifies variables x and y with value 5 as well as refines type T_0 to T_1 along the inheritance relation in type graph TG and there does not exist $q \in \mathcal{M}$ with $q \circ a = p$. Therefore, according to Def. 2.18, we use the concept of an AC-schema, which interprets a constraint ac_P as the disjunction of its possible instances $\mathrm{Inst}(ac_P)$ which may occur in a graph. Based on the merge construction over extremal $\mathcal{E}$-morphisms w.r.t. $\mathcal{M}$ in Def. 2.16, the instances $\mathrm{Inst}(ac_P)$ subsume all possible type refinements and identifications of data values along match morphisms (cf. Def. 2.17). Since, according to Rem. 2.13, the satisfaction of an AC-schema by match $p \in \mathcal{O}$ coincides with the satisfaction of the corresponding instance in the schema by $m \in \mathcal{M}$ that is derived by the $\mathcal{E}$-$\mathcal{M}$ factorisation of p, this allows us to use AC-schemata as a compact notation and to focus on $\mathcal{M}$-matches while leaving the possible type refinements and identifications of data values implicit. For example, $(e, \exists(a' \colon P' \to C', \mathbf{true}))$ is an instance of ac_P in Fig. 2.7 (the identification of variables x and y as well as the type refinement from T_0 to T_1 is transferred to the instance), $m \circ e = p$ is an $\mathcal{E}$-$\mathcal{M}$ factorisation of p and furthermore, there exists $q' \in \mathcal{M}$ with $q' \circ a' = m$ (i.e., m satisfies the instance) and therefore, p satisfies the AC-schema of ac_P. Furthermore, in Prop. 2.2 we show that the standard satisfiability of an AC-schema coincides with the $\mathcal{O}$-satisfiability of the underlying condition. This allows an interpretation of conditions via $\mathcal{O}$-matches and $\mathcal{O}$-satisfiability from a user point of view while an interpretation via the standard satisfiability of AC-schemata with $\mathcal{M}$-matches is used to prove technical results.

Intuitively, the merge construction transfers the identifications of data values and type refinements along the given morphism $b \colon P \to P'$ to $b' \colon C \to C'$ by respecting the identifications and type refinements along $a \colon P \to C$. Additionally, it allows for type refinements of elements and identifications of data elements that are in C but not in P.

Definition 2.16 (Merge over Morphism (Def. 5.5 in [EEGH15])) Given a condition ac over P and a morphism $b : P \to P'$, then $\mathrm{Merge}(b, ac)$ *is a condition over* P' defined by

$$\begin{tikzcd}
ac \arrow[r, phantom, "\triangleright"] & P \arrow[r, "a"] \arrow[d, "b"'] \arrow[dr, phantom, "(1)"] & C \arrow[d, "b' \in \mathcal{O}"] \arrow[r, phantom, "\triangleleft"] & ac' \\
\mathrm{Merge}(b,ac) \arrow[r, phantom, "\triangleright"] & P' \arrow[r, hook, "a' \in \mathcal{M}"] & C' \arrow[r, phantom, "\triangleleft"] & \mathrm{Merge}(b',ac')
\end{tikzcd}$$

- $\mathrm{Merge}(b, ac) = \mathbf{true}$ if $ac = \mathbf{true}$,
- $\mathrm{Merge}(b, ac) = \vee_{(a',b') \in \mathcal{F}} \exists(a', \mathrm{Merge}(b', ac'))$ if $ac = \exists(a, ac')$ and $\mathcal{F} = \{(a', b') \mid a' \in \mathcal{M} \wedge b' \in \mathcal{O} \wedge (1) \text{ commutes } \wedge (a', b') \text{ are jointly epimorphic}\}$,
- $\mathrm{Merge}(b, ac) = \neg \mathrm{Merge}(b, ac')$ if $ac = \neg ac'$,
- $\mathrm{Merge}(b, ac) = \wedge_{i \in I} \mathrm{Merge}(b, ac_i)$ if $ac = \wedge_{i \in I} ac_i$, or
- $\mathrm{Merge}(b, ac) = \vee_{i \in I} \mathrm{Merge}(b, ac_i)$ if $ac = \vee_{i \in I} ac_i$. △

Remark 2.11 *In* $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$*, note that if morphism a identifies elements of P or refines types in P that are not identified or not refined to equal or finer types by morphism b, respectively, then* (1) *cannot be constructed, since, it is required that* $a' \in \mathcal{M}$ *while* (1) *commutes, and the merge construction returns* ***false*** *(an empty disjunction over* $(a', b') \in \mathcal{F}$*). Analogously, if morphism b identifies graph elements of P that are not identified by morphism a, then* (1) *cannot be constructed and* ***false*** *is returned, since, it is required that* $b' \in \mathcal{O}$ *while* (1) *commutes. For a characterisation of* $\mathcal{M}$*- and* $\mathcal{O}$*-morphisms in* $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ *we refer to Sec. 2.2.2 and Rem. 2.3.* △

Definition 2.17 (Instances of Conditions) Given a condition ac_P over P with $\mathcal{E}_P = \{e \in \mathcal{E} \mid \mathrm{dom}(e) = P\}$ being the set of all extremal $\mathcal{E}$-morphisms w.r.t. $\mathcal{M}$ with domain P. The *instances of* ac_P are given by $\mathrm{Inst}(ac_P) = \bigcup_{f \in \mathcal{E}_P} \{(f, \mathrm{Merge}(f, ac_P))\}$. Given a set of conditions C, then the instances of C are given by $\mathrm{Inst}(C) = \bigcup_{ac \in C} (\mathrm{Inst}(ac))$. △

Remark 2.12 (Instances in $\mathcal{M}$-normal Form) *Note that the conditions in instances are in* $\mathcal{M}$*-normal form by merge construction.* △

Definition 2.18 (AC-schema (Def. 5.6 in [EEGH15])) Given a condition ac_P over P, then *the AC-schema* $\overline{ac}_P$ *of* ac_P is a condition over P given by $\overline{ac}_P = \bigvee_{(f,ac) \in \mathrm{Inst}(ac_P)} \exists(f, ac)$. For $ac_P = \mathbf{true}$, $\overline{ac}_P = \mathbf{true}$. △

$$\begin{tikzcd}
\overline{ac}_P \triangleright P \arrow[rr, "p"] \arrow[dr, "e"'] & & G \\
& \mathrm{Merge}(e, ac_P) \triangleright P' \arrow[ur, hook, "m"'] &
\end{tikzcd}$$

Remark 2.13 (AC-schema satisfaction (Fact 5.8 in [EEGH15])) *Given an AC-schema $\overline{ac}_P$ of condition ac_P over P and a morphism $p\colon P \to G \in \mathcal{O}$ with an extremal $\mathcal{E}$-$\mathcal{M}$-factorisation $m \circ e = p$, then $p \models \overline{ac}_P$ if and only if $m \models \mathrm{Merge}(e, ac_P)$ with $(e, \mathrm{Merge}(e, ac_P)) \in \mathrm{Inst}(ac_P)$. Note that the satisfaction of conditions by morphisms in Sec. 2.2.3 and Def. 2.12 is defined based on $\mathcal{O}$-morphisms. However, $m \in \mathcal{O}$ by $m \in \mathcal{M}$ in $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ (cf. Sec. 2.2.2 and Rem. 2.3).* △

Note that by Def. 2.18, the AC-schema $\overline{ac}_P$ of a constraint ac_P over P is again a constraint over P. However, the satisfaction of AC-schemata by objects cannot be directly defined by Rem. 2.10. For example, constraint $\neg ac_P$ in Fig. 2.7 seems to be a contradiction with $G \not\models \neg ac_P$ but $G \models \neg ac_P$ if both algebras of G and $\neg ac_P$ are not isomorphic, i.e., $p\colon P \to G \in \mathcal{M}$ does not exist (cf. Sec. 2.2.2 and Rem. 2.3). Therefore, based on Rem. 2.10 the satisfaction of AC-schemata is defined as follows.

Definition 2.19 (Initial & General Satisfaction of AC-schemata) *An object G initially satisfies AC-schema* $\overline{ac}_P$ of constraint ac_P over P, if $\exists p\colon P \to G \in \mathcal{O}$ such that $p \models \overline{ac}_P$. *An object G generally satisfies* $\overline{ac}_P$, if $\forall p\colon P \to G \in \mathcal{O}$ it holds that $p \models \overline{ac}_P$. By Rem. 2.13, in $\mathcal{M}$-adhesive categories with extremal $\mathcal{E}$-$\mathcal{M}$ factorisations, G initially satisfies $\overline{ac}_P$ if and only if $\exists$ instance $(e\colon P \to P', \mathrm{Merge}(e, ac_P)) \in \mathrm{Inst}(ac_P)$ of ac_P such that $G \overset{I}{\models} \mathrm{Merge}(e, ac_P)$. Analogously, G generally satisfies $\overline{ac}_P$ if and only if $\forall$ instances $(e\colon P \to P', \mathrm{Merge}(e, ac_P)) \in \mathrm{Inst}(ac_P)$ it holds that $G \models \mathrm{Merge}(e, ac_P)$. For initial and general satisfaction of constraints $\mathrm{Merge}(e, ac_P)$, we refer to Def. 2.15 and Rem. 2.10. △

Remark 2.14 (Data in Constraints and Application Conditions) *Consider condition $ac'_P = \exists(\underline{a}\colon P \to \underline{C}, \textbf{true})$ in Fig. 2.7 with $P, \underline{C}$ being attributed graphs and which claims that there is an additional* : $\mathsf{T0}$ *node with attribute* $\mathsf{attr1}$ *of value* z. *Assume that P contains a variable z' in its algebra that is mapped to variable z in $\underline{C}$ along morphism $\underline{a}$, then the interpretation of ac'_P according to Def. 2.19 may be misleading, e.g., in contrast to the intended result, $\underline{G}$ does not generally satisfy the AC-schema $\overline{ac}'_P$ of constraint ac'_P. This is due to the fact that the mapping of z along $\underline{q}\colon \underline{C} \to \underline{G} \in \mathcal{O}$ is prescribed by the mapping of z' along match $\underline{p}\colon P \to \underline{G} \in \mathcal{O}$ in order to obtain $\underline{q} \circ \underline{a} = \underline{p}$ (cf. Prop. 2.2). By Def. 2.19, for general satisfaction all matches $\underline{p}\colon P \to \underline{G} \in \mathcal{O}$ are considered, i.e., z' and therefore also z may be mapped to the same value than x or y (i.e., value* 5 *for the example) but also to any other value along $\underline{p}$ and $\underline{q}$, respectively. Thus, more precisely, for general satisfaction and graphs $\underline{G}$ that allow different values for variables x, y and z, ac'_P claims that there are additional* : $\mathsf{T0}$ *nodes, one with attribute* $\mathsf{attr1}$ *of value* z *that equals to* x, *one with attribute* $\mathsf{attr1}$ *of value* z *that equals to* y *and one with attribute* $\mathsf{attr1}$ *of value* z *that differs from* x *and* y. *The same situation arises for variables that are not used as attribute values in attributed premise and conclusion graphs of conditions but that are non-injectively mapped along the internal morphisms of conditions, since, for matches $\underline{p}$ that are injective on these variables, the corresponding $\underline{q}$-morphism with $\underline{q} \circ \underline{a} = \underline{p}$ does not exist. Therefore, for conditions ac over attributed graphs we generally assume that for all internal morphisms $a\colon P \to C$ in ac it is true that all variables that are used as attribute values in C but not in P are not in the image of a. Furthermore, all variables in P that are not used as attribute values are injectively mapped along a. Under this general assumption for constraint ac'_P, graph $\underline{G}$ both initially and generally satisfies $\overline{ac}'_P$.* △

According to Prop. 2.1, for $\mathcal{M}$-matches and the case of category $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ (and underlying categories $(\mathbf{Graphs}, \mathcal{M})$ and $(\mathbf{Graphs}_{TG}, \mathcal{M})$), the interpretation of a condition ac_P

that is type strict via the standard satisfiability of its AC-schema coincides with the standard satisfiability of ac_P itself. According to Def. 2.20, a condition ac_P over P is type strict, if it is in $\mathscr{M}$-normal form and furthermore, the types of all nodes in conclusions that are not in premise P cannot be refined along morphisms. Therefore, for $\mathscr{M}$-matches and type strict conditions, the direct interpretation of conditions is equivalent to their interpretation via AC-schemata. This may not hold for general $\mathscr{O}$-matches and for conditions that are not type strict. However, Prop. 2.2 shows for $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ and underlying categories that the standard satisfiability of an AC-schema coincides with the $\mathscr{O}$-satisfiability of the underlying condition.

Definition 2.20 (Type Strict Condition) In $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, *a condition ac_P over P is type strict*, if ac_P is in $\mathscr{M}$-normal form and for all morphisms $a\colon P \to C$ in ac_P with domain P it holds that for all nodes $n \in V_G^C \setminus a_{G,V_G}(V_G^P)$ that are in C but not in P, node type $type_C(n)$ does not have sub-types in the inheritance relation of type graph $ATGI$ and therefore, cannot be refined along morphisms. △

Proposition 2.1 (Relationship between Satisfiability of Conditions and AC-Schemata) *In $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, given a type strict condition ac_P over P, its AC-schema $\overline{ac}_P$ and a match $p : P \to G \in \mathscr{M}$. Then, $p \models ac_P$ if and only if $p \models \overline{ac}_P$.* △

Proof. The proof is presented in appendix A.1. □

Proposition 2.2 (Relationship between Standard- and $\mathscr{O}$-Satisfiability of Conditions) *In $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, given a condition ac_P over P, its AC-schema $\overline{ac}_P$ and a match $p\colon P \to G \in \mathscr{O}$. Then, $p \models_{\mathscr{O}} ac_P$ if and only if $p \models \overline{ac}_P$.* △

Proof. The proof is presented in appendix A.2. □

General Assumption Note that we interpret conditions by AC-schemata that are formed over (extremal) $\mathscr{E}$-morphisms and their satisfaction can be defined based on (extremal) $\mathscr{E}$-$\mathscr{M}$ factorisations by Rem. 2.13. Therefore, in addition to the general assumption from Sec. 2.2.2, we assume $\mathscr{M}$-adhesive categories with unique (extremal) $\mathscr{E}$-$\mathscr{M}$ factorisation. Furthermore, we assume that all application conditions and graph constraints are interpreted via their AC-schemata according to Def. 2.18, if not made explicit. Moreover, we generally assume the assumption from Rem. 2.14 for constraints.

2.2.4 Graph Grammars, Transformations & $\mathscr{M}$-adhesive Transformation Systems

As discussed in Sec. 1.3, models are represented by graphs and model transformations and synchronisations are performed based on graph transformations. Furthermore, graph transformations are performed by applying graph transformation rules to graphs. According to Def. 2.21, a rule p is a span $p = (L \xleftarrow{L} K \xrightarrow{r} R)$ of $\mathscr{M}$-morphisms l and r with left-hand side L, gluing K and right-hand side R. A rule is applied to a graph G via a match-morphism $m\colon L \to G$ as defined by direct transformations in Def. 2.22.

Definition 2.21 ((Transformation) Rule (Def. 5.12 in [EEGH15])) A *plain (transformation) rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of objects L, K and R, called left-hand side, gluing and right-hand side, respectively and two morphisms $l, r \in \mathscr{M}$. A *(transformation) rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac_L)$ consists of a plain rule and a condition ac_L over L, called application condition. △

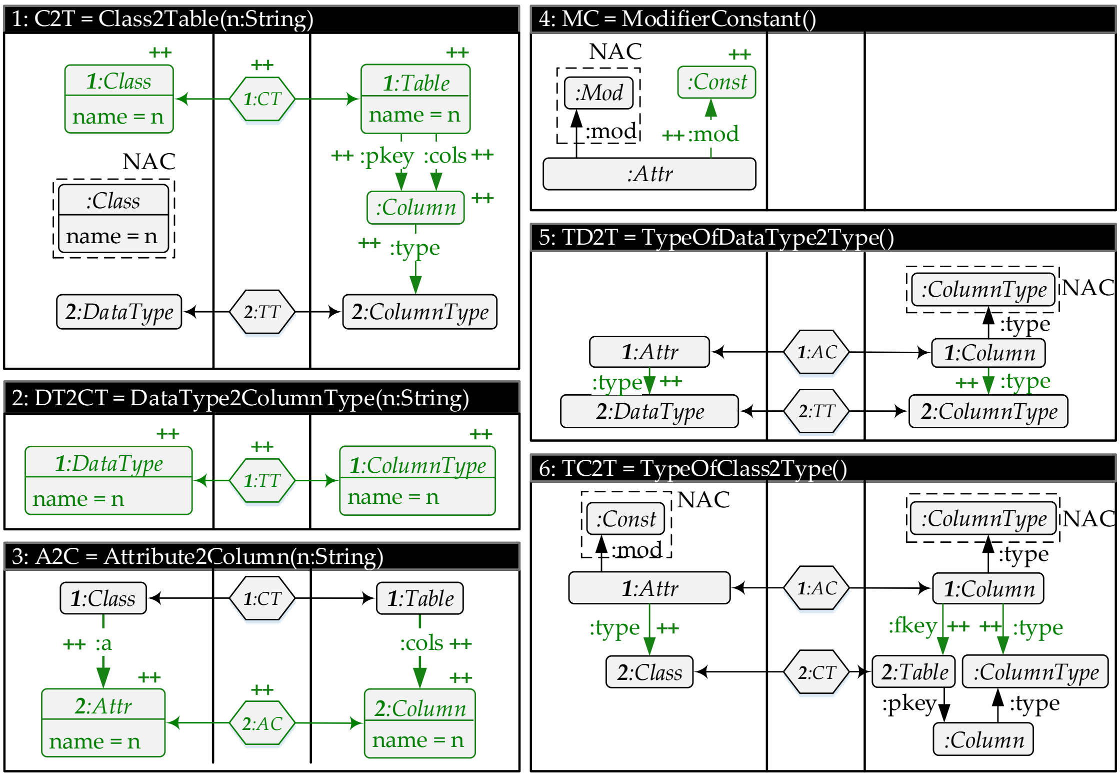


Figure 2.8: Triple Graph Grammar (CD2RDBM)

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

Definition 2.22 (Transformation (Def. 5.13 in [EEGH15])) Given a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac_L)$, an object G, and a morphism $m\colon L \to G$, called match, such that $m \models ac_L$, *a direct transformation (step)* $G \xRightarrow{(p,m)} H$ from G to an object H via p and match m is given by the pushouts (1) and (2) with comatch n. Given a set of rules P, a sequence of direct transformations from G to H via $p \in P$ is called *a transformation (sequence) via* P, written $t\colon G \overset{*}{\Rightarrow} H$. △

Intuitively, the application leads to a graph H where $m(L) \setminus m(l(K))$ is deleted from G and $R \setminus r(K)$ is added while $m(l(K))$ is preserved. Furthermore, a rule may be equipped with an application condition ac_L which may restrict the application of the rule to specific matches, i.e, for rule applications, match m must additionally satisfy application condition ac_L. According to Def. 2.13, we distinguish between positive application conditions (PACs) and negative application conditions (NACs). NACs declare forbidden patterns and PACs declare patterns that must exist for rule applications. We say that a *rule is non-deleting*, if $l = id_L$ and $K = L$. In case of non-deleting rules $p = (L \xleftarrow{id} L \overset{r}{\hookrightarrow} R, ac_L)$, we write $p = (L \overset{r}{\hookrightarrow} R, ac_L)$ and pushout (1) in Def. 2.22 is omitted with $D = G$ and $k = m$. According to Sec. 2.2.1 and Def. 2.2 and Sec. 2.2.3 and Def. 2.12, we say that a rule $p = (L \leftarrow K \rightarrow R, ac_L)$ is typed over type graph TG, if L, K, R and ac_L are typed over common TG. We take production as synonym for rule.

A transformation system is a set of rules. A transformation system together with a start object constitutes a grammar. Usually, we take graphs as objects and speak of graph grammars (cf. Sec. 2.2.2 and Rem. 2.2). The language over a grammar is given by all objects that are reachable from the start object by transformations via the rules of the grammar (cf. Sec. 3.1 and Def. 3.1).

We say that a grammar is non-deleting, if all its rules are non-deleting. We say that a grammar is typed over *TG*, if the start object and all its rules are typed over *TG*. We say that a grammar is without application conditions, if all its rules are plain rules. A grammar is finite, if the set of rules is finite.

Definition 2.23 ($\mathcal{M}$-adhesive Transformation System & Grammar) Given an $\mathcal{M}$-adhesive categoriy $(\mathbf{C},\mathcal{M})$, *an $\mathcal{M}$-adhesive transformation system* $AS = (P)$ is given by a set of rules P. *A grammar* $GG = (S,AS)$ is given by a transformation system AS and a start object S. △

Visual Notation In addition to the notational conventions from Sec. 2.2.1, transformation rules are visualised as depicted in Fig. 2.8. All graph elements that are marked with $++$ are added by the rule and therefore, are only contained in the right-hand side of the rule. All graph elements that are marked with $--$ are deleted by the rule and therefore, are only contained in the left-hand side of the rule. All graph elements that are unmarked are preserved by the rule and therefore, are contained in the left-hand side and the gluing of the rule. Additionally, a rule may be equipped with an application condition which consists of the left-hand side of the rule together with those graph elements that are enclosed by PAC or NAC boxes.

Example 2.4 (Rules & Transformations) *The rules in Fig. 2.8 simultaneously create class diagrams and relational database models. For each rule p, we focus on the projection* p^{S} *to the source domain of class diagrams (left parts). Rule* 1^{S} *creates a* Class *of* name n *in addition to an existing* DataType *but only of there does not already exist a class of the same name (cf. NAC). Rule* 2^{S} *creates a* DataType *of* name n. *Rule* 3^{S} *creates an* Attribute *of* name n *and assigns it to an existing* Class *via edge* : a. *Rule* 4^{S} *creates a* Constant Mod*ifier to an existing attribute but only of the attribute does not already have a modifier (cf. NAC). Rule* 5^{S} *assigns an existing data type to an existing attribute as* type. *Rule* 6^{S} *assigns an existing class to an existing attribute as* type *but only if the attribute is not* Constant *(cf. NAC). There is a transformation* $\varnothing \xRightarrow{(2^{\mathrm{S}},\text{-})} G_1 \xRightarrow{(1^{\mathrm{S}},\text{-})} G_2 \xRightarrow{(3^{\mathrm{S}},\text{-})} G_3 \xRightarrow{(4^{\mathrm{S}},\text{-})} G_4 \xRightarrow{(5^{\mathrm{S}},\text{-})} CD$ *from the empty graph* $\varnothing$ *to graph CD in Sec. 2.2.1 and Fig. 2.4. Analogously, we derive six rules by a projection to the right parts that create the elements of relational database models.* △

In view of Sec. 3.4, we recall the notion of the derived span of transformations.

Definition 2.24 (Derived Span [EEGH15]) Let $t\colon G_0 \overset{*}{\Rightarrow} G_n$ be a transformation, then the *derived span* $der(t)$ *of* t is inductively defined by

$$der(t) = \begin{cases} G \xleftarrow{id_G} G \xrightarrow{id_G} G & \text{, for identical (empty) transformation } t\colon G \overset{id}{\Longrightarrow} G \\ G_{i-1} \xleftarrow{f_i} D_i \xrightarrow{g_i} G_i & \text{, for } t\colon G_{i-1} \xRightarrow{(p_i,m_i)} G_i \text{ being a direct transformation} \\ & \text{with pushouts}(1) \text{ and } (2) \\ G_0 \xleftarrow{d'_0 \circ d} D \xrightarrow{g_n \circ d_n} G_n & \text{, for } t\colon G_0 \overset{*}{\Rightarrow} G_{n-1} \xRightarrow{(p_n,m_m)} G_n \text{ with} \\ & der(G_0 \overset{*}{\Rightarrow} G_{n-1}) = (G_0 \xleftarrow{d'_0} D' \xrightarrow{d'_{n-1}} G_{n-1}) \\ & \text{and pullback } (PB) \end{cases}$$

△

$$\begin{tikzcd}
p_i = (L_i \arrow[d, "m_i"'] \arrow[dr, phantom, "(1)"] & K_i \arrow[l, "l_i"'] \arrow[r, "r_i"] \arrow[d] \arrow[dr, phantom, "(2)"] & R_i) \arrow[d] \\
G_{i-1} & D_i \arrow[l, "f_i"] \arrow[r, "g_i"'] & G_i
\end{tikzcd}
\qquad
\begin{tikzcd}
G_0 & D' \arrow[l, "d'_0"'] \arrow[r, "d'_{n-1}"] & G_{n-1} \arrow[d, phantom, "(PB)"] & D_n \arrow[l, "f_n"'] \arrow[r, "g_n"] & G_n \\
& & D \arrow[ul, "d"] \arrow[ur, "d_n"'] & &
\end{tikzcd}$$

Remark 2.15 (Derived Span for Non-Deleting Rules) *For transformations $t\colon G_0 \overset{*}{\Rightarrow} G_n$ via non-deleting rules only and with direct transformations $(G_{i-1} \xRightarrow{(p_i,m_i)} G_i)_{i\in\{1,\dots,n\}}$, the derived span $der(t)\colon G_0 \to G_n$ of t is given by $der(t) := g_n \circ \ldots \circ g_1$. Note that in $\mathcal{M}$-adhesive categories, the derived span for transformations via non-deleting productions is in $\mathcal{M}$ by productions being spans of $\mathcal{M}$-morphisms ($r_i \in \mathcal{M}$), $\mathcal{M}$-morphisms are closed under pushouts, i.e., $g_i \in \mathcal{M}$, and $\mathcal{M}$-composition, i.e., $g_n \circ \ldots \circ g_1 \in \mathcal{M}$.* △

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & E \arrow[d] \\
C \arrow[r] & D \arrow[r] & F
\end{tikzcd}$$

We review the following general existing results, as, they are used in definitions or proofs: **Restriction Theorem** (Thm. 6.18 in [EEPT06]) : Direct transformations can be restricted along specific decompositions of match-morphisms, **PO-(De)-Composition** (Lem. A.21 in [EEPT06]) : For pushouts (1) and (2), also $(1)+(2)$ is a pushout, and if (1) and $(1)+(2)$ are pushouts, then also (2), and **PB-(De)-Composition** (Lem. A.25 in [EEPT06]) : For pullbacks (1) and (2), also $(1)+(2)$ is a pullback, and if (2) and $(1)+(2)$ are pullbacks, then also (1). **Critical Pair** : Given a transformation system P, a critical pair $(K_1 \xLeftarrow{(p_1,m_1)} O \xRightarrow{(p_2,m_2)} K_2)$ for rules P is a pair of direct transformations $t_1\colon O \xRightarrow{(p_1,m_1)} K_1$ and $t_2\colon O \xRightarrow{(p_2,m_2)} K_2$ with common object O and $p_1, p_2 \in P$ where both transformations are in conflict to each other, intuitively, in the sense that (1) transformation t_1 deletes elements from O to K_1 that are “used” by match m_2 for transformation t_2 or vice versa, or (2) K_1 does not satisfy the application condition of p_2 anymore or vice versa and furthermore, where object O is minimal. Object O is minimal means that matches m_1 and m_2 are jointly epimorphic (surjective) on O, i.e., O only contains elements that are covered by transformations t_1 and t_2. In the context of graphs, O is called the conflict graph of the critical pair. For technical details we refer to Def. 2.39 in [EEGH15]. Note that we concentrate on critical pairs for rules over graphs with translation (marking) attributes (cf. Sec. 2.3.2 and Rem. 2.17). This involves forward translation rules (cf. Sec. 2.3.2 and Def. 2.26), consistency creating rules (cf. Sec. 2.3.3 and Def. 2.29) and marking rules (cf. Sec. 3.2 and Def. 3.9) that only update translation (marking) attributes from **F** to **T** while preserving the remaining graph structure in terms of deletions. Note that technically, the update of an attribute is a deletion of the old attribute value followed by a creation of the new attribute value (cf. Sec. 2.2.1 and Def. 2.3). Fig. 2.9 depicts a critical pair $(K_1 \xLeftarrow{(m(4^S),m_1)} O \xRightarrow{(m(4^S),m_2)} K_2)$ for marking rule $m(4^S)$ of rule 4^S in Ex. 2.4. Both transformations simultaneously update the marking attribute of node 1 : Const from **F** to **T**. **Strict Confluence** : A critical pair $(K_1 \xLeftarrow{(p_1,m_1)} O \xRightarrow{(p_2,m_2)} K_2)$ is strictly confluent, if there are transformations $t_1\colon K_1 \overset{*}{\Rightarrow} O'$ and $t_2\colon K_2 \overset{*}{\Rightarrow} O'$ to common O' such that all elements that are preserved by both transformations of the critical pair are also preserved by transformations t_1 and t_2 such that they can be embedded into bigger contexts. For technical details we refer to Def. 2.42 in [EEGH15]. **Local Confluence Theorem** (Thm. 2.43 in [EEGH15]) : A transformation system P is locally confluent if, for all direct transformations $G \xRightarrow{(p_1,m_1)} H_1$ and $G \xRightarrow{(p_2,m_2)} H_2$ via $p_1, p_2 \in P$, there is an object X and transformations $H_1 \overset{*}{\Rightarrow} X$ and $H_{\cdot\cdot} \overset{*}{\Rightarrow} X$ via P. A transformation system is locally confluent if all its critical pairs are strictly confluent. **Confluence** (Lem. 3.32 in [EEPT06]) : A transformation system P is confluent if, for all transformations $G \overset{*}{\Rightarrow} H_1$

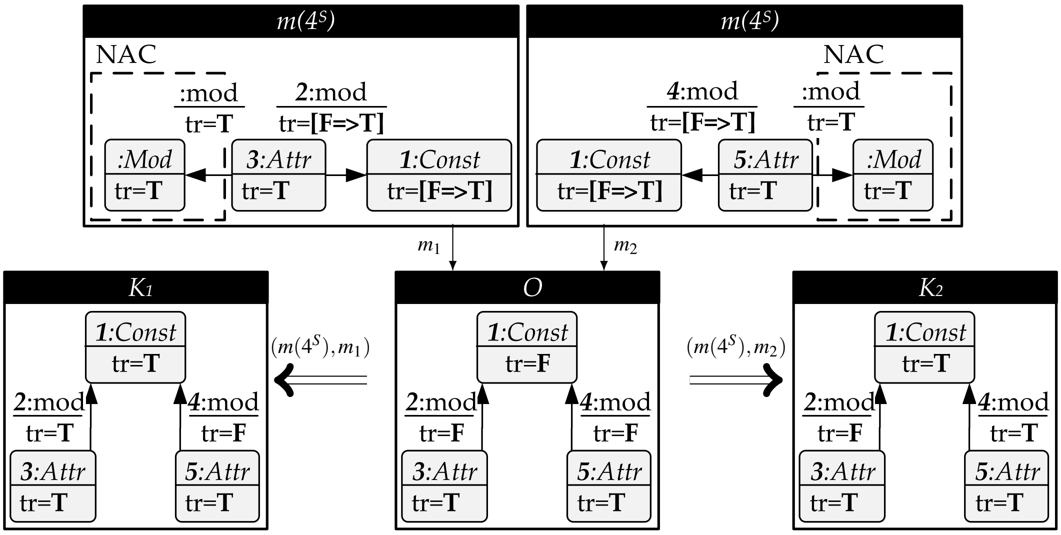


Figure 2.9: Critical Pair $(K_1 \xLeftarrow{(m(4^S),m_1)} O \xRightarrow{(m(4^S),m_2)} K_2)$

and $G \overset{*}{\Rightarrow} H_2$ via P, there is an object X and transformations $H_1 \overset{*}{\Rightarrow} X$ and $H_2 \overset{*}{\Rightarrow} X$ via P. Every terminating and locally confluent transformation system is also confluent. A transformation system P is terminating if there is no infinite transformation sequence via P.

General Assumption For $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ and underlying categories, we restrict the application of rules to almost injective matches $m \in \mathscr{O}$ (cf. Sec. 2.2.2 and Rem. 2.3). Furthermore, we assume that all application conditions are interpreted via their AC-schemata according to Sec. 2.2.3 and Def. 2.18. Moreover, we generally assume the assumption from Sec. 2.2.3 and Rem. 2.14 for application conditions.

2.3 Model Transformation & Synchronisation based on TGGs

This section reviews the concept of triple graph grammars (TGGs) as well as model transformations and synchronisations based on TGGs.

2.3.1 Triple Graphs & Triple Graph Grammars (TGGs)

According to Sec. 2.1.1, triple graph grammars (TGGs) allow the definition of visual, declarative transformation specifications for model transformations and synchronisations. Therefore, we review basic notions of TGGs from [EEGH15].

A triple graph $G = (G^{\mathrm{S}} \xleftarrow{s_G} G^{\mathrm{C}} \xrightarrow{t_G} G^{\mathrm{T}})$ is an integrated model consisting of a source graph G^{S}, a target graph G^{T} and explicit correspondences between them (cf. Def. 3.3 in [EEGH15]). The correspondences are given by correspondence graph G^{C} together with morphisms $s_G\colon G^{\mathrm{C}} \to G^{\mathrm{S}}$ and $t_G\colon G^{\mathrm{C}} \to G^{\mathrm{T}}$ specifying a correspondence relation between elements of G^{S} and elements of G^{T}. Triple graphs G and H are related by triple graph morphisms $m = (m^{\mathrm{S}}, m^{\mathrm{C}}, m^{\mathrm{T}})$: $G \to H$ [Sch94, EEE$^+$07] consisting of three morphisms $m^{\mathrm{S}}\colon G^{\mathrm{S}} \to H^{\mathrm{S}}, m^{\mathrm{C}}\colon G^{\mathrm{C}} \to H^{\mathrm{C}}$ and $m^{\mathrm{T}}\colon G^{\mathrm{T}} \to H^{\mathrm{T}}$ that preserve the associated correspondences, i.e., $m^{\mathrm{S}} \circ s_G = s_H \circ m^{\mathrm{C}}$ and $m^{\mathrm{T}} \circ t_G = t_H \circ m^{\mathrm{C}}$. Therefore, analogously to attributed graphs and morphisms in Sec. 2.2.1 and Def. 2.3, an attributed triple graph $G = (G^{\mathrm{S}} \xleftarrow{s_G} G^{\mathrm{C}} \xrightarrow{t_G} G^{\mathrm{T}})$ is defined by attributed graphs $G^{\mathrm{S}}, G^{C}, G^{\mathrm{T}}$ and attributed graph morphisms s_G, t_G. An attributed triple graph morphism

$f = (f^{\mathrm{S}}, f^{\mathrm{C}}, f^{\mathrm{T}})\colon G \to H$ between two attributed triple graphs G and H is defined by three attributed graph morphisms $f^{\mathrm{S}}, f^{\mathrm{C}}, f^{\mathrm{T}}$. Furthermore, analogously to typed attributed graphs and morphisms (with node type inheritance) in Sec. 2.2.1, Def. 2.4, and Rem. 2.1, a typed attributed triple graph over attributed triple graph TG as attributed triple type graph is defined by an attributed triple graph G together with an attributed triple graph morphism $type_G\colon G \to TG$. A typed attributed triple graph morphism $f\colon G \to H$ between typed attributed triple graphs G and H is an attributed triple graph morphism f such that $(type_H^X \circ f^X = type_G^X)_{X \in \{\mathrm{S},\mathrm{C},\mathrm{T}\}}$. The category $(\mathbf{ATrGraphs}_{ATGI}, \mathscr{M})$ of all typed attributed triple graphs over triple type graph $ATGI$ and all typed attributed triple graph morphisms with node type inheritance is $\mathscr{M}$-adhesive where according to Sec. 2.2.2 and Rem. 2.3, triple graph $\mathscr{M}$-($\mathscr{O}$-)morphisms are componentwise $\mathscr{M}$-($\mathscr{O}$-) morphisms in $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ (cf. Def. 3.4 & Thm. 7.2 in [EEGH15]). Analogously to the definition of rules in Sec. 2.2.4 and Def. 2.21 and according to [GEH11] and Def. 3.8 in [EEGH15], a triple rule $tr = (\overline{tr}\colon L \hookrightarrow R, ac_L)$ is given by a triple graph $\mathscr{M}$-morphism $\overline{tr}$ and an application condition ac_L over L. Thus, triple rules are non-deleting and specify how a given consistently integrated model (triple graph) can be extended simultaneously on all three components source, correspondence and target yielding again a consistently integrated model. Analogously to (direct) transformations in Sec. 2.2.4 and Def. 2.22, for a given triple graph G, triple rule $tr = (\overline{tr}\colon L \hookrightarrow R, ac_L)$ and triple graph match-morphism $m\colon L \to G$ with $m \models ac_L$, (direct) triple graph transformations $G \xRightarrow{(tr,m)} H$ via tr and m are defined (cf. Def. 3.8 in [EEGH15]). Analogously to grammars in Sec. 2.2.4 and Def. 2.23, a triple graph grammar $TGG = (S, TR)$ consists of a triple start graph S and a set TR of triple rules, and generates the triple graph language of consistently integrated models $\mathscr{L}(TGG) = \{G \mid \exists$ triple graph transformation $S \overset{*}{\Rightarrow} G$ via $TR\}$ with consistent source and target languages $\mathscr{L}(TGG)^{\mathrm{S}} = \{G^{\mathrm{S}} \mid (G^{\mathrm{S}} \leftarrow G^{\mathrm{C}} \to G^{\mathrm{T}}) \in \mathscr{L}(TGG)\}$ and $\mathscr{L}(TGG)^{\mathrm{T}} = \{G^{\mathrm{T}} \mid (G^{\mathrm{S}} \leftarrow G^{\mathrm{C}} \to G^{\mathrm{T}}) \in \mathscr{L}(TGG)\}$ (cf. Def. 3.12 in [EEGH15]).

Visual Notation As depicted in Sec. 2.2.4 and Fig. 2.8, the three components of triple graphs are visualised in three separate boxes. According to the conventions for the visual notation of graphs in Sec. 2.2.1, the mapping of graph elements along correspondence morphisms s_G and t_G of triple graphs G correspond to their naming in visual notation. Additionally, the conventions for the visual notation of rules in Sec. 2.2.4 are also used for triple rules. For example, each triple rule in Sec. 2.2.4 and Fig. 2.8 visualises the different domains of UML class diagrams (left boxes), correspondences (boxes inbetween) and relational database models (right boxes) in separate boxes. Furthermore, each rule adds those graph elements that are marked with ++ while all unmarked elements are preserved when being applied. Moreover, for example in rule 3 node 1 : CT is mapped to nodes 1 : Class and 1 : Table along the correspondence morphisms, respectively.

Example 2.5 (Triple Graph, Triple Type Graph & Triple Graph Grammar (CD2RDBM)) *Attributed triple graph $G = (CD \xleftarrow{s} C \xrightarrow{t} RDBM)$ in Sec. 2.2.1 and Fig. 2.4 is typed over attributed triple type graph $TG = (TG_{CD} \leftarrow TG_C \to TG_{RDBM})$ via attributed triple graph type morphism $(type_{CD}, type_C, type_{RDBM})\colon G \to TG$. According to the type graph, each* Attr*ibute in CDs correspond to a* Column *in RDBMs via a node of type* AC*, each* Class *in CDs correspond to a* Table *in RDBMs via a node of type* CT*, and each* DataType *in CDs correspond to a* ColumnType *in RDBMs via a node of type* TT*. Therefore, node* 1 : DataType *in source graph CD corresponds to node* 1 : ColumnType *in target graph RDBM via node* 1 : TT *in correspondence graph C, node* 2 : Attr *in CD corresponds to node* 1 : Column *in RDBM via node* 2 : AC *in C, and node* 4 : Class *in CD corresponds to node* 4 : Table *in RDBM via node* 4 : CT *in C. Sec. 2.2.4 and Fig. 2.8 depicts the triple rules for creating consistently integrated models of UML class diagrams (CDs)* *together with corresponding relational database models (RDBMs). The rules are all typed over TG. Given a triple graph with class diagram CD' in the source and database model $RDBM'$ in the target such that CD' contains a* DataType *with corresponding* ColumnType *in $RDBM'$, then the application of triple rule* 1 *extends the triple graph simultaneously on all three components in the sense that it simultaneously adds a* Class *to CD' together with a corresponding* Table *to $RDBM'$ both of* name n *but only if CD' does not already contain a class of the same name (cf. NAC). Furthermore, the table is equipped with a dedicated* Column *as primary key (*pkey*) of* type ColumnType*. Rule* 2 *simultaneously adds a* DataType *to CDs together with a corresponding* ColumnType *to RDBMs, both of* name n*. Rule* 3 *simultaneously adds an* Attribute *to CDs together with a corresponding* Column *to RDBMs, both of* name n*, and assigns both to an existing class in CDs and the corresponding table in RDBMs. Rule* 4 *adds a* Constant *modifier to an existing attribute in CDs but only if the attribute does not already have a modifier (cf. NAC). Rule* 5 *simultaneously assigns an existing data type to an existing attribute as type in CDs and the corresponding column type to the corresponding column as type in RDBMs but only if the column does not already have a type (cf. NAC). Rule* 6 *simultaneously assigns an existing class to an existing attribute as type in CDs and the corresponding table to the corresponding column as foreign key (*fkey*) in RDBMs but only if the column does not already have a type (cf. NAC), i.e., the type of the primary key column (*pkey*) of the table is additionally assigned to the column as type. The TGG $CD2RDBM = (\varnothing, \{1,2,3,4,5,6\})$ for transforming CDs into RDBMs is given by the empty triple start graph $\varnothing$ together with triple rules* 1 *to* 6*, i.e., the TGG is typed over TG. Triple graph $G = (CD \xleftarrow{s} C \xrightarrow{t} RDBM)$ in Sec. 2.2.1 and Fig. 2.4 can be obtained via direct triple graph transformations $\varnothing \xRightarrow{(2,-)} G_1 \xRightarrow{(1,-)} G_2 \xRightarrow{(3,-)} G_3 \xRightarrow{(4,-)} G_4 \xRightarrow{(5,-)} G$ via triple rules* 1 *to* 5*, i.e., $G \in \mathscr{L}(CD2RDBM), CD \in \mathscr{L}(CD2RDBM)^{\mathrm{S}}$ and $RDBM \in \mathscr{L}(CD2RDBM)^{\mathrm{T}}$.* △

Remark 2.16 (Meta-Modelling & Model Transformation) *As discussed in Sec. 2.1.1, a model transformation between DSLs $\mathscr{L}(D_1)$ and $\mathscr{L}(D_2)$ transforms models from language $\mathscr{L}(D_1)$ in source domain D_1 to language $\mathscr{L}(D_2)$ in target domain D_2 where each DSL is defined by a meta-model in the corresponding domain. In the given context of graph transformations, a meta-model is defined by a type graph together with a set of graph constraints. Therefore, a DSL $\mathscr{L}(D)$ in domain D is given by all graphs that are typed over the domain type graph and that satisfy the domain constraints (cf. Sec. 3.1 and Def. 3.1). The attributed triple type graph $TG = (TG_{CD} \leftarrow TG_C \rightarrow TG_{RDBM})$ in Sec. 2.2.1 and Fig. 2.4 contains both the type graph TG_{CD} for the domain of class diagrams (CDs) and type graph TG_{RDBM} for the domain of relational database models (RDBMs) together with type graph TG_C for correspondences between both. Additionally, Sec. 2.2.3 and Fig. 2.6 represents the graph constraints for the domain of CDs (constraints for RDBMs can be defined analogously). Thus, for the model transformation CD2RDBM from CDs to RDBMs and in view of Sec. 2.1.1 and Fig. 2.3 (a) and (c), a graph M (M') conforms to the meta-model in domain CD (RDBM), if M (M') is typed over TG_{CD} (TG_{RDBM}) and satisfies the CD (RDBM) constraints. Graph CD in Fig. 2.4 conforms to the CD meta-model and graph RDBM conforms to the RDBM meta-model. The transformation language of CD2RDBM is given by the formalism of TGGs and contains all TGGs that conform to the meta-models in domains CD and RDBM, i.e., all TGGs that are typed over triple type graph TG and that satisfy the domain constraints. According to Ex. 2.5, the TGG CD2RDBM in Sec. 2.2.4 and Fig. 2.8 is in (conforms to) the transformation language of CD2RDBM and therefore, is a valid transformation specification. Transformation CD2RDBM may take graph CD in Fig. 2.4 as input and outputs triple graph $(CD \xleftarrow{s} C \xrightarrow{t} RDBM)$ with correspondences C if being executed based on the TGG in Fig. 2.8. In Sec. 2.3.2 and Def. 2.28, we review the execution of transformations based on model transformation sequences and a given TGG in more detail.* △

2.3.2 Model Transformations based on TGGs

A model transformation $MT\colon \mathscr{L}(D_1) \Rrightarrow \mathscr{L}(D_2)$ is specified by a TGG (cf. Sec. 2.1.1). In the following we review the existing concept of executing model transformations from [EEGH15] based on model transformation sequences via operational rules of a given TGG, called forward translation triple rules. According to Def. 2.26, the operational forward translation rules of a given TGG are derived from the set of triple rules of the TGG by an automatic construction [SK08, HEGO10]. For each triple rule *tr*, the construction yields a corresponding forward translation triple rule tr_{FT} which is identical to *tr* on the correspondence and target components, i.e., tr_{FT} creates the same graph elements as *tr* in the correspondence and target parts. For the source part, tr_{FT} does not create elements but already contains the created source elements of *tr* in the left-hand side. Each source graph element in tr_{FT} is equipped with a translation attribute with attribute value false (**F** - not yet transformed) for those elements that are created by *tr* and attribute value true (**T** - has been already transformed) for all other elements (cf. Rem. 2.17). Therefore, the application of a forward translation rule tr_{FT} is the transformation of all source elements that are marked with **F** to corresponding target elements. Furthermore, rule tr_{FT} updates all translation attribute values from **F** to **T** in order to mark that the corresponding elements has been transformed. According to Def. 2.28, a forward model transformation is executed by applying operational forward translation rules successively in so called model transformation sequences. Thus, the execution of a model transformation corresponds to the abstract description in Sec. 1.3 where the update of the translation attribute values from **F** to **T** corresponds to the marking of graph elements to keep track of the elements that already have been transformed during the execution. The backward case of transforming models from $\mathscr{L}(D_1)$ in source domain D_1 to $\mathscr{L}(D_2)$ in target domain D_2 via backward translation rules is defined analogously and omitted in the following. For technical details we refer to Chapter 7 in [EEGH15].

The extension of rules with translation attributes from Rem. 2.17 is used for the definition of forward translation rules in Def. 2.26 and for the definition of marking rules in Sec. 3.2 that are part of verifying domain completeness.

Remark 2.17 (Graphs with Translation Attributes) *Given an attributed graph $AG = (G,D)$ and a subset $M \subseteq G$ of its elements (nodes, edges or attributes), we call AG'* a graph with translation attributes over *AG if it extends AG by one Boolean-valued translation attribute for each element in M. The translation attribute for a node or edge is specified by an attribute tr. The translation attribute for an attribute a of a node or edge is specified by an attribute tr_a. With $AG \oplus Att_M^{\mathbf{T}}$ we denote the graph with translation attributes over AG which extends AG by a translation attribute for each element in $M \subseteq G$, and all these translation attributes are set to* **T**. *Similarly, $AG \oplus Att_M^{\mathbf{F}}$ denotes adding to AG all these translation attributes, but this time they are set to* **F**. *With $Att^x(AG), x \in \{\mathbf{F},\mathbf{T}\}$ we denote $AG \oplus Att_G^x$. For technical details we refer to Sec. 7.4.1 in [EEGH15].* △

According to Def. 2.25, for forward translation rules with application conditions *ac*, each element in *ac* that is not in the premise of *ac* also need to be extended by a translation attribute of value **T**. For triple rules, X restricts the extension to elements of specific triple components (source, correspondence and target).

Definition 2.25 (**T**-Extension of Application Conditions (Def. 7.28 in [EEGH15])) Given an application condition ac_P over premise graph P, a graph P' with translation attributes over P and a subset of triple components $X \subseteq \{S,C,T\}$, then the **T**-extension $tExt(ac_P,P',X)$ of ac_P is inductively defined by:

- $tExt(\mathbf{true}, P', X) = \mathbf{true}$
- $tExt(\exists(a = (inc_P, a_D) \colon P \to C, ac_C), P', X) = \exists(a_E \colon P' \to C', tExt(ac_C, C', X))$ with
  1. $C' = P' +_P C \oplus \cup_{x \in X}(Att^{\mathbf{T}}_{C^x \setminus P^x})$, and
  2. $a_E = (inc'_P, a_D)$ with algebra homomorphism a_D on the data part and inclusion inc'_P on the graph part as derived from inc_P.
- $tExt(\neg(ac'_P), P', X) = \neg(tExt(ac'_P, P', X))$
- $tExt(ac_{P,1} \wedge ac_{P,2}, P', X) = tExt(ac_{P,1}, P', X) \wedge tExt(ac_{P,2}, P', X)$
- $tExt(ac_{P,1} \vee ac_{P,2}, P', X) = tExt(ac_{P,1}, P', X) \vee tExt(ac_{P,2}, P', X)$ △

According to [EEGH15], for model transformations and synchronisations, we restrict the application conditions of triple rules to S-consistent application conditions.

Remark 2.18 (S-consistent Application Conditions) *According to Def. 7.8 in [EEGH15], an application condition ac_P is source consistent (S-consistent), if it can be decomposed into a semantically equivalent conjunction $ac_P \equiv ac_S \wedge ac_F$ such that ac_S does restrict the source component only and ac_F does restrict the correspondence and target components only, i.e., relations of restricting elements between the correspondence and source parts in ac_P may be problematic. All application conditions in running examples of this thesis are S-consistent.* △

Forward translation rules are used in Def. 2.28 for executing model transformations and in Sec. 4.1 and Thm. 4.2 for verifying the domain completeness of model transformations.

Definition 2.26 (Forward Translation Rule (Def. 7.29 in [EEGH15])) Given a triple rule $tr = ((tr^{\mathrm{S}}, tr^{\mathrm{C}}, tr^{\mathrm{T}}) \colon L = (L^{\mathrm{S}} \xleftarrow{s_L} L^{\mathrm{C}} \xrightarrow{t_L} L^{\mathrm{T}}) \to R = (R^{\mathrm{S}} \xleftarrow{s_R} R^{\mathrm{C}} \xrightarrow{t_R} R^{\mathrm{T}}), ac_L)$ with S-consistent application condition ac_L over L, then *the forward translation rule tr_{FT} of tr* is given by $tr_{FT} = (L_{FT} \xleftarrow{l_{FT}} K_{FT} \xrightarrow{r_{FT}} R_{FT}, ac_{FT})$ with: 1. $L_{FT} = (R^{\mathrm{S}} \xleftarrow{tr^{\mathrm{S}} \circ s_L} L^{\mathrm{C}} \xrightarrow{t_L} L^{\mathrm{T}}) \oplus Att^{\mathbf{T}}_{(tr^{\mathrm{S}}(L^{\mathrm{S}}) \leftarrow \varnothing \to \varnothing)} \oplus Att^{\mathbf{F}}_{(R^{\mathrm{S}} \leftarrow \varnothing \to \varnothing) \setminus (tr^{\mathrm{S}}(L^{\mathrm{S}}) \leftarrow \varnothing \to \varnothing)}$, 2. $K_{FT} = (R^{\mathrm{S}} \xleftarrow{tr^{\mathrm{S}} \circ s_L} L^{\mathrm{C}} \xrightarrow{t_L} L^{\mathrm{T}}) \oplus Att^{\mathbf{T}}_{(tr^{\mathrm{S}}(L^{\mathrm{S}}) \leftarrow \varnothing \to \varnothing)}$, 3. $R_{FT} = (R^{\mathrm{S}} \xleftarrow{s_R} R^{\mathrm{C}} \xrightarrow{t_R} R^{\mathrm{T}}) \oplus Att^{\mathbf{T}}_{(R^{\mathrm{S}} \leftarrow \varnothing \to \varnothing)}$, 4. l_{FT} and r_{FT} are the induced inclusions, and 5. $ac_{FT} = tExt(ac_L, L_{FT}, \{S\})$. With TR_{FT} we denote the set of forward translation rules tr_{FT} of all triple rules $tr \in TR$ for a given set of triple rules TR. △

Definition 2.27 (Complete Forward Translation Sequence (Def. 7.33 in [EEGH15])) A forward translation sequence $G_0 \xRightarrow{tr^*_{FT}} G_n$ via TR_{FT} with almost injective matches is *complete*, if no further forward translation rule is applicable and all translation attributes in G_n are set to $\mathbf{T}$. △

Definition 2.28 (Model Transformation based on Forward Translation Rules (Def. 7.34 in [EEGH15])) Given a triple type graph $TG = (TG^{\mathrm{S}} \leftarrow TG^{\mathrm{C}} \to TG^{\mathrm{T}})$ and a set TR of triple rules typed over TG, then a *model transformation sequence* $(G^S, G'_0 \xRightarrow{tr^*_{FT}} G'_n, G^T)$ based on forward translation rules TR_{FT} from a source graph G^S in the source domain to a target graph G^T in the target domain consists of a complete forward translation sequence $G'_0 \xRightarrow{tr^*_{FT}} G'_n$ typed over $TG' = TG \oplus Att^{\mathbf{F}}_{(TG^S \leftarrow \varnothing \to \varnothing)} \oplus Att^{\mathbf{T}}_{(TG^S \leftarrow \varnothing \to \varnothing)}$ based on TR_{FT} with $G'_0 = (Att^{\mathbf{F}}(G^S) \leftarrow \varnothing \to \varnothing)$ and $G'_n = (Att^{\mathbf{T}}(G^S) \leftarrow G^C \to G^T)$. A *model transformation $MT \colon \mathscr{L}(TG^S) \Rrightarrow \mathscr{L}(TG^T)$* based on TR_{FT} is defined by all model transformation sequences as above with $G^S \in \mathscr{L}(TG^S)$ and $G^T \in \mathscr{L}(TG^T)$ (cf. Sec. 3.1 and Def. 3.1 for $\mathscr{L}(TG^{\mathrm{S}})$ and $\mathscr{L}(TG^{\mathrm{T}})$). △

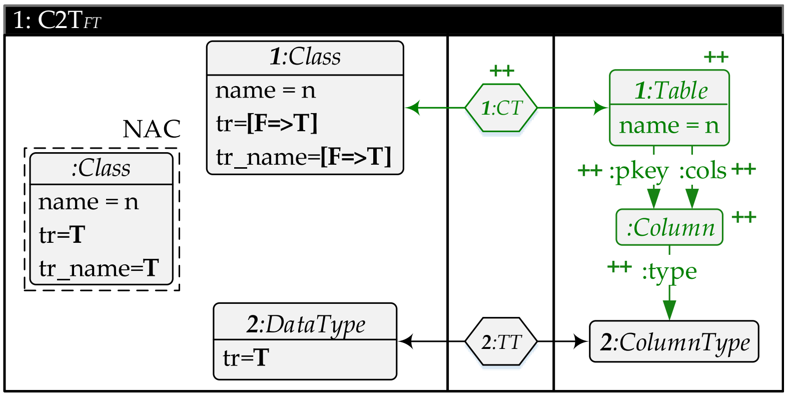


Figure 2.10: Forward Translation Rule

Example 2.6 (Forward Translation Rule & Model Transformation) *Rule* C2T$_{FT}$ *in Fig. 2.10 is the forward translation rule of triple rule* C2T *in Sec. 2.2.4 and Fig. 2.8. Node* 1 : Class *and attribute* name *are created by rule* C2T *in the source and therefore, are initially marked with translation attribute* **F** *and updated to* **T** *in rule* C2T$_{FT}$*, denoted by* $[\mathbf{F} => \mathbf{T}]$*. All other source and NAC elements of* C2T *are initially marked with* **T** *in* C2T$_{FT}$ *and remain unchanged. Furthermore, the correspondence and target components of* C2T$_{FT}$ *are identical to those of rule* C2T*. Note that a forward translation rule does also contain the application condition of the target component but without any markings if existent in the underlying triple rule as it is the case for triple rules* TD2T *and* TC2T*. Graph CD in Sec. 2.2.1 and Fig. 2.4 can be transformed to graph RDBM via model transformation sequence* $(CD, G'_0 \xRightarrow{\mathsf{DT2CT}_{FT}(\text{“}INT''\text{)}} G'_1 \xRightarrow{\mathsf{C2T}_{FT}(\text{“}Person''\text{)}} G'_2 \xRightarrow{\mathsf{A2C}_{FT}(\text{“}DOB''\text{)}} G'_3 \xRightarrow{\mathsf{MC}_{FT}} G'_4 \xRightarrow{\mathsf{TD2T}_{FT}} G'_5, RDBM)$ *based on the forward translation rules of the triple rules in Sec. 2.2.4 and Fig. 2.8 with integrated model* $G'_5 = (Att^{\mathbf{T}}(CD) \leftarrow G^C \rightarrow RDBM)$*. The model transformation CD2RDBM is given by all corresponding model transformation sequences based on the forward translation rules of the triple rules in Sec. 2.2.4 and Fig. 2.8.* △

General Assumption Note that the definition of forward translation rules is based on attributions in attributed graphs (cf. Rem. 2.17 and Def. 2.26). Therefore, we assume category $(\mathbf{ATrGraphs}_{ATGI}, \mathscr{M})$ for model transformations and synchronisations. According to the general assumption for model transformations based on TGGs in Sec. 7.1 in [EEGH15] and analogously to the general assumption for rule applications in Sec. 2.2.4, we assume that triple rules are applied based on transformations via almost injective matches $m \in \mathscr{O}$ and where all internal morphisms of application conditions are almost injective (cf. Sec. 2.3.1 for $\mathscr{O}$-morphisms in $(\mathbf{ATrGraphs}_{ATGI}, \mathscr{M})$). In the context of model transformations and synchronisations, we generally assume the empty triple graph $\varnothing$ as start graph for TGGs and that all application conditions of triple rules are S-consistent.

2.3.3 Model Synchronisations based on TGGs

Given a triple type graph $TG = (TG^S \leftarrow TG^C \rightarrow TG^T)$ with source domain $\mathscr{L}(TG^S)$ and target domain $\mathscr{L}(TG^T)$ (cf. Sec. 3.1 and Def. 3.1 for $\mathscr{L}(TG^S)$ and $\mathscr{L}(TG^T)$). A source (target) model update δ on source graph $G \in \mathscr{L}(TG^S)$ (target graph $G \in \mathscr{L}(TG^T)$) is a span of inclusions $\delta = (G \overset{u_1}{\hookleftarrow} H \overset{u_2}{\hookrightarrow} G'), u_1, u_2 \in \mathscr{M}$ with $G, H, G' \in \mathscr{L}(TG^S)$ $(G, H, G' \in \mathscr{L}(TG^T))$. Elements $G \setminus u_1(H)$ are deleted in G whereas elements $G' \setminus u_2(H)$ are added to G. With Δ^S (Δ^T) we

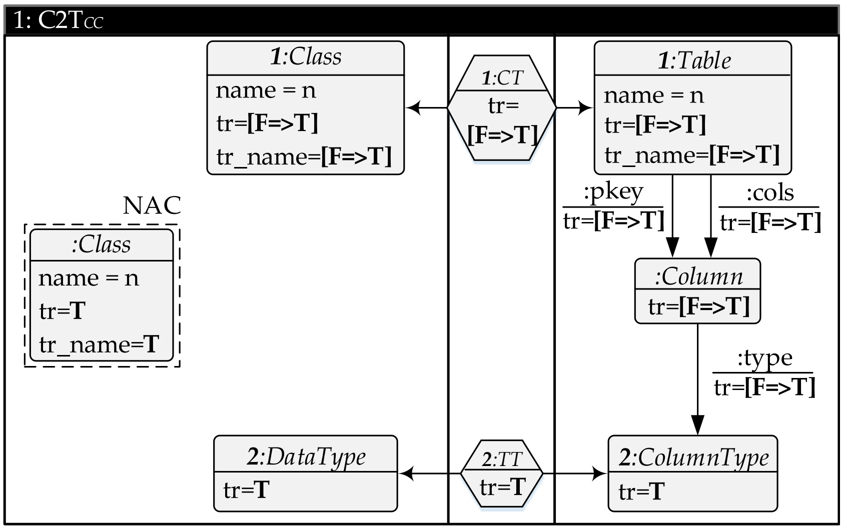


Figure 2.11: Consistency Creating Rule

denote the set of all source (target) model updates. A model synchronisation is a propagation of model updates from the source to the target domain via forward propagation operation *fPpg*. Operation *fPpg* takes a triple graph $(G^{\mathrm{S}} \leftarrow G^{\mathrm{C}} \rightarrow G^{\mathrm{T}})$ typed over *TG* together with a source model update $\delta^{\mathrm{S}} = (G^{\mathrm{S}} \leftarrow H^{\mathrm{S}} \rightarrow G'^{\mathrm{S}}) \in \Delta^{\mathrm{S}}$ on G^{S} as input and outputs a target model update $\delta^{\mathrm{T}} = (G^{\mathrm{T}} \leftarrow H^{\mathrm{T}} \rightarrow G'^{\mathrm{T}}) \in \Delta^{\mathrm{T}}$ on G^{T} together with a triple graph $(G'^{\mathrm{S}} \leftarrow G'^{\mathrm{C}} \rightarrow G'^{\mathrm{T}})$ typed over *TG* that interrelates the results G'^{S} and G'^{T} of both updates via correspondence G'^{C}. This corresponds to the informal description of model synchronisations in Sec. 2.1.2. The backward case of propagating updates from the target to the source domain via backward propagation operation *bPpg* is defined analogously and omitted in the following. According to Def. 9.18 in [EEGH15], the synchronisation operation *fPpg* is a composition of three auxiliary operations *fAln*, *Del* and *fAdd*. For technical details we refer to Chapter 9 in [EEGH15]. Operation *Del* relies on operational consistency creating (CC) rules [HEO$^+$15]. Similarly to forward translation rules in Sec. 2.3.2 and Def. 2.26, the CC rule tr_{CC} of a given triple rule *tr* does not create elements in the sense that it already contains all elements of *tr* including the created elements in the left-hand side and furthermore, each element that is created by *tr* is initially marked with translation attribute **F** und updated to **T**, written [**F** => **T**], while all other elements are initially marked with **T** and remain unchanged. Rule $\mathsf{C2T}_{CC}$ in Fig. 2.11 is the CC rule of triple rule C2T in Sec. 2.2.4 and Fig. 2.8. For CC rules we review the existing result in Fact 2.1 which is used to prove Sec. 3.2 and Lem. 3.2.

Definition 2.29 (Consistency Creating (CC) Rule (Def. 7.44 in [EEGH15])) Given a triple rule $tr = (L \rightarrow R, ac_L)$, then *the consistency creating rule* tr_{CC} *of* *tr* is given by $tr_{CC} = (L_{CC} \xleftarrow{l_{CC}} K_{CC} \xrightarrow{r_{CC}} R_{CC}, ac_{CC})$ with: 1. $L_{CC} = (R \oplus Att_L^{\mathbf{T}} \oplus Att_{R\setminus L}^{\mathbf{F}})$, 2. $K_{CC} = (R \oplus Att_L^{\mathbf{T}})$, 3. $R_{CC} = (R \oplus Att_L^{\mathbf{T}} \oplus Att_{R\setminus L}^{\mathbf{T}})$, 4. l_{CC} and r_{CC} are the induced inclusions, and 5. $ac_{CC} = tExt(ac_L, L_{CC}, \{S, C, T\})$. With TR_{CC} we denote the set of consistency creating rules tr_{CC} of all triple rules $tr \in TR$ for a given set of triple rules *TR*. △

Remark 2.19 (Meta-Modelling & Model Synchronisation) *Given the triple type graph* $TG = (TG_{CD} \leftarrow TG_C \rightarrow TG_{RDBM})$ *from Fig. 2.4 with triple graph* $G = (CD \leftarrow C \rightarrow RDBM) \in \mathscr{L}(TG)$ *and source model update* $\delta = (CD \xleftarrow{u_1} H \xrightarrow{u_2} CD')$ *in Fig. 2.12, then* $fPpg(G, \delta) = (G', \delta')$

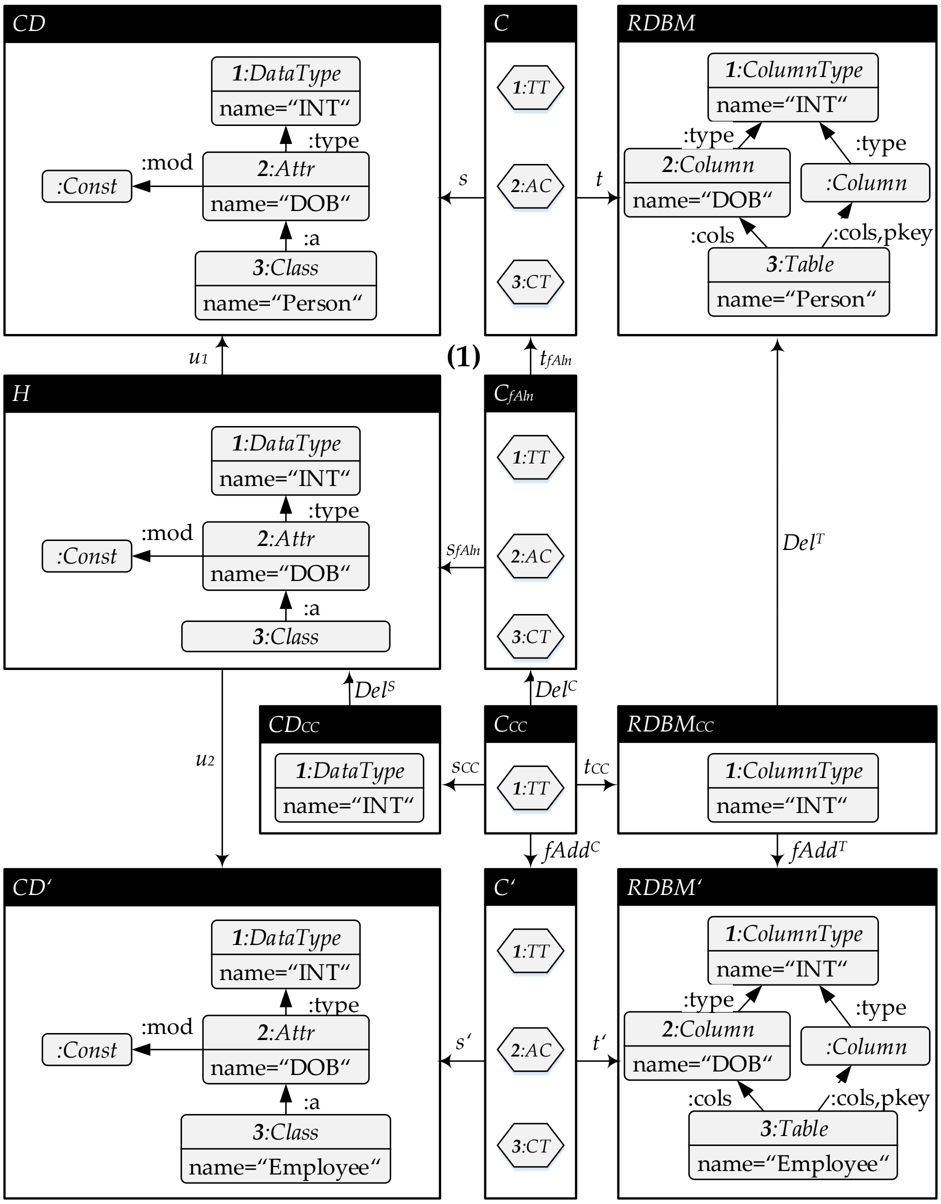


Figure 2.12: Model Synchronisation via Forward Propagation Operation *fPpg*

with $G' = (CD' \xleftarrow{s'} C' \xrightarrow{t'} RDBM') \in \mathscr{L}(TG)$ and target model update $\delta' = (RDBM \xleftarrow{Del^{T}} RDBM_{CC} \xrightarrow{fAdd^{T}} RDBM')$. Target update δ' on RDBM reflects the change of source update δ on CD (the name *of the* Class *is changed from* Person *to* Employee*) in the target domain. The output (G', δ') is obtained by three sequential operations fAln, Del and fAdd: Alignment operation fAln creates pullback (1) in order to align deletions of update δ to the correspondence component and yields triple graph $G_1 = (H \xleftarrow{s_{fAln}} C_{fAln} \xrightarrow{t \circ t_{fAln}} RDBM)$. Operation Del creates the maximal consistently integrated triple sub-graph of G_1 by creating the graph $Att^{\mathbf{F}}(G_1)$ with translation attributes over G_1 and applying CC rules TR_{CC} as long as possible afterwards. The maximal consistently integrated sub-graph is given by all* **T***-marked elements, i.e., by triple graph $G_2 = (CD_{CC} \xleftarrow{s_{CC}} C_{CC} \xrightarrow{t_{CC}} RDBM_{CC})$ with inclusion $(Del^{S}, Del^{C}, Del^{T}) \colon G_2 \to G_1$ (the translation attributes are omitted in Fig. 2.12). Operation fAdd adds those elements to G_2 that* *are created by update δ leading to triple graph $G_3 = (CD' \xleftarrow{u_2 \circ Del^{S} \circ s_{CC}} C_{CC} \xrightarrow{t_{CC}} RDBM_{CC})$ where additionally all elements that are creatd by δ are marked with translation attributes $\mathbf{F}$. Finally, a complete forward translation sequence $G_3 \xRightarrow{tr^*_{FT}} G'$ leads to output (G', δ').* △

Fact 2.1 (Equivalence of Triple and Extended Consistency Creating Sequences (Fact 10 in [HEO^{+}11])) *Let $TGG = (\varnothing, TR)$ be a triple graph grammar typed over triple type graph ATGI with empty start graph $\varnothing$ and derived consistency creating rules TR_{CC} of triple rules TR. Let $G \in \mathscr{L}(ATGI)$ be a graph according to Sec. 3.1 and Def. 3.1, then the following are equivalent for almost injective matches $m \in \mathscr{O}$.*

- *There exists a consistency creating sequence $Att^{\mathbf{F}}(G) \xRightarrow{tr^*_{CC}} G \oplus Att^{\mathbf{T}}_{G_k} \oplus Att^{\mathbf{F}}_{G \setminus G_k}$ via consistency creating rules TR_{CC}.*
- *There exists a triple graph transformation $\varnothing \xRightarrow{tr^*} G_k$ via TR with injective embedding $f \colon G_k \to G$.* △

General Assumption Note that the definition of CC rules is based on attributions in attributed graphs. Therefore, we assume all general assumptions from Sec. 2.3.2.

2.4 Properties of Model Transformations & Synchronisations ("Classical" Completeness & Correctness)

Model transformations and synchronisations based on TGGs share the following important properties: 1. efficiency, 2. TGGs allow intuitive and maintainable tranformation specifications with sufficient expressiveness, 3. bidirectionality, 4. type consistency, 5. termination, 6. functional behaviour (termination + confluence), and 7. information preservation (cf. Sec. 3.1 in [EEGH15]). Two further essential properties for the results in Sec. 4.1 and Chap. 4 are syntactical correctness and completeness referred to as "classical" correctness and completeness to not confuse with domain completeness. Syntactical correctness means that for each input in the source domain, if the model transformation (synchronisation) leads to an outpout, then the output conforms to the meta-model of the target domain. Syntactical completeness means that the model transformation (synchronisation) can be applied on each model (model update) in the source domain.

Definition 2.30 ("Classical" Syntactical Correctness and Completeness of Model Transformations (Def. 8.3 in [EEGH15])) A model transformation $MT \colon \mathscr{L}(TG^S) \Rrightarrow \mathscr{L}(TG^T)$ based on forward rules is

1. *syntactically correct* if for each model transformation sequence $(G^S, G_0 \xRightarrow{tr^*_F} G_n, G^T)$ there is $G \in \mathscr{L}(TGG)$ with $G = (G^S \leftarrow G^C \rightarrow G^T)$ implying further that $G^S \in \mathscr{L}(TGG)^S$ and $G^T \in \mathscr{L}(TGG)^T$, and it is
2. *syntactically complete* if for each $G^S \in \mathscr{L}(TGG)^S$ there is $G = (G^S \leftarrow G^C \rightarrow G^T) \in \mathscr{L}(TGG)$ with a model transformation sequence $(G^S, G_0 \xRightarrow{tr^*_F} G_n, G^T)$ and $G_n = G$. The backward direction is defined analogously. △

Definition 2.31 ("Classical" Syntactical Correctness and Completeness of Model Synchronisations (Fig. 9.5 in [EEGH15])) A forward model synchronisation via *fPpg* is

1. *syntactically correct* if for all consistently integrated models $M = (M^{\mathrm{S}} \leftarrow M^{\mathrm{C}} \rightarrow M^{\mathrm{T}}) \in \mathscr{L}(TGG)$ and source model updates $u \colon M^{\mathrm{S}} \leftarrow H^{\mathrm{S}} \rightarrow M'^{\mathrm{S}}, u' \in \Delta^{S}, M'^{\mathrm{S}} \in \mathscr{L}(TGG)^{\mathrm{S}}$ it holds

that $fPpg(M, id^S) = (M, id^T)$ and $fPpg(M, u) = (M', u')$ such that $M' = (M'^{\mathrm{S}} \leftarrow M'^{\mathrm{C}} \rightarrow M'^{\mathrm{T}})$ and $M' \in \mathscr{L}(TGG)$, if $fPpg$ yields a result, and it is

2. *syntactically complete* if $fPpg$ can be applied on each input integrated models $M \in \mathscr{L}(TGG)$ and source model updates $u \in \Delta^S$ and it always yields a result for any valid input. △

Remark 2.20 ("Classical" Syntactical Correctness and Completeness) *According to Thm. 8.4 and Cor. 8.5 in [EEGH15], each model transformation MT based on forward (translation) rules is "classically" syntactically correct and complete. For forward translation rules, each model transformation sequence $(G^S, G_0 \xRightarrow{tr_F^*} G_n, G^T)$ based on forward rules in Def. 2.30 is substituted by a corresponding model transformation sequence $(G^S, G'_0 \xRightarrow{tr_{FT}^*} G'_n, G^T)$ based on forward translation rules. According to Thm. 9.25 in [EEGH15], $fPpg$ is syntactically correct and complete for kernel-grounded and deterministic sets of operational rules TR_{CC} and TR_{FT}.* △

Chapter 3

Domain Completeness

In Sec. 3.1, we introduce the domain completeness problem and show its undecidability in $(\mathbf{Graphs}_{TG}, \mathscr{M})$ and derivative categories of graphs with finite restrictions to the graphs, underlying type graph *TG*, grammar, constraints and application conditions. Intuitively, the fact that the domain completeness problem is undecidable means that it is impossible to construct a single algorithm which completely solves the problem in the sense that the algorithm terminates and returns the correct yes-or-no answer concerning $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$ for each input $\mathscr{L}(C)$ and $\mathscr{L}(GG)$. This led to the development of sufficient conditions that can be checked by a procedure for verifying domain completeness in $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ and underlying categories. It is guaranteed that the procedure terminates under certain assumptions. Note that the conditions are sufficient only without being necessary, i.e., the proposed verification technique is an under-approximation approach for solving the domain completeness problem in the sense that the approach does not yield false positives but may yield false negatives. Thus, if the conditions are fulfilled, then domain completeness holds but if the conditions are not fulfilled, then domain completeness does not necessarily not hold but may also hold. Therefore, we can give a concrete answer only to a subset of all constraint-grammar pairs for which domain completeness holds.

Sec. 3.2 presents the verification technique. Sec. 3.3 presents the limitations of the approach. Sects. 3.4 and 3.5 presents various extensions of the approach. Sec. 3.4 introduces the notion of recursive constraints for enabling the verification of domain completeness in view of infinite constraints which describe recursive graph patterns. Finally, in Sec. 3.5 we consider the verification of domain completeness under restrictions of the domain type graph. This reflects the situation where only a subset of all constituents of the given domain is subjected to the verification.

3.1 Domain Completeness Problem & Undecidability

We assume that the constituents of a domain are given by a type graph whereas valid sentences in that domain are additionally restricted by a set of graph constraints. Thus, valid sentences in a domain are expressed by typed graphs as domain models that satisfy the given domain constraints. The domain-specific language (DSL) is given by all valid sentences in that domain, i.e., by all graphs typed over the domain type graph and that satisfy the domain constraints.

Definition 3.1 ((Domain-Specific) Languages over Graph Constraints, Grammars & Type Graphs) Let *TG* be a type graph, $C = C_I \cup C_G$ be a set of nested graph constraints C_I that are designated for initial satisfaction and C_G that are designated for general satisfaction (all typed over *TG*). Moreover, let $GG = (S, P)$ be a graph grammar with start graph *S* and productions *P* (all typed over TG). With $\mathscr{L}(TG)$ we denote *the language over type graph TG* which is given by the set of all graphs that are typed over TG. With $\mathscr{L}(GG) := \{G \mid \exists S \overset{*}{\Rightarrow} G \text{ via } P\}$ we denote the *language over (TG) and grammar GG* which is given by all graphs (typed over TG) that are reachable from S by transformation sequences via productions P. With $\mathscr{L}_I(C_I) := \{G \mid G \overset{I}{\models} C_I\}$ we denote the *domain-specific language over (TG) and constraints C_I* which is given by all graphs (typed over TG) that initially satisfy constraints C_I. With $\mathscr{L}(C_G) := \{G \mid G \models C_G\}$ we denote the *domain-specific language over (TG) and constraints C_G* which is given by all graphs (typed over TG) that generally satisfy constraints C_G. We write $\mathscr{L}(C)$ short for $\mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ and denote the *domain-specific language over (TG) and constraints C*. △

Example 3.1 ((Domain-Specific) Languages) *Sec. 2.2.3 and Fig. 2.5 depicts the type graph TG of UML class diagrams as well as the constraint* $ac_P = \vee i = (1,2)(\exists(a\colon P \to C, \textbf{true}))$ *which claims that:*

1. *For initial satisfaction of ac_P - There exists a* Class *in the diagram such that the class has a* Constructor *with visibility* Protected *or* Public*.*

2. *For general satisfaction of ac_P - For each class in the diagram it holds that the class has a* Constructor *with visibility* Protected *or* Public*.*

Graph G in Fig. 2.5 is typed over TG, therefore $G \in \mathscr{L}(TG)$. Moreover, graph G both initially and generally satisfies ac_P, i.e., $G \in \mathscr{L}_I(\{ac_P\})$ and $G \in \mathscr{L}(\{ac_P\})$. Furthermore, given type graph TG_{CD}, graph CD in Sec. 2.2.1 and Fig. 2.4 typed over TG_{CD} and the rules P in Sec. 2.2.4 and Ex. 2.4 typed over TG_{CD} with grammar $GG = (\varnothing, P)$ and empty start graph $\varnothing$, then according to Ex. 2.4, CD can be created by a transformation from $\varnothing$ via rules P, i.e., $CD \in \mathscr{L}(GG)$. △

We introduce the notion of domain completeness in view of model transformations and synchronisations based on TGGs. Thus, domain completeness means that all valid sentences in a domain are completely covered by the TGG. More precisely, given a domain type graph TG, domain constraints C and a graph grammar $GG = (S,P)$ both typed over TG, then domain completeness holds if $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$, i.e., the constraints C are more restrictive than (or as restrictive as) grammar GG. Thus, the domain completeness problem is defined in terms of a language inclusion problem. The underlying question to be answered is: "Can all graphs that satisfy C be created from S by successively applying rules P?".

Definition 3.2 (Domain Completeness (Problem)) Given the languages $\mathscr{L}(C)$ and $\mathscr{L}(GG)$ over domain graph constraints C and grammar GG, respectively, and common domain type graph TG. *Domain completeness* holds if $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$. Thus, the *domain completeness problem* is defined as follows: Does it hold that $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$? △

The domain completeness problem for (non-deleting) graph grammars with nested application conditions turns out to be undecidable in general (and in particular, for (finite) typed graphs with injective matches only for rule applications) due to the expressiveness of nested conditions as application conditions for the productions in the grammars. This implies that the problem is also undecidable for derivative categories of graphs such as the (finitary) $\mathscr{M}$-adhesive category $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ of (finite) typed attributed graphs (with node type inheritance) and with almost injective matches only. Based on the existing result for transforming constraints into right application conditions [HP09], we first show that sets of constraints C can be transformed into left application conditions ac_L for productions $p = (L \leftarrow K \to R)$ such that the satisfaction of ac_L by matches $m\colon L \to G$ coincides with the satisfaction of C by G. This result is used for proving the undecidability in Thm. 3.1 afterwards.

Lemma 3.1 (Transformation of Constraints into Left Application Conditions) *Let* $(\mathbf{C},\mathcal{M})$ *be an* $\mathcal{M}$*-adhesive category with* $\mathcal{E}$*-*$\mathcal{M}$*-factorisation and* $\mathcal{M}$*-initial object I. Then, there is* a transformation *LA* from sets of conditions over I into left application conditions *for productions, such that for all sets C of conditions over I, all productions* $p = (L \leftarrow K \rightarrow R)$ *and all matches* $m\colon L \rightarrow G$ *for some G it holds that* $m \models LA(p,C)$ *if and only if* $G \models C$. △

Construction Given a production $p = (L \leftarrow K \rightarrow R)$ and a set $C = (c_j)_{j\in J}$ of conditions c_j over I, then the transformation $LA(p,C) := \wedge_{j\in J}\mathrm{A}(\overline{p},c_j)$ is defined by the transformation A from conditions into right application conditions for $\overline{p}$ as given in Thm. 5 in [HP09] together with a conjunction over all constraints $c_j \in C$ where $\overline{p} = (L \xleftarrow{id_L} L \xrightarrow{id_L} L)$.

Proof. By Thm. 5 in [HP09], for all conditions c over I, all productions $\overline{p} = (L \leftarrow K \rightarrow R)$ and all morphisms $m^*\colon R \rightarrow G$ it holds that $m^* \models \mathrm{A}(\overline{p},c) \Leftrightarrow G \models c$ $^{(*^1)}$. This result can be directly transferred from weak adhesive HLR categories to $\mathcal{M}$-adhesive categories, since, only basic HLR properties and general categorical properties are used in the proofs. Let $C = (c_j)_{j\in J}$ be a set of conditions over I, $p = (L \leftarrow K \rightarrow R)$ and $\overline{p} = (L \xleftarrow{id_L} L \xrightarrow{id_L} L)$ be productions and $m\colon L \rightarrow G$ be a match. $m \models LA(p,C) \overset{Def.\ LA(p,C)}{\Longleftrightarrow} m \models \wedge_{j\in J}\mathrm{A}(\overline{p},c_j) \overset{Sat.}{\Longleftrightarrow} m \models \mathrm{A}(\overline{p},c_j)$, for all $j \in J \overset{(*^1)}{\Longleftrightarrow} G \models c_j$, for all $j \in J \overset{Sat.}{\Longleftrightarrow} G \models C$. □

The properties that are enclosed by parentheses “(property)” are optional.

Theorem 3.1 (Undecidability of the Domain Completeness Problem) *Let C be a (finite) set of (finite) nested graph constraints (in* $\mathcal{M}$*-normal form) and* $GG = (S,P)$ *be a (non-deleting) graph grammar with a (finite) set of productions P (with start graph S being the initial graph* $\varnothing$*) and (finite left) nested application conditions (in* $\mathcal{M}$*-normal form). Furthermore, let* $\mathcal{L}(C)$ *and* $\mathcal{L}(GG)$ *be the languages over C and GG, respectively, and common type graph TG. Then,* the language inclusion problem $\mathcal{L}(C) \subseteq \mathcal{L}(GG)$ is undecidable *in general and in particular in the (finitary)* $\mathcal{M}$*-adhesive category* $(\mathbf{Graphs}_{TG},\mathcal{M})$ $((\mathbf{Graphs}_{TG,fin},\mathcal{M}_{fin}))$ *of (finite) graphs typed over (finite) type graph TG with* $\mathcal{M}$*-matching.* △

Proof. The proof is presented in appendix A.3. □

In addition to Thm. 3.1, it turns out that the domain completeness problem is also undecidable for (non-deleting) graph grammars without application conditions due to the undecidability of the satisfiability problem of constraints.

Theorem 3.2 (Undecidability of the Domain Completeness Problem) *Given the setting from Thm. 3.1 but GG be a graph grammar without application conditions. Then,* $\mathcal{L}(C) \subseteq \mathcal{L}(GG)$ is undecidable *in the same context.*

Proof. The undecidability is shown by a reduction from the undecidable satisfiability problem of finite graph constraints (cf. Cor. 9 in [HP09]). Thus, based on the proof of Thm. 3.1, for a given (finite) set of (finite) constraints C in $\mathbf{C}$ in $\mathcal{M}$-normal form it is undecidable, whether there is a graph in $\mathbf{C}$ that satisfies C $^{(*^1)}$. The reduction is given by a computable mapping from C and TG to TG' and constraints C' together with grammar $GG = (S,P)$ with start graph S and an empty set of productions $P = \varnothing$. Similarly to the proof of Thm. 3.1, type graph TG' extends TG by node $\underline{T}$ with inclusion $i_t\colon TG \rightarrow TG'$, the obvious functor $F\colon \mathbf{C} \rightarrow \mathbf{C}'$, mapping $\overline{F}(C)$ and result

$(*^3)$. The set of constraints C' is given by $C' = \overline{F}(C) \cup \{c\}$ with constraint $c = \exists(\varnothing \to \boxed{:\underline{T}}, \mathbf{true})$. Thus, language $\mathscr{L}(C')$ contains all graphs in $\mathbf{C}'$ that have at least on node of type $\underline{T}$ and that satisfy all constraints in $\overline{F}(C)$. Language $\mathscr{L}(GG)$ with $S = \varnothing$ contains the empty graph S only. It remains to show that C is satisfiable in $\mathbf{C}$ if and only if $\mathscr{L}(C') \nsubseteq \mathscr{L}(GG)$ holds in $\mathbf{C}'$. Then, assuming the decidability of the language inclusion problem would imply the decidability of the satisfiability problem leading to a contradiction by $(*^1)$.

"$\Rightarrow$" There exists $G \in \mathbf{C}$ with $G \models C$. Thus, there exists graph G' in $\mathbf{C}'$ which is $F(G)$ extended by a single node $:\underline{\mathrm{T}}$ with $G' \models \overline{F}(C)$ by $(*^3)$ and furthermore, $G' \models c$ therefore, $G' \models C'$ and $G' \in \mathscr{L}(C')$. However, $G' \notin \mathscr{L}(GG)$.

"$\Leftarrow$" By contradiction assume that C is not satisfiable in $\mathbf{C}$, i.e., for all $G \in \mathbf{C}$, $G \not\models C$. By $(*^3)$ it follows that for all $G' \in \mathbf{C}'$, $G' \not\models \overline{F}(C)$. Thus, $\mathscr{L}(C') = \varnothing$ and therefore, $\mathscr{L}(C') \subseteq \mathscr{L}(GG)$.

The set C' is finite by construction of $\overline{F}(C)$ with C being a finite set by assumption. Analogously, for all $c' \in \overline{F}(C)$, c' is finite and in $\mathscr{M}$-normal form and furthermore, c is finite and in $\mathscr{M}$-normal form. Thus, for all $c' \in C'$, c' is finite and in $\mathscr{M}$-normal form. By $P = \varnothing$, grammar GG is trivially non-deleting, P is finite, all application conditions are finite and in $\mathscr{M}$-normal form and we can restrict to $\mathscr{M}$-matching. □

3.2 Verification of Domain Completeness

This section presents the technique for verifying domain completeness approximately according to Sec. 3.1 and Def. 3.2. Effectively, this means to provide a method for showing that all graphs which satisfy the domain constraints can be created from the empty start graph $\varnothing$ via rule applications of the given graph grammar. In Def. 3.8, for a given type graph TG, graph grammar GG and set C of domain constraints both typed over TG, we introduce the general notion of C-extension completeness of graph languages $\mathscr{L}(GG)$ over TG and graph grammar GG. C-extension completeness of language $\mathscr{L}(GG)$ ensures that for each graph $G \in \mathscr{L}(C)$ which satisfies constraints C it is true that its sub-graphs are in $\mathscr{L}(GG)$ and therefore can be created via rule applications of grammar GG, i.e., the language restrictions that are induced by grammar GG are reflected by constraints C. In more detail, we iterate over all minimal graphs (atoms) of type graph TG, when needed extend them via constraints C, and show that they can be created via rule applications of grammar GG. Def. 3.5 defines the step-wise extension of a graph via constraints C. Only atoms and sub-graphs that occur in graphs $G \in \mathscr{L}(C)$, called effective atoms (cf. Def. 3.7) and significant graphs (cf. Def. 3.4), are considered in C-extensions and C-extension completeness. Note that C-extension completeness only ensures that the sub-graphs of G are in $\mathscr{L}(GG)$ but not G as a whole, as, sub-graphs may overlap. Therefore, in addition to C-extensions completeness a second condition - the C-conflict-freeness of markings rules - is necessary in order to ensure that each $G \in \mathscr{L}(C)$ is in $\mathscr{L}(GG)$ (cf. Def. 3.12 and Thm. 3.3).

General Assumption Additionally to the general assumptions of the sections from before, the following general assumptions are made. As marking rules rely on attributions with translation attributes (cf. Sec. 2.3.2 and Rem. 2.17), we assume finitary $\mathscr{M}$-adhesive category $(\mathbf{AGraphs}_{ATGI,fin}, \mathscr{M}_{fin})$ with a distinct type graph $ATGI$ for all results (in particular, the category is $\mathscr{M}$-adhesive, has extremal $\mathscr{E}$-$\mathscr{M}$-factorisations, initial pushouts (cf. Sec. 2.2.2 and Rem. 2.6) and effective pushouts (cf. Sec. 2.2.2 and Rem. 2.8)). For domain completeness $\mathscr{L}(C) \subseteq \mathscr{L}(GG = (S,P))$ we assume that $\mathscr{L}(C)$ is additionally restricted to attributed graphs with algebras that are isomorphic to the algebra of start graph S, since, otherwise the language inclusion may never hold, as, rules P are spans of $\mathscr{M}$-morphisms that are preserved along pushouts in transformations, i.e., by Sec. 2.2.2 and Rem. 2.3, $\mathscr{L}(GG)$ only contains attributed graphs with algebras that are isomorphic to the algebra of S. However in general, $\mathscr{L}(C)$ is not restricted to specific algebras, e.g., for the definition of significant graphs and effective atoms in Defs. 3.4 and 3.7. Moreover, all constraints $c \in C$ are interpreted via their AC-schemata, i.e., we write $G \models c, G \models C, G \overset{I}{\models} c, G \overset{I}{\models} C$, or $\mathscr{L}(C)$ but mean those graphs that initially (generally) satisfy the AC-schemata of constraint c or constraints C (cf. Sec. 2.2.3 and Def. 2.19). Furthermore, we assume that the constraints C and rules P are defined over the same $DSIG$-term algebra $T_{DSIG}(X)$ and common set of variables X where only variables X are used as values for attributes in graphs of constraints and rules.

Note that for C-extension completeness we are only interested in significant sub-graphs that occur in graphs of $\mathscr{L}(C)$. However, the non-significance of sub-graphs G may not be directly inferable from a single constraint in C but indirectly via a chain of reasoning over several constraints of C, e.g., some constraint $c \in C$ claims for G the existence of some additional node but which is forbidden by some other constraint $c' \in C$ leading to the non-significance of G. Therefore, we introduce the notion of C-inconsistent graphs that are only a subset of not significant graphs but that can be identified more efficiently, directly based on single constraints. A graph G is C-inconsistent if G does not satisfy a constraint $c \in C$ which is violation stable under embedding. A constraint c is violation stable under embedding if for each graph G that does not satisfy c also any bigger context H around G does not satisfy c. Therefore, a C-inconsistent sub-graph does not occur in graphs $G \in \mathscr{L}(C)$ and therefore is not significant (cf. Rem. 3.2).

Definition 3.3 (Violation Stability of Constraints) A constraint c is *violation stable under embedding*, if for any graph G with $G \not\models c$ it holds that for any inclusion $i\colon G \hookrightarrow H, i \in \mathscr{M}$ also $H \not\models c$. △

Example 3.2 (Violation Stability of Constraints) *All constraints that inevitably forbid graph patterns via negations $\neg$ and all constraints that inevitably restrict types along an inheritance relation are violation stable under embedding. This conforms to constraints* $2,3,5-7,9,10,12-14$ *in Fig. 2.6 for forbidden patterns and to the constraints for abstract types in Sec. 2.2.3 and Ex. 2.3 for type restrictions.* △

Remark 3.1 (Violation Stability of Constraints & Initial Satisfaction) *Note that violation stability is only defined in terms of general satisfaction of constraints in Def. 3.3 (cf. Sec. 2.2.3 and Def. 2.15). This is due to the fact that the existential character of initial satisfaction of constraints ac_P usually allows to extend graphs G with $G \overset{I}{\not\models} ac_P$ by premise P to graphs H with inclusion $G \to H \in \mathscr{M}$ such that $H \overset{I}{\models} ac_P$, i.e., constraints that are designated for initial satisfaction usually are not violation stable under embedding. This is not the case for constraints that are contradictory and constraints of the form $\neg\exists(\varnothing \to C, ac_C)$ over initial object $\varnothing$. However, constraints of that form can be semantically equivalently interpreted via general satisfaction by the uniqueness of the initial morphism from $\varnothing$ (cf. Sec. 2.2.3 and Rem. 2.10).* △

Definition 3.4 (C-Inconsistent & Significant Graph) Let C be a set of constraints and let $C' \subseteq C$ be the contained constraints that are designated for general satisfaction and violation stable under embedding. A *graph G is significant w.r.t. $\mathscr{L}(C)$* if there is an inclusion $G \hookrightarrow H \in \mathscr{M}$ with

$H \in \mathscr{L}(C)$. A *graph G is C-inconsistent*, if $G \not\models C'$. △

Remark 3.2 (Relationship between C-inconsistent & Significant Graphs) *By definition, each C-inconsistent graph is not significant w.r.t. $\mathscr{L}(C)$. Furthermore, each graph that is significant w.r.t. $\mathscr{L}(C)$ is not C-inconsistent. The other directions do not hold in general.* △

Example 3.3 (C-Inconsistent & Significant Graph) *Given the constraints C for UML class diagrams from Sec. 2.2.3 and Ex. 2.3. Graph G_1 in Fig. 3.1 is C-inconsistent by violating the constraint for abstract type* Classifier *(cf. Ex. 3.2). The graph in Fig. 3.4 is not C-inconsistent, since, it does not violate a constraint $c \in C$ which is violation stable under embedding. However, the graph is not significant w.r.t. $\mathscr{L}(C)$, since, it cannot be embedded into a graph that satisfies constraints C. Node* 2 : Attr *has a* Const *modifier and therefore,* 2 : Attr *must also be of* type DataType *by following constraint* 15 *but simultaneously constraint* 9 *forbids that* 2 : Attr *has more than one type.* △

For a given set of constraints C, let G be a graph that is significant w.r.t. $\mathscr{L}(C)$ but that does not generally satisfy C, then the idea of extending G via C is to obtain significant graphs that generally satisfy C in order to increase the accuracy of verifying domain completeness in Thm. 3.3 via C-extension completeness in Def. 3.8. Therefore, the extension of G via C may lead to graphs, called C-extensions of G, that may increasingly generally satisfy constraints C. As the constraints are interpreted via their AC-schemata, instances of constraints C and $\mathscr{M}$-matches are used to form the extensions (cf. Sec. 2.2.3, Rem. 2.10, and Def. 2.19). The extension of G via C is defined recursively starting with the initial extension that contains graph G only. A new extension is derived from an existing extension E as follows: *a*) Let G_E be a graph of E, let c be an instance of a constraint in C without negations that may have one or more conclusions connected by disjunctions and let $m\colon P \to G_E \in \mathscr{M}$ be a match from the premise P of c to G_E. *b*) Compute all overlappings of the conclusions of c with G_E with respect to m. *c*) For each overlapping, a new graph G'_E is potentially added to E while removing G_E from E leading to extension E' via extension step $E \xRightarrow{extend(G_E,c,m)} E'$. Graph G'_E is obtained by adding the non-overlapping part of the conclusion to G_E, respectively.

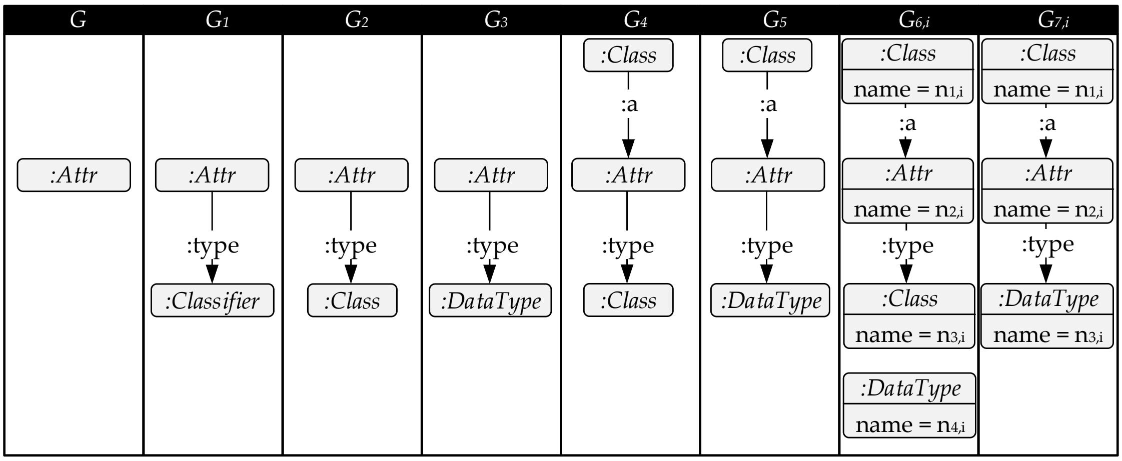
$$\begin{tikzcd}
P \arrow[r, "p"] \arrow[rr, bend left, "f"] \arrow[dr, "m"'] & P' \arrow[r, "f'"] \arrow[d, "m'"'] \arrow[dr, phantom, "(1)"] & C \arrow[d, "e'"] \\
& G_E \arrow[r, "e"] & G'_E
\end{tikzcd}$$

Definition 3.5 (C-Extensions) Let G be a graph. The *extensions of G via morphism f and match m* form the set of graphs given by $extend(G,f,m)$ below. The *extensions of G via a constraint ac_P and a match m* form the set of graphs given by $extend(G,ac_P,m)$ below. The *extensions of G via a set of constraints C* form the set of sets of graphs given by the least fixed point of $Extensions(G,C)$ below with induced morphisms e.

- $extend(G_E,f,m) = \{(e',G'_E) \mid (1)$ above is a pushout with all morphisms being in $\mathscr{M}$, $m' \circ p = m$, $f' \circ p = f$, and G'_E is significant w.r.t. $\mathscr{L}(C)$ (or not C-inconsistent)$\}$

- $extend(G_E,ac_P,m) = \begin{cases} \bigcup_{i\in I}(\bigcup_{(e',G'_E)\in E}(extend(G'_E,ac_{C_i},e'))) & \text{, if } \textbf{Cond} \\ \{G_E\} & \text{, otherwise} \end{cases}$

  with **Cond** is $E = extend(G_E,a_i,m), ac_P \equiv \vee_{i\in I}\exists(a_i\colon P \to C_i, ac_{C_i})$, and $m\colon P \to G_E \in \mathscr{M}$.



$$Extensions(G,C) = \{\{G\},\{G_2,G_3\},\{G_4,G_3\},\{G_4,G_5\},\{G_{6,1},\ldots,G_{6,8},G_{7,1},\ldots,G_{7,5}\},\ldots\}$$

Figure 3.1: C-extensions of Effective Atom : Attr

- $Extensions(G,C) = \{\{G\}\} \cup \{E' \mid E' = E \setminus \{G_E\} \cup extend(G_E, ac_P, m), E \in Extensions(G,C), G_E \in E, (\_ \in \mathcal{O}, ac_P) \in \text{Inst}(C), m\colon P \to G_E \in \mathcal{M}\}$ △

In practice, C-extensions are considered only up to isomorphism.

Remark 3.3 (C-Extensions) *Note that C-extensions are only defined for constraints of the form $\vee_{i \in I} \exists (a_i\colon P \to C_i, ac_{C_i})$ for all subconditions. Furthermore, in $extend(G_E, f, m)$ graph G'_E need to be significant or not C-inconsistent. Claiming that G'_E is significant is more accurate for verifying domain completeness based on C-extension completeness in Def. 3.8 and Thm. 3.3, since, graphs may be not C-inconsistent and not significant at the same time (cf. Ex. 3.3). While not C-inconsistent graphs are considered in C-extensions, not significant graphs are not considered. However, claiming that G'_E is not C-inconsistent can be checked more efficiently.* △

Example 3.4 (C-Extensions) *Fig. 3.1 depicts some extensions $Extensions(G,C)$ of graph G via domain constraints C for UML class diagrams of Sec. 2.2.3 and Fig. 2.6. The extensions are obtained by the following extension steps where we write the actual constraints but mean their instances according to Def. 3.5:*

- $\{G\} \xRightarrow{extend(G,8,\_)} \{G_2, G_3\}$*: Note that G_1 is also considered in C-extensions but is not contained in the actual extension, since, G_1 is C-inconsistent (cf. Ex. 3.3).*
- $\{G_2,G_3\} \xRightarrow{extend(G_2,1,\_)} \{G_4,G_3\} \xRightarrow{extend(G_3,1,\_)} \{G_4,G_5\} \xRightarrow{extend(G_5,11,\_)^*} \{G_4,G_{7,1},\ldots,G_{7,5}\} \xRightarrow{extend(G_4,16,\_)} \circ \xRightarrow{extend(\_,11,\_)^*} \{G_{6,1},\ldots,G_{6,8},G_{7,1},\ldots,G_{7,5}\}$*: Graphs $G_{6,1}$ to $G_{6,8}$ and $G_{7,1}$ to $G_{7,5}$ contain all combinations of equal and unequal attribute values $n_{1,i}$ to $n_{4,i}$ except equal values for $n_{1,i}$ and $n_{3,i}$ in graphs $G_{6,i}$. This is due to the fact that several* Classes *of the same* name *are forbidden by constraint* 12 *and therefore, graphs containing several* Classes *of the same* name *are C-inconsistent, i.e., also not significant w.r.t. $\mathcal{L}(C)$ by Rem. 3.2, and thus, they are neglected by the construction of C-extensions in Def. 3.5.* △

Prop. 3.1 states that any extension of a C-inconsistent graph G again leads to C-inconsistent graphs only yielding an empty extension and therefore, C-inconsistent graphs can be neglected in extensions.

Proposition 3.1 (C-Inconsistency in C-Extensions) *Let G be a graph, C be a set of constraints, $E \in Extensions(G,C)$ be an extension of G and $G_E \in E$ be an extended graph. If G_E is C-inconsistent, then $extend(G_E, ac_P, m) = \varnothing$ for each constraint $ac_P \equiv \vee_{i\in I}\exists(a_i\colon P \to C_i, ac_{C_i})$ and match $m\colon P \to G_E \in \mathscr{M}$.* △

Proof. Let G_E be a C-inconsistent graph, i.e., by Def. 3.4 there exists a violation stable constraint $c \in C$ with $G_E \not\models c$. $\mathscr{M}$-morphisms are closed under pushouts, i.e., $e\colon G_E \to G'_E \in \mathscr{M}$ in Def. 3.5, since, $f' \in \mathscr{M}$ and (1) is a pushout. By Def. 3.3, it follows that also $G'_E \not\models c$, i.e., also G'_E is C-inconsistent. By Def. 3.5 it follows that $extend(G_E, ac_P, m) = \varnothing$ for each constraint $ac_P \equiv \vee_{i\in I}\exists(a_i\colon P \to C_i, ac_{C_i})$ and match $m\colon P \to G_E \in \mathscr{M}$. □

Remark 3.4 *By Def. 3.5, all graphs in $Extensions(G,C)$ are not C-inconsistent, except graph G itself may be C-inconsistent as element of its initial extension $\{G\}$. By following Prop. 3.1, extending the initial extension for C-inconsistent graph G via $extend(G,\_,\_)$ yields an empty extension $\varnothing \in Extensions(G,C)$. Furthermore, contradictions in C may also lead empty extensions $\varnothing \in Extensions(G,C)$, e.g., G is extended via a constraint of C leading to C-inconsistent graphs only that do not satisfy violation stable constraints of C. In both cases, the existing empty extension $\varnothing$ of G indicates that G is not significant w.r.t. $\mathscr{L}(C)$.* △

For C-extension completeness it is sufficient to consider only the smallest graphs that occur in graphs $G \in \mathscr{L}(C)$, namely effective atoms, and from which more complex graphs can be constructed. Atoms are the smallest graphs in the sense that they cannot be splitted into smaller sub-graphs. With $Atoms(ATG)$ we denote the set of atoms that are typed over an attributed type graph ATG. For (typed) attributed graphs the structure of each atom is given by either *a*) an empty graph, or *b*) a single node, or *c*) a single edge together with source and target nodes, or *d*) a single node attribute together with the corresponding node, or *e*) a single edge attribute together with the corresponding edge and its source and target nodes. The idea of an atom a is similar to the idea of an incremental monomorphism $f\colon I \rightarrowtail a$ that must exist with I being the initial object (cf. [CHH+12]). Note that all atoms in $Atoms(ATG)$ share the same $DSIG$-term algebra $T_{DSIG}(X)$ such that the verification of domain completeness via Thm. 3.3 is performed on the topmost level of $T_{DSIG}(X)$ and therefore, can be instantiated to any concrete $DSIG$-algebra for attributed graphs in $\mathscr{L}(C)$ and $\mathscr{L}(GG)$ (Note that according to the general assumption, for domain completeness we assume that graphs in $\mathscr{L}(C)$ and $\mathscr{L}(GG)$ share the same concrete $DSIG$-algebra up to isomorphism).

Definition 3.6 (Atom) A *graph a is an atom*, if for each pushout (1) on the right with morphisms $b, c \in \mathscr{M}$ it is true that $b'\colon B \to a$ is an isomorphism or $c'\colon C \to a$ is an isomorphism. Let $ATG = (TG, Z)$ be an attributed type graph with type graph TG and the final $DSIG$-algebra Z of data signature $DSIG = (S, OP)$ with sorts S and operations OP. With $Atoms(ATG) = \{(a = (G, T_{DSIG}(X)), type_G\colon a \to ATG) \mid a \text{ is an atom}, (\text{for all } e \in E_j^G\colon t_j^G(e) \in X)_{j\in\{NA,EA\}}, X = (X_s)_{s\in S} \text{ being a family of infinite sets } X_s \text{ of variables for each sort } s \in S\}$ we define the set of atoms by attributed graphs that are typed over ATG and that share the same $DSIG$-term algebra $T_{DSIG}(X)$ with an infinite set of variables X_s for each sort $s \in S$ where each node attribute E_{NA}^G and each edge attribute E_{EA}^G has a variable as attribute value. △

$$\begin{tikzcd}
 & B \arrow[dr, "b'"] & \\
I \arrow[ur, "b"] \arrow[dr, "c"'] & (1) & a \\
 & C \arrow[ur, "c'"'] &
\end{tikzcd}$$

Given a set of constraints C that are typed over attributed type graph ATG, then with $EAtoms(C)$ we denote the set of effective atoms w.r.t. language $\mathscr{L}(C)$. Effective atoms are those atoms in $Atoms(ATG)$ that occur in graphs $G \in \mathscr{L}(C)$.

Definition 3.7 (Effective Atom) Given language $\mathscr{L}(C)$ over attributed type graph ATG and constraints C, an *atom* $a \in Atoms(ATG)$ *is effective w.r.t.* $\mathscr{L}(C)$, if there exists an inclusion $i\colon a \to G \in \mathscr{M}$ for some graph $G \in \mathscr{L}(C)$. With $EAtoms(C) = \{a \mid a \in Atoms(ATG), a$ is effective w.r.t. $\mathscr{L}(C)\}$ we denote the set of effective atoms w.r.t. $\mathscr{L}(C)$. △

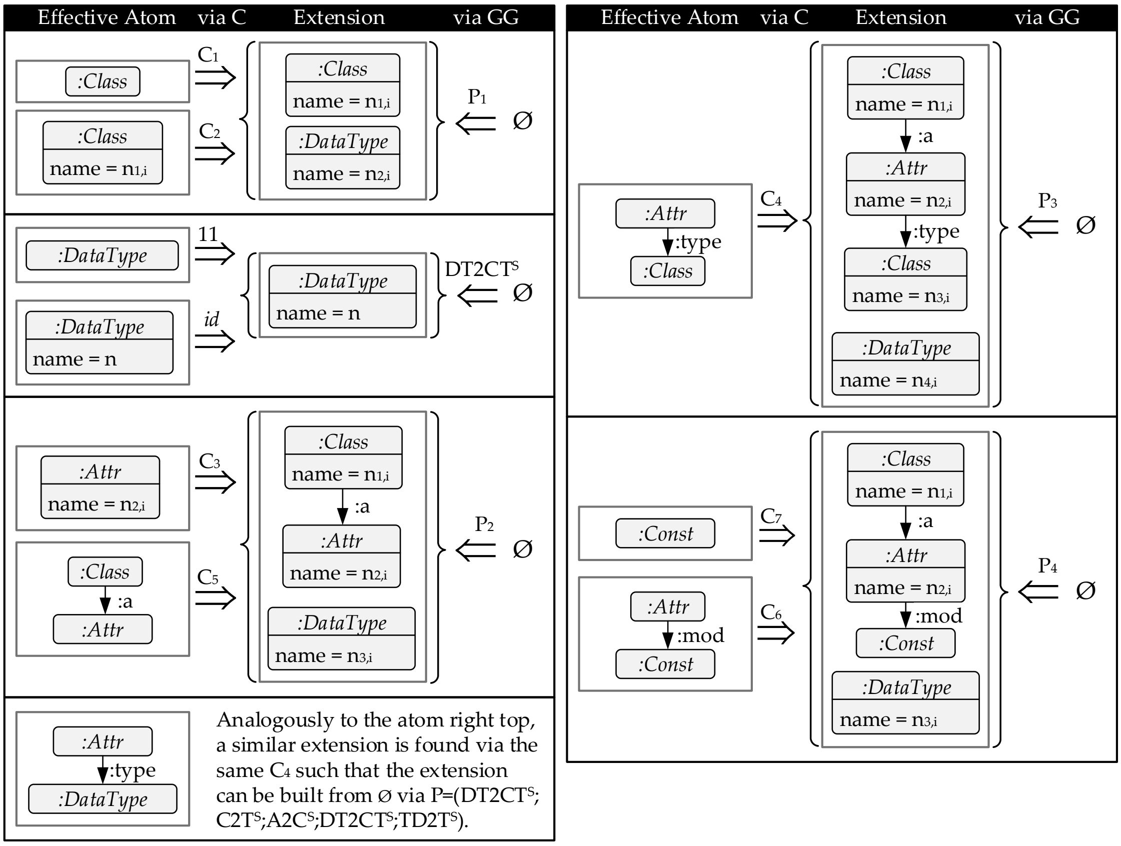
Example 3.5 (Effective Atom) *Given the constraints C for UML class diagrams in Sec. 2.2.3 and Ex. 2.3, then the effective atoms w.r.t. $\mathscr{L}(C)$ are those atoms in $Atoms(TG_{CD})$ that fulfill the domain constraints for abstract types from Ex. 2.3. Graphs G, G_2 and G_3 in Fig. 3.1 are effective atoms w.r.t. $\mathscr{L}(C)$. Graph G_1 is an atom but not effective, since,* Classifier *is an abstract type and therefore, G_1 violates the constraints for abstract types. Graphs G_4 to $G_{7,5}$ are not atoms.* △

For C-extension completeness, in practice we consider atoms up to isomorphism only. Given a set of constraints C that are typed over type graph ATG, then C-extension completeness of a given language $\mathscr{L}$ over ATG states that for all effective atoms w.r.t. $\mathscr{L}(C)$ that are typed over ATG, an extension via constraints C can be found that is in $\mathscr{L}$.

Definition 3.8 (C-Extension Completeness) Let C be a set of constraints typed over ATG and $C' \subseteq C$ be the contained constraints that are designated for general satisfaction. Then, a language $\mathscr{L}$ over ATG is called *C-extension complete*, if $\forall a \in EAtoms(C).\exists S \in Extensions(a, C')$ such that $S \subseteq \mathscr{L}$. △

Example 3.6 (C-Extension Completeness) *Given the rules P for creating UML class diagrams from Sec. 2.2.4 and Ex. 2.4 together with grammar $GG = (\varnothing, P)$ that is typed over TG_{CD} in Sec. 2.2.1 and Fig. 2.4 and the constraints C for class diagrams typed over TG_{CD} from Sec. 2.2.3 and Ex. 2.3. We show C-extension completeness of language $\mathscr{L}(GG)$ over TG_{CD}. For each effective atom in $EAtoms(C)$ an extension via constraints C must be constructed that is a subset of language $\mathscr{L}(GG)$. Similarly to Ex. 3.4, we write the actual constraints but mean their instances according to Def. 3.5 and furthermore, each graph with attribute values $n_{1,i}$ to $n_{4,i}$ in Fig. 3.2 represents a set of graphs that contains all combinations of equal and unequal values $n_{1,i}$ to $n_{4,i}$ except equal values for $n_{1,i}$ and $n_{3,i}$ if both represent the* names *of distinct* Classes. *An extension is a subset of $\mathscr{L}(GG)$, if all graphs of the extension can be constructed via rules P from start graph $\varnothing$ of GG. The effective atoms $EAtoms(C)$ are given by graph G in Fig. 3.1 together with the effective atoms in Fig. 3.2. For atom G, extension $\{G_{6,1}, \ldots, G_{6,8}, G_{7,1}, \ldots, G_{7,5}\} \in Extensions(G, C)$ can be constructed by extending G via constraints $(8;1;1;11;11;11;16;11;11;11;11)$ of C successively (cf. Ex. 3.4). Each graph of the extension can be constructed by applying rules $(\mathsf{DT2CT^S};\mathsf{C2T^S};\mathsf{A2C^S};\mathsf{C2T^S};\mathsf{TC2T^S})$ or $(\mathsf{DT2CT^S};\mathsf{C2T^S};\mathsf{A2C^S};\mathsf{TD2T^S})$ successively with starting at the empty start graph $\varnothing$ of GG. All other effective atoms are also successfully checked as depicted in Fig. 3.2. Therefore, language $\mathscr{L}(GG)$ is C-extension complete.* △

In addition to C-extension completeness another property called C-conflict-freeness of marking rules is neccessary in order to verify full language inclusions $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$ of domain completeness. Based on the notions of translation attributes in Sec. 2.3.2 and Rem. 2.17 and consistency creating (CC) rules in Sec. 2.3.3 and Def. 2.29, we define marking rules for non-deleting flat grammars with application conditions in Def. 3.9. In the context of marking rules, we take marking attribute as synonym for translation attribute. For a non-deleting grammar GG, the set



C_1=(16;11;11), C_2=(16;11), C_3=(1;16;11;11), C_4=(1;16;11;11;11;11), C_5=(16;11;11;11), C_6=(16;1;11;11;11), C_7=(4;16;1;11;11;11)
P_1=(DT2CT^S;C2T^S), P_2=(DT2CT^S;C2T^S;A2C^S), P_3=(DT2CT^S;C2T^S;A2C^S;C2T^S;TC2T^S), P_4=(DT2CT^S;C2T^S;A2C^S;MC^S)

Figure 3.2: Verifying C-Extension Completeness of Language $\mathscr{L}(GG)$ for UML Class Diagrams

of marking rules $m(GG)$ contains for each rule $r \in GG$, a marking rule $m(r)$. Analogously to CC rules, marking rules are derived from rules by adding marking attributes with value **F** (false) or **T** (true) to all elements (nodes, edges or attributes) of the rule. Whenever rule r creates an element, then marking rule $m(r)$ preserves this element and updates its marking attribute from **F** to **T**, denoted by [**F** => **T**]. Whenever rule r preserves an element, then marking rule $m(r)$ also preserves this element and leaves its marking attribute set to **T**. Thus, marking rules are deleting on the marking attributes and allow a conflict analysis of (common) created elements which cannot be performed directly on the non-deleting rules themselves. Furthermore, all elements of the application condition of rule r that are not contained in the left-hand side of the rule need to be extended by a marking attribute with value **T**. Therefore, we use the concept of **T**-extended application conditions from Sec. 2.3.2 and Def. 2.25 and restrict it from triple graphs to flat graphs by omitting the triple components X.

Definition 3.9 (Marking Rule) Given a non-deleting rule $p = (\overline{p}\colon L \hookrightarrow R, ac_L)$, the *marking rule* $m(p) = (L_M \overset{l_M}{\hookleftarrow} K_M \overset{r_M}{\hookrightarrow} R_M, ac_{L_M})$ *of p* is constructed component-wise for L_M, K_M, R_M and ac_{L_M} with induced inclusions l_M, r_M and $L_M := R \oplus Att^{\mathbf{T}}_{\overline{p}(L)} \oplus Att^{\mathbf{F}}_{R\setminus\overline{p}(L)}, K_M := R \oplus Att^{\mathbf{T}}_{\overline{p}(L)}, R_M := R \oplus Att^{\mathbf{T}}_{R}$ and $ac_{L_M} = tExt(ac_L, L_M)$. Let $GG = (S, P)$ be a non-deleting graph grammar. With $m(GG) = \{m(p) \mid p \in P\}$ we define the set of marking rules for all rules in GG. △

Remark 3.5 (Termination of Transformation System $m(GG)$) *Although marking rules are not directly applied for verifying domain completeness, we state the following interesting property:*

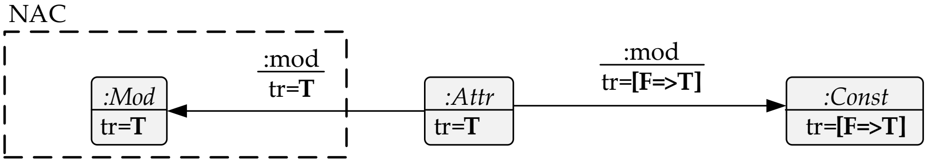


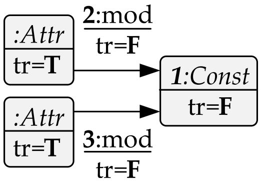
Figure 3.3: Marking Rule $m(4^S)$ of Rule 4^S for UML Class Diagrams

If all rules $p = (L \hookrightarrow R, ac_L)$ of a non-deleting grammar GG are non-trivial in the sense that L and R are not isomorphic ($L \ncong R$) and therefore, each rule creates at least one element, then the transformation system $m(GG)$ of marking rules is terminating for finite graphs, since, each application of marking rules updates the marking attributes of at least one graph element from **F** *to* **T** *as long as all graph elements are marked with* **T** *or no marking rule is applicable anymore.* △

Example 3.7 (Marking Rule) *The marking rule $m(4^S)$ of source rule 4^S from Sec. 2.2.4 and Ex. 2.4 is given by the rule in Fig. 3.3. Marking attributes tr are added to all elements and they are updated from* **F** *to* **T** *(*[**F** => **T**]*) for those elements that are created by rule 4^S while the marking attributes of all other elements are initially set to* **T** *and remain unchanged.* △

The C-conflict-freeness of marking rules is checked based on critical pair analysis. According to Def. 3.10 and similarly to C-inconsistent graphs, a critical pair $(K_1 \Leftarrow O \Rightarrow K_2)$ is C-inconsistent if graph O does not satisfy a constraint in C that is violation stable under embedding. C-inconsistent critical pairs do not need to be analysed for verifying domain completeness, since, such critical pairs and any of their embeddings into larger contexts do not occur in graphs $G \in \mathscr{L}(C)$. The marking rules are C-conflict-free, if for each critical pair $(K_1 \xLeftarrow{(p_1,o_1)} O \xRightarrow{(p_2,o_2)} K_2)$ that is not C-inconsistent with marking rules p_1 and p_2, the rules and matches are the same $(p_1 = p_2, o_1 = o_2)$ (cf. Def. 3.12).

Definition 3.10 (C-Inconsistent Critical Pair) Let C be a set of constraints. A *critical pair $(K_1 \Leftarrow O \Rightarrow K_2)$ is C-inconsistent*, if conflict graph O is C-inconsistent. △



Example 3.8 (C-Inconsistent Critical Pair) *The figure on the right presents the conflict graph O of the critical pair $(K_1 \xLeftarrow{(m(4^S),m_1)} O \xRightarrow{(m(4^S),m_2)} K_2)$ with marking rule $m(4^S)$ from Ex. 3.7. Transformation $O \xRightarrow{(m(4^S),m_1)} K_1$ updates the marking attributes of node* 1 : Const *and edge* 2 : mod *and $O \xRightarrow{(m(4^S),m_2)} K_2$ updates the marking attributes of* 1 : Const *and edge* 3 : mod *from* **F** *to* **T**. *This corresponds to a delete-use conflict at node* 1 : Const *in O, since, both transformations change its marking attribute where the attribute is deleted at first and added with the new value* **T** *afterwards. However, by assuming the constraints C for UML class diagrams from Sec. 2.2.3 and Ex. 2.3, the critical pair is C-inconsistent, since, $O \not\models 7$ and constraint $7 \in C$ is violation stable under embedding, i.e., graph O is C-inconsistent (cf. Ex. 3.2).* △

Analogously, to significant graphs, we introduce significant critical pairs. A critical pair is significant, if it occurs in graphs $G \in \mathscr{L}(C)$. Also analogously to significant and C-inconsistent graphs, C-inconsistent critical pairs are only a subset of not significant critical pairs but can be identified more efficiently.

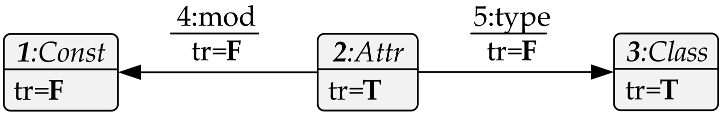
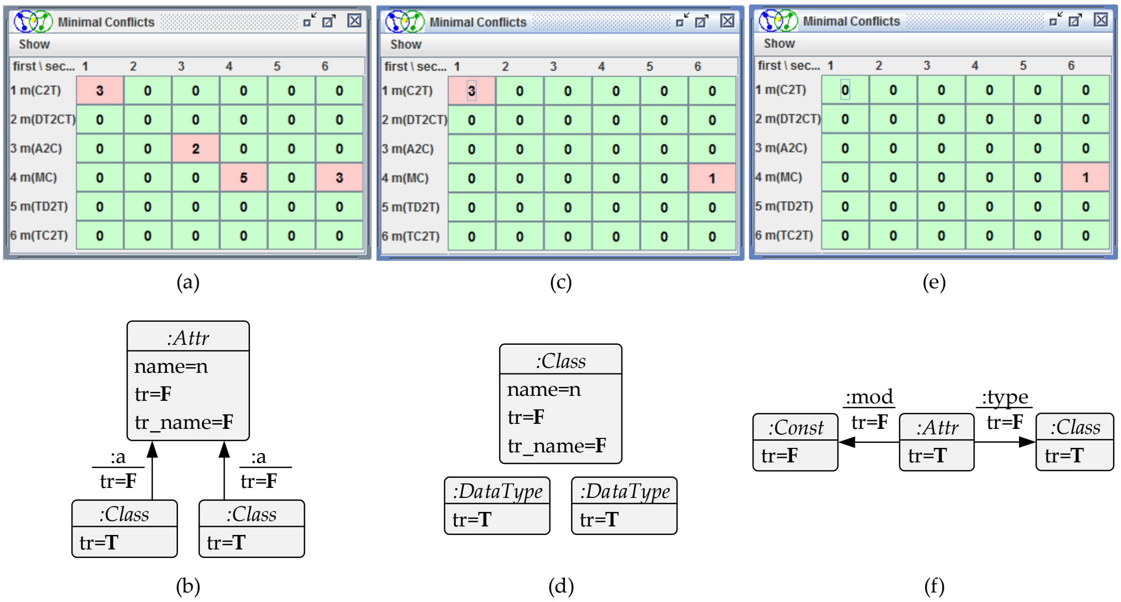


Figure 3.4: Significant Critical Pair



first \ sec...	1	2	3	4	5	6
1 m(C2T)	3	0	0	0	0	0
2 m(DT2CT)	0	0	0	0	0	0
3 m(A2C)	0	0	2	0	0	0
4 m(MC)	0	0	0	5	0	3
5 m(TD2T)	0	0	0	0	0	0
6 m(TC2T)	0	0	0	0	0	0

(a)

first \ sec...	1	2	3	4	5	6
1 m(C2T)	3	0	0	0	0	0
2 m(DT2CT)	0	0	0	0	0	0
3 m(A2C)	0	0	0	0	0	0
4 m(MC)	0	0	0	0	0	1
5 m(TD2T)	0	0	0	0	0	0
6 m(TC2T)	0	0	0	0	0	0

(c)

first \ sec...	1	2	3	4	5	6
1 m(C2T)	0	0	0	0	0	0
2 m(DT2CT)	0	0	0	0	0	0
3 m(A2C)	0	0	0	0	0	0
4 m(MC)	0	0	0	0	0	1
5 m(TD2T)	0	0	0	0	0	0
6 m(TC2T)	0	0	0	0	0	0

(e)

(b)

(d)

(f)

Figure 3.5: Result of Conflict Analysis for Marking Rules of UML Class Diagrams in AGG

Definition 3.11 (Significant Critical Pair) Let C be a set of constraints, GG be a non-deleting grammar and $m(GG)$ be the corresponding set of marking rules. A *critical pair* $(K_1 \xLeftarrow{(p_1,m_1)} O \xRightarrow{(p_2,m_2)} K_2)$ *for marking rules* $p_1, p_2 \in m(GG)$ *is significant w.r.t.* $\mathscr{L}(C)$, if conflict graph O is significant w.r.t. $\mathscr{L}(C)$. △

Remark 3.6 (Relationship between C-Inconsistent & Significant Critical Pairs) *Similarly to the relationship between C-inconsistent and significant graphs in Rem. 3.2, by definition, each C-inconsistent critical pair is not significant w.r.t. $\mathscr{L}(C)$. Furthermore, each critical pair that is significant w.r.t. $\mathscr{L}(C)$ is not C-inconsistent. The other directions do not hold in general.* △

Example 3.9 (Significant Critical Pair) *The graph in Fig. 3.4 presents the conflict graph O of the critical pair $(K_1 \xLeftarrow{(m(4^S),m_1)} O \xRightarrow{(m(6^S),m_2)} K_2)$ with marking rules $m(4^S)$ and $m(6^S)$ of the rules in Sec. 2.2.4 and Ex. 2.4. Transformation $O \xRightarrow{(m(4^S),m_1)} K_1$ updates the marking attributes of edge* 4 : mod *and node* 1 : Const *from* **F** *to* **T** *leading to a graph pattern which is forbidden by the NAC of rule $m(6^S)$ with match m_2 (change-forbid attribute conflict). Given the constraints C for UML class diagrams from Sec. 2.2.3 and Ex. 2.3, then the critical pair is not C-inconsistent, since, O does not violate a constraint $c \in C$ which is violation stable under embedding. However, the critical pair is not significant, since, graph O is not significant w.r.t. $\mathscr{L}(C)$ (cf. Ex. 3.3).* △

Definition 3.12 (C-Conflict-Freeness of Marking Rules) Let C be a set of constraints and let $m(GG)$ be the marking rules of a non-deleting grammar GG. Then, $m(GG)$ *is C-conflict-free*, if for each critical pair $(K_1 \xleftarrow{(p_1,o_1)} O \xrightarrow{(p_2,o_2)} K_2)$ that is significant w.r.t. $\mathscr{L}(C)$ (or not C-inconsistent) with $p_1, p_2 \in m(GG)$ it is true that the rules and matches are the same $(p_1 = p_2, o_1 = o_2)$ (or it is true that the critical pair is strictly confluent). $\triangle$

Remark 3.7 (C-Conflict-Freeness of Marking Rules) *The critical pairs in Def. 3.12 need to be significant or not C-inconsistent. Analogously to Rem. 3.3, claiming that the critical pairs are significant is more accurate for verifying domain completeness based on C-conflict-freeness of marking rules in Def. 3.12 and Thm. 3.3, since, critical pairs may be not C-inconsistent and not significant at the same time (cf. Ex. 3.9). While not C-inconsistent critical pairs are considered in C-conflict-freeness of marking rules, not significant critical pairs are not considered. However, claiming that the critical pairs are not C-inconsistent can be checked more efficiently. Furthermore, claiming that each critical pair is strictly confluent is a stronger condition and harder to check than claiming that each critical pair is of same rules and same matches. Each critical pair of same rules and same matches is directly strict confluent by definition.* $\triangle$

Example 3.10 (C-Conflict-Freeness of Marking Rules) *The AGG-tool [AGG16] is used to help verifying C-conflict-freeness of marking rules $m(GG)$ for grammar $GG = (S,P)$ with rules P for UML class diagrams from Sec. 2.2.4 and Ex. 2.4 and constraints C for UML class diagrams from Sec. 2.2.3 and Ex. 2.3. AGG enables to analise conflicts of rules, outputs all existing critical pairs and allows to ignore 1. critical pairs of same rules and same matches, 2. critical pairs that violate multiplicity constraints, and 3. critical pairs that are directly strict confluent. Fig. 3.5 depicts the result of the analysis for the marking rules $m(GG)$ of UML class diagrams. Fig. 3.5 (a) depicts all 13 critical pairs of marking rules $m(GG)$ (red boxes) while ignoring all critical pairs of same rules and same matches. Fig. 3.5 (b) depicts the conflict graph of the critical pair of rule 3^S that updates the marking attributes of node* : Attr *and edges* : a *in parallel while matching differently to both* : Class *nodes. The conflict graph violates multiplicity constraint* 2 *which is violation stable under embedding (cf. Ex. 3.2). Therefore the conflict graph is C-inconsistent implying further that the critical pair is C-inconsistent. Fig. 3.5 (c) depicts all four critical pairs of marking rules $m(GG)$ while additionally ignoring all critical pairs that violate the multiplicity constraints in C. Fig. 3.5 (d) depicts the conflict graph of the critical pair of rule 1^S that updates the marking attributes of node* : Class *in parallel while matching differently to both* : DataType *nodes. The conflict graph is not C-inconsistent, particularly it respects the multiplicity constraints. However, the conflict graph is directly strict confluent, i.e., the critical pair is directly strict confluent. Fig. 3.5 (e) depicts the only critical pair of marking rules $m(GG)$ while additionally ignoring all critical pairs that are directly strict confluent. Fig. 3.5 (f) depicts the conflict graph of the critical pair of rules 4^S and 6^S that updates the marking attributes of node* : Const *and edges* : mod, : type *in parallel while matching node* : Attr *in common. The critical pair is not strictly confluent due to the NAC of rule 6^S. Furthermore, the critical pair is not C-inconsistent, particularly it does not violate the multiplicity constraints in C. However, the critical pair is not significant (cf. Ex. 3.9). Therefore, the marking rules $m(GG)$ of UML class diagrams are C-conflict-free.* $\triangle$

The main result for verifying domain completeness is stated by Thm. 3.3. Intuitively, the language inclusion $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$ holds if each graph $G \in \mathscr{L}(C)$ can be decomposed into its atoms $a \subseteq G$ such that for each atom a an extension E can be constructed via constraints C that is contained in $\mathscr{L}(GG)$ and the composition of the extensions leads to graph G in $\mathscr{L}(GG)$ again by applying the rules of grammar GG. The verification approach requires that all productions of grammar GG are non-trivial. For non-trivial productions we refer to Rem. 3.5.

Theorem 3.3 (Domain Completeness) *Let $\mathscr{L}(C)$ be a language over type graph ATG and* *constraints C in $\mathcal{M}$-normal form and let $\mathcal{L}(GG)$ be a language over ATG and non-deleting grammar $GG = (\varnothing, P)$ with empty start graph $\varnothing$, all productions $p \in P$ being non-trivial and where all application conditions in productions P are in $\mathcal{M}$-normal form. If the marking rules $m(GG')$ are C-conflict-free and $\mathcal{L}(GG')$ is C-extension complete where $GG' = (\varnothing', P)$ with $\varnothing'$ being the empty start graph with DSIG-term algebra $T_{DSIG}(X)$, then domain completeness holds for almost injective matches $m \in \mathcal{O}$, i.e., it holds that $\mathcal{L}(C) \subseteq \mathcal{L}(GG)$.* △

Idea. Let $G \in \mathcal{L}(C)$. Graph G is abstracted to a graph G_A with *DSIG*-term algebra. G_A can be decomposed into its atoms $Atoms(G_A)$ by Lem. 3.5. C-extension completeness of $\mathcal{L}(GG')$ ensures that each atom can be extended via C such that the extension can be created via grammar GG'. Furthermore, there is a gluing of all extensions leading to graph G_A again by Lem. 3.7. By the equivalence of marking and transformation sequences in Lem. 3.2, each extension can be fully marked with **T**. The C-conflict freeness of the marking rules $m(GG')$ allows to apply confluence results and we derive a marking sequence that fully marks G_A. Thus by Lem. 3.2, there is a transformation sequence $\varnothing' \overset{*}{\Rightarrow} G_A$ via P. By Lem. 3.15, there is a transformation $\varnothing \overset{*}{\Rightarrow} G$ via P, i.e., $G \in \mathcal{L}(GG)$. The full proof is presented in appendix A.11. □

We successfully verified domain completeness $\mathcal{L}(C) \subseteq \mathcal{L}(GG)$ for constraints C of UML class diagrams from Sec. 2.2.3 and Ex. 2.3 and grammar $GG = (\varnothing, P)$ with rules P for UML class diagrams from Sec. 2.2.4 and Ex. 2.4. C-extension completeness of language $\mathcal{L}(GG)$ is successfully verified in Ex. 3.6. C-conflict-freeness of the marking rules $m(GG)$ is successfully verified in Ex. 3.10 by claiming that the critical pairs are significant and strictly confluent (cf. Rem. 3.7). Therefore, all graphs $G \in \mathcal{L}(C)$ can be constructed via grammar GG. This means that the domain constraints C are strict enough to cover all language restrictions that are induced by grammar GG. Note that if C-extension completeness of $\mathcal{L}(GG)$ or C-conflict-freeness of marking rules $m(GG)$ does not hold, then their verification may lead to minimal examples of graphs $G \in \mathcal{L}(C)$ that cannot be constructed via grammar GG. Such examples may serve as helpful hints for refactoring the grammar or determining which constraints need to be added to or removed from C in order to obtain domain completeness ($\mathcal{L}(C) \subseteq \mathcal{L}(GG)$).

In order to ensure termination for the verification of domain completeness via Thm. 3.3, one can define a finite upper bound G_u for the size of graphs $G \in \mathcal{L}(C)$, i.e., only graphs $G \in \mathcal{L}(C)$ with inclusions $G \to G_u \in \mathcal{M}$ are considered (cf. Def. 3.13). In most cases, the verification terminates without restricting to an upper bound as shown in Examples 3.6 and 3.10. Note that by restricting to an upper bound we could also check for all graphs up to the upper bound which satisfy the constraints in C if they can be created via the rules in grammar GG for ensuring the validity of language inclusion $\mathcal{L}(C) \subseteq \mathcal{L}(GG)$ up to the upper bound. However, the verification via C-extension completeness in Thm. 3.3 is more efficient in most cases, since, not all graphs need to be checked but rather a small subset.

Definition 3.13 (Domain Completeness up to Upper Bound) Given the context of domain completeness in Sec. 3.1 and Def. 3.2 and an object G_u as upper bound. Domain completenesss up to upper bound G_u holds if $\mathcal{L}(C)_{G_u} \subseteq \mathcal{L}(GG)$ with $\mathcal{L}(C)_{G_u} = \{G \mid G \in \mathcal{L}(C), \exists G \to G_u \in \mathcal{M}\}$. △

Analogously to the verification of domain completeness in Thm. 3.3, domain completeness up to an upper bound can be verified as follows.

Corollary 3.1 (Domain Completeness up to Upper Bound) *Given the context of verifying domain completeness in Thm. 3.3 and an object G_u as upper bound. Then, domain completeness* *up to upper bound G_u holds if both conditions from Thm. 3.3 hold but where $\mathscr{L}(C)$ is replaced by $\mathscr{L}(C)_{G_u}$ from Def. 3.13 in Defs. 3.4, 3.5, 3.7, 3.11 and 3.12.* △

Theorem 3.4 (Termination of Verification of Domain Completeness) *In the finitary category $(\mathbf{AGraphs}_{ATGI,fin}, \mathscr{M}_{fin})$, let GG be a finite non-deleting grammar with empty start graph over a finite type graph ATGI, C be a finite set of finite graph constraints over ATGI with a finite number of nestings and graph $G_u \in \mathscr{L}(ATGI)$ be a finite upper bound for the size of the graphs in $\mathscr{L}(C)$. The rules of GG are non-trivial in the sense that each rule creates at least one element (node, edge or attribute). Furthermore, all application conditions of rules in GG are finite with a finite number of nestings. Then, the verification of domain completeness up to an upper bound G_u via the conditions in Cor. 3.1 terminates.* △

Proof. The proof is presented in appendix A.4. □

For the rest of this section, we formalise the concepts of the proof idea of Thm. 3.3 in order to finally prove Thm. 3.3. We first show the equivalence of marking and transformation sequences with grammars for the case that the start graph is the empty graph in Lem. 3.2 and that the start graph is an arbitrary graph in Lem. 3.3. Therefore, each graph that can be created via the rules of a given grammar GG can also be completely marked to $\mathbf{T}$ via the marking rules $m(GG)$ of GG and vice versa.

Lemma 3.2 (Equivalence of Marking and Transformation Sequence for empty Start Graph) *Let $GG = (\varnothing, P)$ be a graph grammar with empty start graph $\varnothing$, a set P of non-deleting rules and $m(GG)$ be the set of derived marking rules of GG. Let G be a graph, then the following are equivalent for almost injective matches $m \in \mathscr{O}$.*

- *There exists a transformation $G \oplus Att^{\mathbf{F}}_{G} \overset{*}{\Rightarrow} G \oplus Att^{\mathbf{T}}_{G_k} \oplus Att^{\mathbf{F}}_{G \setminus G_k}$ via marking rules $m(GG)$.*
- *There exists a transformation $\varnothing \overset{*}{\Rightarrow} G_k$ via P with injective embedding $f \colon G_k \to G$.* △

Proof. The proof is presented in appendix A.5. □

Lemma 3.3 (Equivalence of Marking and Transformation Sequence) *Let $GG = (S, P)$ be a graph grammar with start graph S, a set P of non-deleting rules and $m(GG)$ be the set of derived marking rules of GG. Let G be a graph with inclusion $S \to G \in \mathscr{M}$, then the following are equivalent for almost injective matches $m \in \mathscr{O}$.*

- *There exists a transformation $G \oplus Att^{\mathbf{T}}_{S} \oplus Att^{\mathbf{F}}_{G \setminus S} \overset{*}{\Rightarrow} G \oplus Att^{\mathbf{T}}_{G}$ via marking rules $m(GG)$.*
- *There exists a transformation $S \overset{*}{\Rightarrow} G$ via P.* △

Proof. The proof is presented in appendix A.6. □

Def. 3.14 defines the binary split of a given graph G into two sub-graphs which is extended to the general split of G into its atoms $Atoms(G)$ in Def. 3.15 such that the gluing of the sub-graphs (atoms) yields G again by Cor. 3.2 and Lem. 3.5. For proving Lem. 3.5, we show in Lem. 3.4 that each non-atomic graph can be splitted.

Definition 3.14 (Binary Split) Let AG be a graph. The set of *binary splits $BSplits(AG)$* of AG into sub-graphs is given by co-spans of inclusions in $\mathscr{M}$: $BSplits(AG) := \{(AG_1 \overset{f'}{\hookrightarrow} AG \overset{g'}{\hookleftarrow} AG_2) \mid$ (1) is a pushout over

$$\begin{tikzcd}
 & AG & \\
AG_1 \arrow[ur, hook, "f'"] & (1) & AG_2 \arrow[ul, hook', "g'"'] \\
 & AG_0 \arrow[ul, hook, "g"] \arrow[ur, hook', "f"'] &
\end{tikzcd}$$

inclusions $(f\colon AG_0 \hookrightarrow AG_2, g\colon AG_0 \hookrightarrow AG_1)$ with $f, f', g, g' \in \mathcal{M}, AG \ncong AG_1, AG \ncong AG_2\}$ △

Corollary 3.2 (Gluing of Binary Split) *Let* $(AG_1 \overset{f'}{\hookrightarrow} AG \overset{g'}{\hookleftarrow} AG_2) \in BSplits(AG)$ *be a binary split of AG into sub-graphs* AG_1 *and* AG_2*. Then, there exist* AG_0 *and a span of* $\mathcal{M}$*-morphisms* $(AG_1 \overset{g}{\hookleftarrow} AG_0 \overset{f}{\hookrightarrow} AG_2)$ *such that (1) is a pushout.* △

Lemma 3.4 (Binary Split of Non-Atomic Graphs) *Let AG be a graph that is not an atom, then AG can be splitted into sub-graphs* AG_1 *and* AG_2 *with* $(AG_1 \to AG \leftarrow AG_2) \in BSplits(AG)$. △

Proof. We construct initial pushout (1) for id_{AG} with $g' \in \mathcal{M}$. Analogously, we construct initial pushout (2) for g' with $f, f' \in \mathcal{M}$. By Sec. 2.2.2 and Def. 2.7, (2) is also a pullback, i.e., $g \in \mathcal{M}$ by $\mathcal{M}$-morphisms g' are closed under pullbacks.

$$\begin{tikzcd}
 & & AG & \\
 & AG \arrow[ur, "id_{AG}"] & (1) & AG'' \arrow[ul] \\
AG_1 \arrow[ur, hook, "f'"] & (2) & AG_2 \arrow[ul, hook', "g'"'] \arrow[ur] & \\
 & AG_0 \arrow[ul, hook', "g"] \arrow[ur, hook, "f"'] & &
\end{tikzcd}$$

Assumption AG is not an atom, Def. 3.6 and $\mathcal{M}$-morphisms $f, g \in \mathcal{M}$ are closed under pushouts imply that there exists pushout (2) with $f, f', g, g' \in \mathcal{M}$ such that $AG \ncong AG_1$ and $AG \ncong AG_2$. Thus according to Def. 3.14, there is $(AG_1 \overset{f'}{\hookrightarrow} AG \overset{g'}{\hookleftarrow} AG_2) \in BSplits(AG)$. □

Note that a graph G may be splitted binary in different ways, possibly leading to several sets of atoms of G in $Atoms(G)$. However, for category $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ all sets are isomorphic, i.e., only one set of atoms (decision path of binary splits) in $Atoms(G)$ need to be considered.

Definition 3.15 (Atomic Split) The *set of atoms of a given graph AG* is defined by $Atoms(AG)$ as given below:

$$Atoms(AG) := \begin{cases} \{\{AG\}\} & ,AG \text{ is an atom} \\ \{A_1 \sqcup A_2 \mid (AG_1 \hookrightarrow AG \hookleftarrow AG_2) \in BSplits(AG), & ,AG \text{ is not an atom} \\ A_1 \in Atoms(AG_1), A_2 \in Atoms(AG_2)\} & \end{cases}$$ △

Lemma 3.5 (Gluing of Atomic Split) *Let G be a graph in* $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$. *Then, there are atoms* $(a_i)_{i\in\{1,\dots,n\}} \in Atoms(G)$ *and for all* $(a_i)_{i\in\{1,\dots,n\}} \in Atoms(G)$ *there exist graphs* $(G_j)_{1\leq j\leq n-1}$ *and pushouts* $(PO_k +_{G_k} a_{k+1})_{k\in\{1,\dots,n-1\}}$ *with pushout objects* PO_{k+1} *as depicted by pushout* (1) *on the right with all morphisms in* $\mathcal{M}$ *and with* $PO_1 = a_1$ *and induced morphisms* $a_1 \xrightarrow{f'_1 \in \mathcal{M}} PO_2 \xrightarrow{f'_{n-1}\circ\dots\circ f'_2 \in \mathcal{M}} PO_n \in \mathcal{M}$ *and* $a_{k+1} \xrightarrow{g'_k \in \mathcal{M}} PO_{k+1} \xrightarrow{f'_{n-1}\circ\dots\circ f'_{k+1} \in \mathcal{M}} PO_n \in \mathcal{M}$ *such that pushout object* PO_n *is G.* △

$$\begin{tikzcd}
 & PO_{k+1} & \\
PO_k \arrow[ur, hook, "f'_k"] & (1) & a_{k+1} \arrow[ul, hook', "g'_k"'] \\
 & G_k \arrow[ul, hook', "g_k"] \arrow[ur, hook, "f_k"'] &
\end{tikzcd}$$

Proof. Let G be a graph.

Case (G is an atom) By construction Def. 3.15, there is $\{G\} \in Atoms(G) = \{\{G\}\}$ with $n = 1$, i.e., for all $(a_i)_{i\in\{1,\dots,n\}} \in Atoms(G)$ the assumption holds with induced morphism id_G.

Case (G is not an atom) By Lem. 3.4, there exists a binary split $(G_1 \xrightarrow{f'} G \xleftarrow{g'} G_2) \in BSplits(G)$. By construction of binary splits in Def. 3.14 with $G \ncong G_1$, $G \ncong G_2$ and injective $\mathcal{M}$-morphisms f' and g' it follows that G_1 and G_2 are smaller than G on the graph part.

Therefore, analogously we can proceed with splitting G_1 and G_2 binary in $Atoms(G_1)$ and $Atoms(G_2)$ recursively and terminate in each decision path of binary splits when obtaining atoms. Thus, there is $(a_i)_{i\in\{1,...,n\}} \in Atoms(G)$. Let $(a_i)_{i\in\{1,...,n\}} \in Atoms(G)$ be the atoms of graph G. By induction over the number n of atoms: **Basis.** For $n = 2$, by construction there exists $(a_1 \xrightarrow{f'_1} G \xleftarrow{g'_1} a_2) \in BSplits(G)$ with a_1, a_2 being atoms, i.e., by Cor. 3.2 there exist G_1 and a span of $\mathscr{M}$-morphisms $(a_1 \xleftarrow{g_1} G_1 \xrightarrow{f_1} a_2)$ such that $(f'_1 \in \mathscr{M}, g'_1 \in \mathscr{M})$ is a pushout over (g_1, f_1) with induced morphisms $f'_1, g'_1 \in \mathscr{M}$ and with pushout object $PO_2 = G$. **Hypothesis.** There is n such that the assumption holds. **Step.** For $n+1$, note that in $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, all decision paths of binary splits in $BSplits(G)$ lead to the same set of atoms $(a_i)_{i\in\{1,...,n+1\}}$ up to isomorphism, i.e., we focus on that path where atoms are splitted from G step-wise. Let G be binary splitted into $(G_1 \to G \leftarrow a_{n+1}) \in BSplits(G)$ with atom a_{n+1} and furthermore, let $(a_i)_{i\in\{1,...,n\}} \in Atoms(G_1)$ and $(a_{n+1}) \in Atoms(a_{n+1})$. By induction hypothesis, for $Atoms(G_1)$ there is pushout object $PO_n = G_1$ with induced morphisms in $\mathscr{M}$. Analogously to the base case by Cor. 3.2, for $(G_1 \xrightarrow{f'_n} G \xleftarrow{g'_n} a_{n+1}) \in BSplits(G)$ there is G_n and pushout object PO_{n+1} obtained by pushout $(f'_n \in \mathscr{M}, g'_n \in \mathscr{M}) = PO_n +_{G_n} a_{n+1}$ over $(g_n \in \mathscr{M}, f_n \in \mathscr{M})$ with $PO_{n+1} = G$ and with induced morphisms in $\mathscr{M}$ by $\mathscr{M}$-composition with $f'_n \in \mathscr{M}$.

□

Given a set of constraints C that are designated for general satisfaction. Then, Lem. 3.6 states that for each atom a of a graph G which generally satisfies C, the extension of a via C is in G again. The result does not generally hold for constraints that are designated for initial satisfaction, since, atom a may be extended at parts that are not covered by the satisfaction of the constraint potentially leading to extensions of a that are not in G.

Lemma 3.6 (Closure under C-Extensions of Atoms) *In $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, let C be a set of constraints that are designated for general satisfaction, $G \in \mathscr{L}(C)$ be a graph that generally satisfies C and $a \in A \in Atoms(G)$ be an atom of G with induced morphism $e\colon a \to G \in \mathscr{M}$ by Lem. 3.5. Then, G is* closed under C-Extensions of *a, i.e., for all extensions $E \in Extensions(a, C)$ there is $a_E \in E$ with morphisms $e_1\colon a \to a_E \in \mathscr{M}, e_2\colon a_E \to G \in \mathscr{M}$ and $e = e_2 \circ e_1$.* △

Proof. The proof is presented in appendix A.7. □

Def. 3.16 extends the construction of C-extensions in Def. 3.5 by defining the simultaneous extension of a set of graphs via a given set of constraints C. This leads to the result in Lem. 3.7 as an extension of Lem. 3.5. Based on Lem. 3.6, Lem. 3.7 states that given a graph G which generally satisfies a given set of constraints C, then all simoultaneous extensions of the atoms of G via C contains a variant such that the gluing of the extended atoms yields graph G again.

Definition 3.16 (C-Extensions of Sets of Graphs) Let A be a set of graphs and C be a set of constraints. The set $SELECT_E(A, C) = \{f\colon A \to B \mid B = \bigcup_{a\in A}(Extensions(a, C)), \forall a \in A\colon f(a) \in Extensions(a, C)\}$ contains all functions f that select for a given graph $a \in A$ an extension $E \in Extensions(a, C)$. Let $f_E \in SELECT_E(A, C)$, then the set $SELECT_{a_E}(A, C, f_E) = \{f\colon A \to \bigcup_{E\in B}(E) \mid B = \bigcup_{a\in A}(Extensions(a, C)), \forall a \in A\colon f(a) \in f_E(a)\}$ contains all functions f that select for a given graph $a \in A$ an extended graph $a_E \in f_E(a)$. △

Remark 3.8 *Note that for a graph G, the set of C-extensions $Extensions(G, C)$ may be infinite. Therefore, for a given set of graphs A, there may exists an infinite set of functions $SELECT_E(A, C)$. Furthermore, a C-extension of a graph may be an infinite set of extended* *graphs. Therefore, for each selection $f_E \in SELECT_E(A,C)$, there may exists an infinite set of functions $SELECT_{a_E}(A,C,f_E)$.* △

Lemma 3.7 (Gluing of C-Extended Atoms) *In $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, let C be a set of constraints that are designated for general satisfaction, $G \in \mathscr{L}(C)$ be a graph that generally satisfies C and $A = (a_i)_{i\in\{1,\dots,n\}} \in Atoms(G)$ be the atoms of G. Then, for all functions $f_E \in SELECT_E(A,C)$ that select a C-extension for each atom $a_i \in A$, there exists a function $f_{a_E} \in SELECT_{a_E}(A,C,f_E)$ that selects an extended atom for each atom $a_i \in A$ such that there exist graphs $(G^E_j)_{1\leq j\leq n-1}$ and pushouts $(PO^E_k +_{G^E_k} f_{a_E}(a_{k+1}) = PO^E_{k+1})_{k\in\{1,\dots,n-1\}}$ with pushout objects PO^E_{k+1}, all morphisms being in $\mathscr{M}$ and $PO^E_1 = f_{a_E}(a_1)$ where (pushout object) PO^E_n is G.* △

Proof. The proof is presented in appendix A.8. □

Note that in $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, verifying domain completeness in Thm. 3.3 via C-extension completeness in Def. 3.8 is performed on the level of attributed graphs sharing the *DSIG*-term algebra $T_{DSIG}(X)$, since, effective atoms and their extensions share algebra $T_{DSIG}(X)$ up to isomorphism by construction Def. 3.5 with $e \in \mathscr{M}$ and Sec. 2.2.2 and Rem. 2.3. In contrast to that, graphs of languages $\mathscr{L}(C)$ and $\mathscr{L}(GG)$ may have concrete algebras with concrete values, in general (cf. general assumption of this section). In order to close the algebra gap, in Def. 3.17 we define instance morphisms $i\colon A \to B$ from attributed graphs A with *DSIG*-term algebra $T_{DSIG}(X)$ to instance graphs B with concrete algebras where concrete attribute values in B are substituted by variables $x \in X$ in A such that i is an isomorphism on the graph part and injective for the data part of assigned attribute values. Furthermore, we show that the verification of domain completeness on the term level does also hold for all (possibly infinitely many) instantiations to concrete values. Therefore, given a set of productions P, in Lem. 3.15 we show that for each transformation $G \overset{*}{\Rightarrow} H$ via P on the term level with instance morphism $i_H\colon H \to H'$ there is a corresponding transformation $G' \overset{*}{\Rightarrow} H'$ via P with instance morphism $i_G\colon G \to G'$. For proving Lem. 3.15, based on the results in Lemmas 3.12 and 3.13 we first show in Lem. 3.14 that a match satisfies an AC-schema if and only if the match extended by a given instance morphism satisfies the AC-schema. Lem. 3.12 states that the successive merge over two morphisms is equivalent to the merge over the composition of both morphisms, i.e., in proofs for the satisfaction of AC-schemata, the merge over a composition of morphisms can be constructed step-wise (morphism by morphism successively). For proving Lem. 3.12, we additionally show the general results in Lemmas 3.8 to 3.11.

Definition 3.17 (Instance Morphism) Let $DSIG = (S, OP)$ be a data signature. In category $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, *a morphism $i\colon A \to B \in \mathscr{E}, \mathscr{O}$ is an instance morphism*, if:

1. Attributed graph A shares *DSIG*-term algebra $T_{DSIG}(X)$ with $X = (X_s)_{s\in S}$ being a family of infinite sets X_s of variables for each sort $s \in S$,
2. attributed graph B shares *DSIG*-algebra D_B,
3. morphism i is type strict,
4. all attribute values in A are variables $x \in X$:$(\forall e \in E^A_j . t^A_j(e) \in X)_{j\in\{NA,EA\}}\}$, and
5. the data part of assigned attribute values is injective: $\forall d_1, d_2 \in D_A.(i_D(d_1) = i_D(d_2)) \wedge i_D(d_1) \in (t^B_{NA}(E^B_{NA}) \cup t^B_{EA}(E^B_{EA})) \implies d_1 = d_2$. △

Lemma 3.8 ($\mathscr{O}$-Morphisms are Closed Under De-Composition and Pushouts) *Let $f : AG^1 \to AG^2$, $g\colon AG^2 \to AG^3$ and $f' : AG'^2 \to AG^3$ be morphisms in $\mathbf{AGraphs}_{ATGI}$.*

$$\begin{tikzcd}
AG^1 \arrow[r, "f"] \arrow[d] \arrow[dr, phantom, "(1)"] & AG^2 \arrow[d, "g"] \\
AG'^2 \arrow[r, "f'"] & AG^3
\end{tikzcd}$$

1. *If $f,g \in \mathcal{O}$, then $g \circ f \in \mathcal{O}$.*
2. *Let $g \circ f = h$.*
   (a) *If $h \in \mathcal{O}$, then $f \in \mathcal{O}$.*
   (b) *If $h \in \mathcal{O}$ and graph part f_S of f is an isomorphism, then $g \in \mathcal{O}$.*
3. *For a given pushout (1), $f \in \mathcal{O}$ implies $f' \in \mathcal{O}$.* △

Proof. We proof the three facts as follows.

1. The proof is given in [GLEO12], Thm. 7, item 7.
2. (a) The proof is given in [GLEO12], Thm. 7, item 7.
   (b) Let $f_S^{-1}\colon AG^2 \to AG^1$ be the inverse isomorphism of isomorphism f_S with $f_S \circ f_S^{-1} = id_{AG^2}$. From $g_S \circ f_S = h_S$ we obtain $g_S \circ f_S \circ f_S^{-1} = h_S \circ f_S^{-1} \Leftrightarrow g_S \circ id_{AG^2} = h_S \circ f_S^{-1} \Leftrightarrow g_S = h_S \circ f_S^{-1}$. Since, h_S, f_S^{-1} are componentwise injective in **Sets**, it follows that g_S is componentwise injective in **Sets**, i.e., $g \in \mathcal{O}$.
3. Since (1) is a pushout, f_S being componentwise injective in **Sets** implies that f'_S is componentwise injective in **Sets** (cf. Fact 2.17, item 1 in [EEPT06]) and thus, $f' \in \mathcal{O}$.

□

Lemma 3.9 (De-Composition of Pairs of Jointly Epimorphic Morphisms)

$$\begin{tikzcd}
G_0 \arrow[r, "e_2"] \arrow[d, "e"] \arrow[dr, phantom, "(1)"] & G_1 \arrow[d, "e_3"] & H \arrow[l, "e_1"'] \\
G_2 \arrow[r, "e_4"] & G &
\end{tikzcd}$$

1. *Let (e_1,e_2) and (e_3,e_4) be pairs of jointly epimorphic morphisms and furthermore, (1) commutes. Then, $(e_3 \circ e_1, e_4)$ is a pair of jointly epimorphic morphisms.*
2. *Let $(e_3 \circ e_1, e_4)$ be a pair of jointly epimorphic morphisms. Then, (e_3,e_4) is a pair of jointly epimorphic morphisms.*
3. *In the category $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$, let (e_3,e_4) be a pair of jointly epimorphic morphisms, $e_4 \in \mathcal{M}$ and $e_1 \in \mathcal{E}$ be an extremal morphism w.r.t. $\mathcal{M}$. Then, $(e_3 \circ e_1, e_4)$ is a pair of jointly epimorphic morphisms.*
4. *Let (e_1,e_2) be jointly epimorphic and e_3 be an epimorphism. Then, $(e_3 \circ e_1, e_3 \circ e_2)$ is a pair of jointly epimorphic morphisms.* △

Proof. 1. We assume that (1) commutes and pairs (e_1,e_2) and (e_3,e_4) are jointly epimorphic. By the definition of jointly epimorphic morphisms, we have to show that for all graphs G' and morphisms $f,g\colon G \to G'$ it holds that if $f \circ e_3 \circ e_1 = g \circ e_3 \circ e_1$ $^{(*^1)}$ and $f \circ e_4 = g \circ e_4$ $^{(*^2)}$, then $f = g$. Let $f,g\colon G \to G'$ be two morphisms with assumptions $(*^1)$ and $(*^2)$. Assumption $(*^2)$ implies $f \circ e_4 \circ e = g \circ e_4 \circ e$. Since, (1) commutes it follows that $f \circ e_3 \circ e_2 = g \circ e_3 \circ e_2$. By assumption $(*^1)$ and (e_1,e_2) being jointly epimorphic it follows that $f \circ e_3 = g \circ e_3$. By assumption $(*^2)$ and (e_3,e_4) being jointly epimorphic it follows that $f = g$.

2. We have to show for all morphisms f, g that $f \circ e_3 = g \circ e_3$ and $f \circ e_4 = g \circ e_4$ implies $f = g$. Thus, $f \circ e_3 = g \circ e_3 \implies f \circ e_3 \circ e_1 = g \circ e_3 \circ e_1$. With $f \circ e_4 = g \circ e_4$ it follows that $f = g$, since, $(e_3 \circ e_1, e_4)$ are jointly epimorphic.

3. By assumption $e_4 \in \mathcal{M}$ and Sec. 2.2.2 and Rem. 2.3, e_4 is an isomorphism, i.e., an epimorphism, on the data part $e_{4,D}$. It remains to show that $(e_3 \circ e_1, e_4)$ are jointly epimorphic on the graph part. We assume the opposite. Then, there exists $g \in G$ with $g \notin e_4(G_2)$ and $g \notin e_3(e_1(H))$. The assumption that (e_3, e_4) are jointly epimorphic implies that there exists $g' \in G_1$ with $e_3(g') = g$ but $g' \notin e_1(H)$. Let $m \circ e' = e_1$ be the extremal $\mathcal{E}$-$\mathcal{M}$ factorisation of e_1 with $m\colon \overline{G_1} \to G_1 \in \mathcal{M}$. By the construction of the factorisation via the standard epi-mono factorisation on the graph structure part (cf. [BEGG10]), it follows that $g' \notin m(\overline{G_1})$ and therefore, m is not an epi-(iso-)morphism. This contradicts with assumption $e_1 \in \mathcal{E}$ is extremal where $m \in \mathcal{M}$ implies that m is an isomorphism (cf. Def. 2.9).

4. Given morphisms $g\colon G \to A$ and $h\colon G \to B$ with $g \circ e_3 \circ e_1 = h \circ e_3 \circ e_1$ and $g \circ e_3 \circ e_2 = h \circ e_3 \circ e_2$. By (e_1, e_2) being jointly epimorphic, it follows that $g \circ e_3 = h \circ e_3$ and e_3 being an epimorphism implies that $g = h$.

□

In general, it is not true that a monomorphism that is also an epimorphism is an isomorphism, since, the inverse morphism may not exist. However, in $\mathcal{M}$-adhesive categories this assumption holds for a subclass $\mathcal{M}$ of monomorphisms as shown by Lem. 3.10.

Lemma 3.10 (Relationship between $\mathcal{M}$-, Epi- and Iso-Morphisms in $\mathcal{M}$-adhesive Categories) *Given an $\mathcal{M}$-adhesive category $(\mathbf{C}, \mathcal{M})$, then any $\mathcal{M}$-morphism that is also an epimorphism is an isomorphism.* △

Proof. The proof is based on the proof of Lemma 4.9 in [LS99] for adhesive categories. Let (1) be a commuting diagram with epimorphism $m\colon A \to B \in \mathcal{M}$. From m being an epimorphism, it follows that (1) is a pushout, since, diagram (1) already commutes. It remains to show the universal pushout property. For all morphisms $b_1, b_2\colon B \to \underline{B}$ with $b_1 \circ m = b_2 \circ m$ it follows that $b_1 = b_2$ by the definition of epimorphisms. Furthermore, there exists a morphism $\underline{b} = b_1 = b_2$ with $\underline{b} \circ id_B = b_1 \circ id_B = b_2 \circ id_B = b_1 = b_2$. Moreover, $\underline{b}$ is unique. Assume that there exists a morphism $\underline{b}'\colon B \to \underline{B}$ with $\underline{b}' \neq \underline{b}$ and $\underline{b}' \circ id_B = \underline{b}' = b_1 = b_2 = \underline{b}$ leading to a contradiction.

Since, (1) is a pushout along $\mathcal{M}$-morphism m, (1) is also a pullback (cf. Theorem 4.26, item 1 in [EEPT06]). By the universal pullback property, there exists the inverse morphism $m^{-1}\colon B \to A$ with $m \circ m^{-1} = id_B$. Furthermore, $m \circ id_A = id_B \circ m = m \circ m^{-1} \circ m$ and $m \in \mathcal{M}$ being a monomorphism by the definition of class $\mathcal{M}$ (cf. Def. 4.13 in [EEPT06]) implies $m^{-1} \circ m = id_A$.

$$\begin{tikzcd}
B \arrow[rrd, bend left, "id_B"] \arrow[rd, "m^{-1}"] \arrow[rdd, bend right, "id_B"'] & & & \\
& A \arrow[r, hook, two heads, "m"] \arrow[d, hook, "m"'] \arrow[rd, phantom, "(1)"] & B \arrow[d, "id_B"] \arrow[rdd, bend left, "b_2"] & \\
& B \arrow[r, "id_B"'] \arrow[rrd, bend right, "b_1"'] & B \arrow[rd, "\underline{b}"] & \\
& & & \underline{B}
\end{tikzcd}$$

□

Lemma 3.11 (Preservation of Extremal Morphisms)
Let $(\mathbf{C},\mathcal{M})$ be an $\mathcal{M}$-adhesive category and $e \in \mathcal{E}$ be an extremal morphism with respect to $\mathcal{M}$ in $\mathbf{C}$. Given diagram (1), if (1) commutes and morphisms (a',e') are jointly epimorphic, then morphism e' is an extremal morphism with respect to $\mathcal{M}$ in $\mathbf{C}$, i.e., $e' \in \mathcal{E}$. △

$$\begin{tikzcd}
C \arrow[d, "e'"'] \arrow[dr, phantom, "(1)"] & P \arrow[l, "a"'] \arrow[d, "e\in\mathcal{E}"] \\
C' & P' \arrow[l, "a'"]
\end{tikzcd}$$

Proof. Let $(\mathbf{C},\mathcal{M})$ be an $\mathcal{M}$-adhesive category, $e \in \mathcal{E}$, $e' \circ a = a' \circ e$ and (a',e') be jointly epimorphic. By the definition of extremal morphisms (cf. Def. 2.9), morphism e' is extremal ($e' \in \mathcal{E}$) if for all morphisms c,c' with $c' \circ c = e'$ it is true that $c' \in \mathcal{M}$ implies c' is an isomorphism. Given a decomposition of $e' \colon C \to C'$ with morphisms $c \colon C \to \underline{C}$, $c' \colon \underline{C} \to C'$, $c' \circ c = e'$ and $c' \in \mathcal{M}$. Since pullbacks exist along M-morphisms in $\mathbf{C}$ and $\mathcal{M}$-morphisms are closed under pullbacks (cf. Def. 4.9 in [EEPT06]), we can construct pullback (PB) along $c' \in \mathcal{M}$ with $p' \in \mathcal{M}$. By the universal pullback property and assumption $e' \circ a = c' \circ c \circ a = a' \circ e$, there exists morphism $p \colon P \to \underline{P}$ with $\underline{a} \circ p = c \circ a$ and $p' \circ p = e$. Thus, $e \in \mathcal{E}$, $p' \circ p = e$ and $p' \in \mathcal{M}$ implies p' is an isomorphism (cf. Sec. 2.2.2 and Def. 2.9). It remains to show that c' is an epimorphism. By $c' \in \mathcal{M}$ and Lem. 3.10 it would follow that c' is an isomorphisms and thus, e' is extremal. Morphism c' is an epimorphism means that for all morphisms $f,g \colon C' \to G$ it holds that $f \circ c' = g \circ c'$ implies $f = g$ (cf. Def. 2.13 in [EEPT06]). Let $f \circ c' = g \circ c'$. Therefore, $f \circ e' = f \circ c' \circ c = g \circ c' \circ c = g \circ e'$ $(*^1)$. Furthermore, $f \circ a' = f \circ a' \circ id_{P'} = f \circ a' \circ p' \circ p'^{-1} = f \circ c' \circ \underline{a} \circ p'^{-1} = g \circ c' \circ \underline{a} \circ p'^{-1} = g \circ a' \circ p' \circ p'^{-1} = g \circ a'$ $(*^2)$ with p'^{-1} being the inverse morphism of isomorphism p'. Since, (a',e') are jointly epimorphic, $(*^1)$ and $(*^2)$ implies $f = g$. Therefore, c' is an epimorphism and by Lem. 3.10 c' is an isomorphism and thus, $e' \in \mathcal{E}$.

$$\begin{tikzcd}
& C \arrow[dr, "c"] \arrow[dd, "e'"'] & & & P \arrow[llll, phantom] \arrow[lll, "a"'] \arrow[dl, "p"'] \arrow[dd, "e\in\mathcal{E}"] \\
& & \underline{C} \arrow[dl, hook, "c'"] \arrow[dr, phantom, "(PB)"] & \underline{P} \arrow[l, "\underline{a}"'] \arrow[dr, hook, "p'"] & \\
G & C' \arrow[l, shift right, "f"'] \arrow[l, shift left, "g"] & & & P' \arrow[lll, "a'"]
\end{tikzcd}$$

□

In general, the merge of a condition over two morphisms successively is not equivalent to the merge of the condition over the composition of both morphisms as illustrated by Ex. 3.11. Lem. 3.12 defines sufficient conditions under which the equivalence holds.

Example 3.11 (General Equivalence of Successive Merge and Merge over Composition) *Fig. 3.6 illustrates the merge* $\mathrm{Merge}(b_2 \circ b_1, ac_P)$ *of condition* $ac_P = \exists(a \colon P \to C, \textbf{true})$ *over the composition of morphisms* b_1 *and* b_2 *as well as the merge* $\mathrm{Merge}(b_2, \mathrm{Merge}(b_1, ac_P))$ *of* ac_P *over morphisms* b_1 *and* b_2 *successively. The example serves as a counterexample for showing that the merge over a composition of two morphisms is not equivalent to the merge over both morphisms successively, in general. For simplicity, the graphs in the example share no algebras and all nodes are of type* $T0$*. The nodes are mapped to nodes of the same name along morphisms. The merge over composition is constructed by the commuting outer diagram with morphisms* $b' \in \mathcal{O}$ *and* $a'' \in \mathcal{M}$ *and results in condition* $\exists(a'' \colon P'' \to C'', \textbf{true}) = \mathrm{Merge}(b_2 \circ b_1, ac_P)$*. The successive merge results in condition* $\textbf{false} = \mathrm{Merge}(b_2, \mathrm{Merge}(b_1, ac_P))$*. Diagram* (1) *can be constructed for the first merge* $\mathrm{Merge}(b_1, ac_P)$ *of* ac_P *over morphism* b_1 *but diagram* (2) *does not* *exist for the subsequent merge over b_2, since, the merge construction requires that the morphism between graphs C' and C'' must be an $\mathcal{O}$-morphism where the nodes are mapped injectively (cf. Def. 2.16) but no $\mathcal{O}$-morphism can be found. This results in condition* **false** *for the sucessive merge (cf. Rem. 2.11). Thus,* $\mathrm{Merge}(b_2, \mathrm{Merge}(b_1, ac_P)) \not\equiv \mathrm{Merge}(b_2 \circ b_1, ac_P)$*, i.e., the merge of a condition over two morphisms successively is not equivalent to the merge of the condition over the composition of both morphisms, in general. The same situation arises if ac_P is not in $\mathcal{M}$-normal form ($a \notin \mathcal{M}$) and therefore, a identifies elements that are not identified by b_1 but by b_2 and therefore, (1) cannot be constructed resulting into* $\mathrm{Merge}(b_1, ac_P) = \textbf{false} \Rightarrow \mathrm{Merge}(b_2, \mathrm{Merge}(b_1, ac_P)) = \textbf{false} \not\equiv \mathrm{Merge}(b_2 \circ b_1, ac_P)$ *(cf. Rem. 2.11).* △

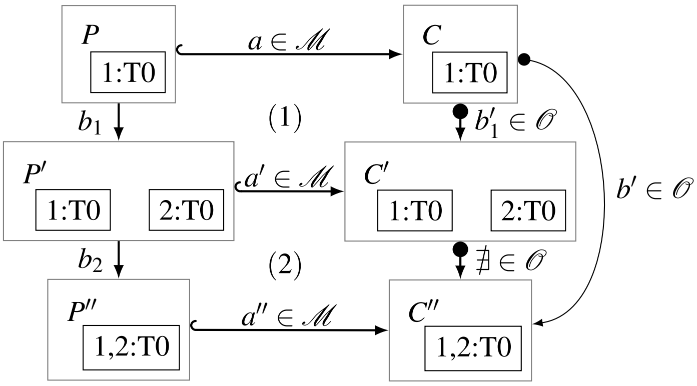


Figure 3.6: Counterexample for General Equivalence

Lemma 3.12 (Equivalence of Successive Merge and Merge over Composition) *In* $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$*, let ac_P be a condition over P in $\mathcal{M}$-normal form. Furthermore, let $b_1 \colon P \to P'$ and $b_2 \colon P' \to P''$ be morphisms from P or P' to some P' or P'', respectively. Furthermore, let $b_1 \in \mathcal{E}$. It holds that* $\mathrm{Merge}(b_2, \mathit{Merge}(b_1, ac_P)) \equiv \mathrm{Merge}(b_2 \circ b_1, ac_P)$. △

$$\begin{tikzcd}
ac_P \triangleright P \arrow[d, "b_1 \in \mathcal{E}"'] \\
\mathrm{Merge}(b_1, ac_P) \triangleright P' \arrow[d, "b_2"'] \\
\mathrm{Merge}(b_2 \circ b_1, ac_P) \equiv \mathrm{Merge}(b_2, \mathrm{Merge}(b_1, ac_P)) \triangleright P''
\end{tikzcd}$$

Proof. The proof is presented in appendix A.9. □

Lemma 3.13 (Composition of Extremal Morphisms) *Let $(\mathbf{C}, \mathcal{M})$ be an $\mathcal{M}$-adhesive category and $e_1 \colon A \to B, e_2 \colon B \to C \in \mathcal{E}$ be extremal morphisms with respect to $\mathcal{M}$ in* $\mathbf{C}$*. The composition $e_2 \circ e_1$ is also an extremal morphism with respect to $\mathcal{M}$ in* $\mathbf{C}$ *($e_2 \circ e_1 \in \mathcal{E}$).* △

Proof. Let $e_1, e_2 \in \mathcal{E}$ and $m \circ f$ be a factorisation of $e_2 \circ e_1$ with $m \circ f = e_2 \circ e_1$ and $m \in \mathcal{M}$. It remains to show that m is an isomorphism (cf. Def. 2.9). We can construct pullback (1) along $m \in \mathcal{M}$ with $m_1 \in \mathcal{M}$, since, pullbacks exist along $\mathcal{M}$-morphisms in $\mathbf{C}$ and $\mathcal{M}$-morphisms are closed under pullbacks by definition (cf. Def. 4.13, item 2 in [EEPT06]). By the universal pullback property we obtain morphism f_1 with $m_1 \circ f_1 = e_1$. Thus, the assumption $e_1 \in \mathcal{E}$ with $m_1 \in \mathcal{M}$ imply that m_1 is an isomorphism (cf. Def. 2.9). (1) being a pullback implies $e_2 \circ m_1 = m \circ f_2 \overset{m_1\ iso}{\Longrightarrow} e_2 \circ m_1 \circ m_1^{-1} = m \circ f_2 \circ m_1^{-1} \Leftrightarrow e_2 \circ id_B = m \circ f_2 \circ m_1^{-1} \Leftrightarrow e_2 = m \circ f_2 \circ m_1^{-1}$. Thus, the assumptions $e_2 \in \mathcal{E}$ and $m \in \mathcal{M}$ imply that m is an isomorphism (cf. Def. 2.9).

$$\begin{tikzcd}
 & \overline{B} \arrow[ddr, hook, "m\in\mathcal{M}"] & \\
 & D \arrow[u, "f_2"'] \arrow[d, hook, "m_1"] \arrow[r, phantom, "(1)"] & {} \\
A \arrow[uur, "f"] \arrow[ur, "f_1"] \arrow[r, two heads, "e_1\in\mathcal{E}"'] & B \arrow[r, two heads, "e_2\in\mathcal{E}"'] & C
\end{tikzcd}$$

(=)

□

Lem. 3.14 states that for a given match $m\colon P \to G$ and instance morphism $i\colon G \to G'$, m satisfies the AC-schema $\overline{ac_P}$ of a given condition ac_P in $\mathcal{M}$-normal form ($m \models \overline{ac_P}$) if and only if m extended by i satisfies $\overline{ac_P}$ ($i \circ m \models \overline{ac_P}$). For direction $i \circ m \models \overline{ac_P} \Rightarrow m \models \overline{ac_P}$, the restriction to conditions in $\mathcal{M}$-normal form becomes essential as illustrated by Ex. 3.12 leading to the restriction to application conditions in $\mathcal{M}$-normal form in Lem. 3.15.

Example 3.12 (AC-schema Satisfaction by Instance Morphisms for General Conditions) *For category* ($\mathbf{AGraphs}_{ATGI}, \mathcal{M}$)*, Fig. 3.7 illustrates that given a condition* $ac_P = \exists(a\colon P \to C, \mathbf{true})$ *that is not in* $\mathcal{M}$*-normal form, then it may be true that* $i \circ m \models \overline{ac_P}$ *but* $\neg(m \models \overline{ac_P})$*. Condition* ac_P *is not in* $\mathcal{M}$*-normal form, since, variables* y *and* z *are identified by* z *along morphism* $a\colon P \to C$*, i.e.,* $a \notin \mathcal{M}$ *(cf. Sec. 2.2.2 and Rem. 2.3). By Sec. 2.2.3 and Rem. 2.13,* $m \models \overline{ac_P}$ *means that* $m_1 \models \mathrm{Merge}(e_1, ac_P)$ *and* $i \circ m \models \overline{ac_P}$ *means that* $m_2 \models \mathrm{Merge}(e_2 \circ e_1, ac_P)$ *for the extremal* $\mathcal{E}$*-*$\mathcal{M}$ *factorisations* $m_1 \circ e_1 = m$ *and* $m_2 \circ e_2 \circ e_1 = i \circ m$*. Note that extremal* $\mathcal{E}$*-*$\mathcal{M}$ *factorisation* $m_2 \circ e_2 \circ e_1$ *of* $i \circ m$ *is obtained as follows: Given extremal* $\mathcal{E}$*-*$\mathcal{M}$ *factorisation* $m_2 \circ e_2$ *of* $i \circ m_1$*, then by Lem. 3.13 it follows that* $e_2 \circ e_1 \in \mathcal{E}$ *with* $m_2 \circ e_2 \circ e_1 = i \circ m$ *and furthermore, by the uniqueness of factorisations (cf. Sec. 2.2.2 and Rem. 2.5), it follows that* $m_2 \circ e_2 \circ e_1$ *is the*

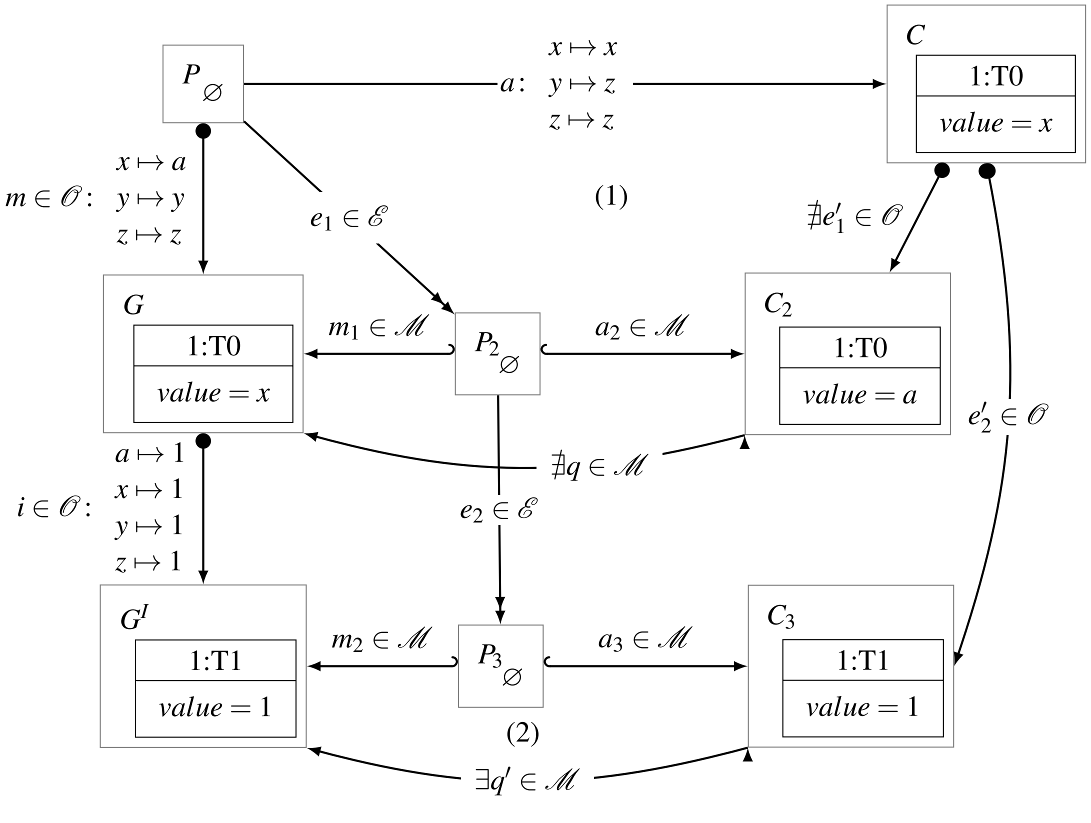


Figure 3.7: AC-Schema Satisfaction by Instance Morphisms for General Conditions

extremal $\mathcal{E}$-$\mathcal{M}$ factorisation of $i \circ m$. Furthermore, $\mathrm{Merge}(e_1, ac_P) = \textbf{\textit{false}}$*, since, there does not exist $e'_1 \colon C \to C_2$ such that (1) commutes, as, variables y and z in P are identified by z in C along $a \colon P \to C$ but not by morphism m and therefore, also not by e_1 (cf. Sec. 2.2.3 and Rem. 2.11). Contrarily,* $\mathrm{Merge}(e_2 \circ e_1, ac_P) = \exists(a_3 \colon P_3 \to C_3, \textbf{true})$*. Thus, $m_2 \models \exists(a_3 \colon P_3 \to C_3, \textbf{true})$ for morphism $q' \colon C_3 \to G^I \in \mathcal{M}$ with commuting (2) but $\neg(m_1 \models \textbf{\textit{false}})$. Consequently, $i \circ m \models \overline{ac}_P$ but $\neg(m \models \overline{ac}_P)$. For negative conditions $\neg\overline{ac}_P$ this means that $m \models \neg\overline{ac}_P$ but $\neg(i \circ m \models \overline{ac}_P)$. For Lem. 3.15 this means that without the restriction to application conditions in $\mathcal{M}$-normal form, we obtain the undesired result that transformations on the term level via productions with negative application conditions not necessarily induce corresponding transformations on the level of concrete values, as, the application condition may be fulfilled by match m but not by match $i \circ m$. Independently from conditions not in $\mathcal{M}$-normal form, a similar situation may arise if we disregard Def. 3.17 and Item 5 for instance morphisms and focus on variable x only in the example from above while neglecting variables y and z, i.e., this time condition ac_P is in $\mathcal{M}$-normal form. Therefore, the following example demonstrates the importance of Item 5 for the definition of instance morphisms. Variable x in P is mapped to variable a in G along m. Furthermore, variables x and a in G are both mapped to* 1 *in G^I along i. Note that this is only possible when explicitly disregarding Def. 3.17 and Item 5 for instance morphism i. We can construct* (1) *with* $\mathrm{Merge}(e_1, ac_P) = \exists(a_2 \colon P_2 \to C_2, \textbf{true})$*. However, there does not exist $q \colon C_2 \to G$ such that $q \circ a_2 = m_1$, since, this requires that variable a in C_2 is simultaneously mapped to variable a and x in G along q. Thus,* $\neg(m_1 \models \mathrm{Merge}(e_1, ac_P))$ *implying further that $\neg(m \models \overline{ac}_P)$. In contrast to that, we can construct* $\mathrm{Merge}(e_2 \circ e_1, ac_P) = \exists(a_3 \colon P_3 \to C_3, \textbf{true})$ *such that there is $q' \colon C_3 \to G^I$ with $q' \circ a_3 = m_2$, i.e., $m_2 \models$* $\mathrm{Merge}(e_2 \circ e_1, ac_P)$ *implying further that $i \circ m \models \overline{ac}_P$. Therefore, again $i \circ m \models \overline{ac}_P$ but $\neg(m \models \overline{ac}_P)$.* △

Lemma 3.14 (AC-schema Satisfaction by Instance Morphisms) *Given a condition ac_P over P in $\mathcal{M}$-normal form and its AC-schema $\overline{ac}_P$. Furthermore, given a match $m \colon P \to G \in \mathcal{O}$ to some graph G and an instance morphism $i \colon G \to G^I$. Then, in $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ it holds that $m \models \overline{ac}_P$ if and only if $i \circ m \models \overline{ac}_P$.* △

Proof. The proof is presented in appendix A.10. □

Remark 3.9 *In Lem. 3.14, $i \circ m \models \overline{ac}_P$ is well defined w.r.t. Def. 2.12, since, each instance morphism i is in $\mathcal{O}$ by definition. Lem. 3.8, item 1 with $m \in \mathcal{O}$ imply that $i \circ m \in \mathcal{O}$.* △

Lemma 3.15 (Abstract transformations induce transformations of instances) *Let $G_0 \overset{*}{\Rightarrow} G_n$ be a transformation via productions P and almost injective matches $m \in \mathcal{O}$ only, and with all application conditions in $\mathcal{M}$-normal form. Let $i_n \colon G_n \to G_n^I$ be an instance morphism. Then, in* $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ *there exists a transformation $G_0^I \overset{*}{\Rightarrow} G_n^I$ via productions P with instance morphism $i_0 \colon G_0 \to G_0^I$.* △

Proof. Let $G_{n-1} \xRightarrow{(\overline{p}_{n-1}, m_{n-1})} G_n$ with pushout (1) be the last direct transformation step of transformation $G_0 \overset{*}{\Rightarrow} G_n$ via production $\overline{p}_{n-1} = (p_{n-1} \in \mathcal{M}, ac_{n-1})$ with application condition ac_{n-1} in $\mathcal{M}$-normal form and via match $m_{n-1} \in \mathcal{O}$ by assumption. Note that $g \in \mathcal{M}$ by $\mathcal{M}$-morphisms p_{n-1} are closed under pushouts. Therefore, $m_{n-1} \models \overline{ac}_{n-1}$ by Sec. 2.2.4 and Def. 2.22, with $\overline{ac}_{n-1}$ being the AC-schema of ac_{n-1}.

$$\begin{tikzcd}
 & \overline{ac}_{n-1} \arrow[r, phantom, "\triangleright"] & L_{n-1} \arrow[r, hook, "p_{n-1}\in\mathcal{M}"] \arrow[d, "m_{n-1}"] \arrow[dr, phantom, "(1)"] & R_{n-1} \arrow[d, "m'_{n-1}"] & \\
G_0 \arrow[r, Rightarrow, "(\overline{p}_0{,}m_0)"] \arrow[d, dotted, "i_0"] & \cdots \arrow[r, Rightarrow, "(\overline{p}_{n-2}{,}m_{n-2})"] & G_{n-1} \arrow[r, hook, "g\in\mathcal{M}"] \arrow[d, dotted, "i_{n-1}"] \arrow[dr, phantom, "(2)"] & G_n \arrow[d, "i_n"] \arrow[dr, phantom, "(3)"] & B \arrow[l, hook', "b\in\mathcal{M}"'] \arrow[ll, bend right, "b^*"'] \arrow[d, "f"] \\
G^I_0 \arrow[r, dotted, Rightarrow, "(\overline{p}_0{,}i_0\circ m_0)"] & \cdots \arrow[r, dotted, Rightarrow, "(\overline{p}_{n-2}{,}i_{n-2}\circ m_{n-2})"] & G^I_{n-1} \arrow[r, dotted, hook, "g'\in\mathcal{M}"] & G^I_n & C \arrow[l, hook', "c\in\mathcal{M}"']
\end{tikzcd}$$

By Def. 3.17, instance morphism i_n is in $\mathcal{E}$ and $\mathcal{O}$ and therefore, graph part $i_{n,S}$ is an isomorphism in $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$. Therefore, initial pushout (3) for i_n can be constructed with empty boundary graph B on the graph part with data part D_{G_n} and empty context graph C on the graph part with data part $D_{G^I_n}$ where b_S is the empty morphism on the graph part and b_D is the identity on the data part (cf. Fact 10.7, Item 2, and Def. 10.5, Item 2, in [EEPT06]). Note that $g \in \mathcal{M}$ implies that g_D is an isomorphism with inverse isomorphism g_D^{-1} by Sec. 2.2.2 and Rem. 2.3. Thus, there is morphism $b^*\colon B \to G_{n-1}$ with b^*_S being the empty morphism and $b^*_D = g_D^{-1}$ such that $g \circ b^* = b$. Therefore, pushout complement (g', i_{n-1}) over (g, i_n) exists leading to pushout (2) (cf. Sec. 2.2.2 and Rem. 2.2). By pushout composition, $(1)+(2)$ is a pushout. It remains to show that $i_{n-1}\colon G_{n-1} \to G^I_{n-1}$ is an instance morphism which would imply that $i_{n-1} \circ m_{n-1} \models \overline{ac}_{n-1}$ by Lem. 3.14 and therefore, $(1)+(2)$ is a direct transformation step $G^I_{n-1} \xRightarrow{(\overline{p}_{n-1}, i_{n-1}\circ m_{n-1})} G^I_n$ via production $\overline{p}_{n-1}$ and match $i_{n-1} \circ m_{n-1}$. According to Def. 3.17:

1. Morphism $i_{n-1} \in \mathcal{O}$, since, $g \in \mathcal{M} \overset{Sec.\ 2.2.2 and\ Rem.\ 2.3}{\Rightarrow} g \in \mathcal{O}$ and instance morphism $i_n \in \mathcal{O}$ by Def. 3.17 $\overset{Lem.\ 3.8 and\ Item\ 1}{\Rightarrow} i_n \circ g \in \mathcal{O} \overset{(2)}{\Rightarrow} g' \circ i_{n-1} \in \mathcal{O} \overset{Lem.\ 3.8 and\ Item\ 2a}{\Rightarrow} i_{n-1} \in \mathcal{O}$,

2. Morphism $i_{n-1} \in \mathcal{E}$, i.e., graph part $i_{n-1,S}$ is an epimorphism (surjective) in $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$: Assume that $i_{n-1,S}$ is not surjective, i.e., there is graph element $e \in G^I_{n-1}$ with $e \notin i_{n-1,S}(G_{n-1})$. Furthermore, graph element $g'(e) \notin i_n(G_n)$ by construction of pushouts in $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ (cf. Fact 2.17 in [EEPT06]). This contradicts with assumption i_n being an instance morphism, i.e., $i_n \in \mathcal{E}$ implying further that $i_{n,S}$ is an epimorphism (surjective) by Def. 3.17. Thus, $i_{n-1,S}$ is surjective implying further that $i_{n-1} \in \mathcal{E}$ in $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$,

3. Graph G_{n-1} shares *DSIG*-term algebra by $g\colon G_{n-1} \to G_n \in \mathcal{M}$ being an isomorphism on the data part and G_n shares *DSIG*-term algebra by Def. 3.17 and Item 1 for instance morphism $i_n\colon G_n \to G^I_n$,

4. Morphism i_{n-1} is type strict: Morphism $g \in \mathcal{M}$ and instance morphism i_n are type strict by Def. 3.17 and Item 3 (so $i_n \circ g$ is type strict). Assume that i_{n-1} is not type strict, so $g' \circ i_{n-1} \overset{(2)}{=} i_n \circ g$ is not type strict contradicting with $i_n \circ g$ is type strict. Thus, i_{n-1} is type strict,

5. All attribute values in G_{n-1} are variables $x \in X$: Assume that $\exists e \in E_j^{G_{n-1}}.t_j^{G_{n-1}}(e) \notin X, j \in \{NA, EA\}$. Note that $g_D(t_j^{G_{n-1}}(e)) \overset{g\in Mor}{=} t_j^{G_n}(g_{G,E_j}(e)) \in X$ by Def. 3.17 and Item 4 for instance morphism i_n, i.e., homomorphism g_D maps a term $t \notin X$ to a variable $x \in X$ which contradicts Fact B.16, Item 1, in [EEPT06] where $\overline{asg}$ is the unique homomorphism between two algebras for a given variable assignment that explicitly maps terms $t \notin X$ to

terms $t' \notin X$ by Def. B.14 in [EEPT06]. Therefore, $\forall e \in E_j^{G_{n-1}}.t_j^{G_{n-1}}(e) \in X, j \in \{NA, EA\}$, and

6. The data part of assigned attribute values is injective: Assume that $\exists d_1, d_2 \in D_{G_{n-1}}.(i_{n-1,D}(d_1) = i_{n-1,D}(d_2)) \land i_{n-1,D}(d_1) \in (t_{NA}^{G^I_{n-1}}(E_{NA}^{G^I_{n-1}}) \cup t_{EA}^{G^I_{n-1}}(E_{EA}^{G^I_{n-1}}))$ and where $d_1 \neq d_2$ implying further that $g'_D(i_{n-1,D}(d_1)) = g'_D(i_{n-1,D}(d_2)) \overset{(2)}{\Rightarrow} i_{n,D}(g_D(d_1)) = i_{n,D}(g_D(d_2))$ where $g_D(d_1) \in D_{G_n} \neq g_D(d_2) \in D_{G_n}$ by $g \in \mathscr{M}$ and therefore, g_D being an isomorphism. Furthermore, $i_{n-1,D}(d_1) \in (t_{NA}^{G^I_{n-1}}(E_{NA}^{G^I_{n-1}}) \cup t_{EA}^{G^I_{n-1}}(E_{EA}^{G^I_{n-1}})) \Rightarrow$ for $j \in \{NA, EA\}$, $\exists e \in E_j^{G^I_{n-1}}.t_j^{G^I_{n-1}}(e) = i_{n-1,D}(d_1) \Rightarrow t_j^{G^I_n}(g'_{G,E_j}(e)) \overset{g' \in Mor}{=} g'_D(t_j^{G^I_{n-1}}(e)) = g'_D(i_{n-1,D}(d_1)) \overset{(2)}{=} i_{n,D}(g_D(d_1)) \Rightarrow i_{n,D}(g_D(d_1)) \in (t_{NA}^{G^I_n}(E_{NA}^{G^I_n}) \cup t_{EA}^{G^I_n}(E_{EA}^{G^I_n}))$. This contradicts Def. 3.17 and Item 5 for instance morphism i_n. Therefore, $\forall d_1, d_2 \in D_{G_{n-1}}.(i_{n-1,D}(d_1) = i_{n-1,D}(d_2)) \land i_{n-1,D}(d_1) \in (t_{NA}^{G^I_{n-1}}(E_{NA}^{G^I_{n-1}}) \cup t_{EA}^{G^I_{n-1}}(E_{EA}^{G^I_{n-1}})) \implies d_1 = d_2$.

Analogously, we can iterate over all direct transformation steps back to G_0 and yield a transformation $G^I_0 \overset{*}{\Rightarrow} G^I_n$ via the same productions with instance morphism $i_0 \colon G_0 \to G^I_0$. □

Effectively, Lemmas 3.2, 3.5, 3.7 and 3.15 are used to prove the main result for verifying domain completeness in Thm. 3.3.

3.3 Limitations

The verification approach to solve the domain completeness problem in Sec. 3.2 and Thm. 3.3 is defined under the following assumptions which are likewise the limitations of the presented approach:

1. The conditions for verifying domain completeness are only sufficient but not necessary.
2. The approach is only applicable to graph grammars with empty start graphs.
3. The approach is only applicable to graph grammars with non-deleting productions.
4. The approach only involves constraints that are designated for general satisfaction while neglecting constraints that are designated for initial satisfaction.
5. Constraints and application conditions need to be in $\mathscr{M}$-normal form.
6. Termination of the approach requires an upper bound.

3.3.1 Conditions are Sufficient but not Necessary

The conditions for verifying domain completeness in Thm. 3.3 are only sufficient but not necessary. Consider a grammar GG with duplicates of creating productions but for which the domain completeness holds. The duplicates lead to a non-C-conflict-freeness of the marking rules $m(GG)$ when claiming critical pairs of same rules and same matches. Thus, the condition does not hold. However, it is simple to exclude duplicates from the grammar. For the confluence of marking rules we can find a similar example.

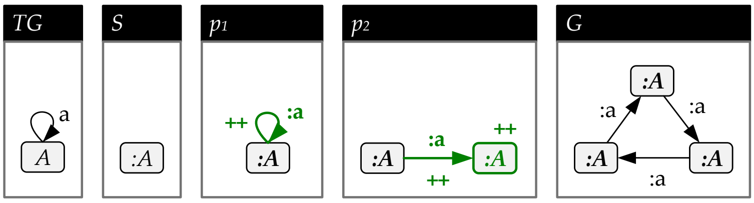


Figure 3.8: Limitation: Graph Grammar with Non-Empty Start Graph S

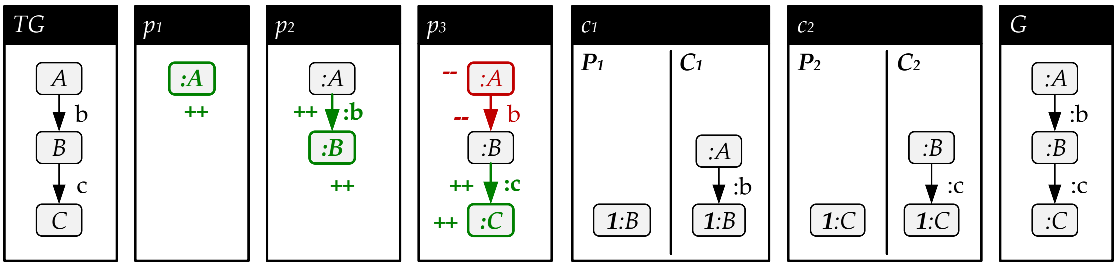


Figure 3.9: Limitation: Graph Grammar with Deleting Production p_3

3.3.2 Graph Grammars with Non-Empty Start Graph

We assume graph grammar $GG = (S,P)$ typed over type graph TG, with start graph S and productions $P = \{p_1, p_2\}$ as depicted in Fig. 3.8. Furthermore, we assume that the set of domain constraints $C = \varnothing$ is empty. By Def. 3.8, $\mathscr{L}(GG)$ is C-extension complete and furthermore, by Def. 3.12 $m(GG)$ is C-conflict free, i.e., the conditions in Sec. 3.2 and Thm. 3.3 hold. However, for graph G in Fig. 3.8 we obtain that $G \in \mathscr{L}(C)$ but $G \notin \mathscr{L}(GG)$. Therefore the language inclusion does not hold although the conditions for domain completeness hold. This is due to the gap between the declarative nature of graph constraints and the constructive nature of graph grammars. Thus, the approach is only applicable to graph grammars with empty start graphs.

3.3.3 Graph Grammars with Deleting Productions

We assume graph grammar $GG = (S = \varnothing, P)$ typed over type graph TG with the empty start graph and productions $P = \{p_1, p_2, p_3\}$ as depicted in Fig. 3.9. Furthermore, we assume the set of domain constraints $C = \{c_1\colon P_1 \to C_1, c_2\colon P_2 \to C_2\}$ as depicted in Fig. 3.9. Note that $\mathscr{L}(GG)$ is C-extension complete by Def. 3.8 and $m(GG)$ seems to be C-conflict free by Def. 3.12 when neglecting the deleting elements in production p_3. Therefore, the conditions for domain completeness in Sec. 3.2 and Thm. 3.3 seem to hold. However, for graph G in Fig. 3.9 we obtain that $G \in \mathscr{L}(C)$ but $G \notin \mathscr{L}(GG)$. Therefore the language inclusion does not hold. This is due to the definition of marking rules which are only defined for non-deleting productions. The example shows that elements that are deleted via productions cannot simply be neglected in marking rules. Thus, the approach cannot be trivially extended to graph grammars with deleting productions and is only applicable to graph grammars with non-deleting productions.

3.3.4 Initial & General Satisfaction

Given type graph TG, constraints $C_G = \{c_3, c_4\}$ designated for general satisfaction and $C_I = \{c_1, c_2\}$ designated for initial satisfaction, and grammar $GG = (\varnothing, \{p\})$ where $c_1 = \exists(P_1 \to$

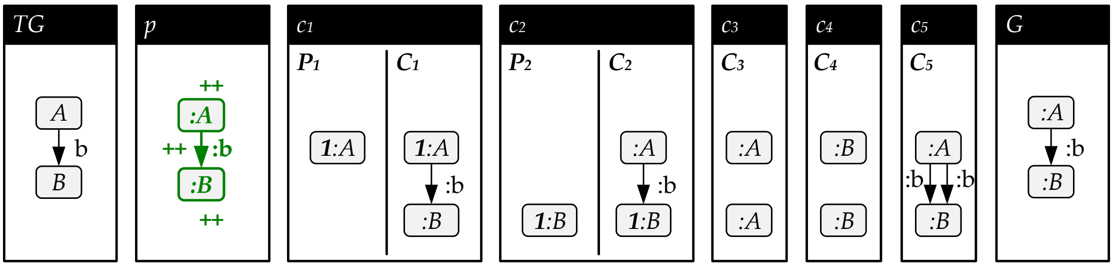


Figure 3.10: Limitation: Initial & General Satisfaction

$C_1,\mathbf{true}), c_2 = \exists(P_2 \to C_2,\mathbf{true}), c_3 = \neg\exists(\varnothing \to C_3,\mathbf{true}), c_4 = \neg\exists(\varnothing \to C_4,\mathbf{true})$ and $c_5 = \neg\exists(\varnothing \to C_5,\mathbf{true})$. Note that $\mathscr{L}(C) = \{G\} \subseteq \mathscr{L}(GG)$, i.e., domain completeness holds. However, constraints c_1 and c_2 are not used for C-extensions in Sec. 3.2 and Def. 3.5, since, both are designated for initial satisfaction. Therefore, C-extension completeness of $\mathscr{L}(GG)$ does not hold, since, effective atoms P_1, P_2 and G are not extended and cannot be created via grammar GG. The approach may be extended to also involve constraints that are designated for initial satisfaction in future work in order to handle situations from above appropriately.

3.3.5 Constraints & Application Conditions in $\mathscr{M}$-normal Form

Note that for domain completeness, the graphs in $\mathscr{L}(C)$ and $\mathscr{L}(GG)$ share the same "concrete" algebra up to isomorphism by general assumption of Sec. 3.2. For verifying domain completeness $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$ in Sec. 3.2 and Thm. 3.3, we assume that all constraints C and all application conditions in productions of graph grammar GG need to be in $\mathscr{M}$-normal form. This is due to the fact that domain completeness is verified only once on the level of the $DSIG$-term algebra for the data part and domain completeness for all concrete algebras can then be implied by Sec. 3.2 and Lemmas 3.14 and 3.15 but only when assuming the restriction to conditions in $\mathscr{M}$-normal form. Therefore, a verification for each case of concrete algebras is not necessary. Sec. 3.2 and Ex. 3.12 depicts the necessity for conditions in $\mathscr{M}$-normal form and the full proof of Sec. 3.2 and Thm. 3.3 reveals all details. In contrast to the presented approach, when performing the domain completeness verification directly on the concrete algebra for each case instead of a verification on the more abstract level of the term-algebra, then the restriction to conditions in $\mathscr{M}$-normal form may be loosened in future work. Note that the restriction to conditions in $\mathscr{M}$-normal form forbids the definition of conditions that identify elements on the data part. Therefore, constraints and application conditions of the following form cannot be expressed: "For two or more nodes that have a node attribute x each, it holds that all attributes x share the same attribute value". However, we are confident that conditions in $\mathscr{M}$-normal form have enough expressive power for real-world scenarios.

3.3.6 Termination Requires Upper Bound

According to Thm. 3.4, we have to define an upper bound for the size of graphs in order to ensure termination of the approach. In most cases, the verification terminates without restricting to an upper bound. However, when restricting to an upper bound we could also check for all graphs up to the upper bound which satisfy the constraints in C if they can be created via the rules in grammar GG for ensuring the validity of language inclusion $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$. On the other hand, the verification via C-extension completeness in Thm. 3.3 may be more efficient, since, not all graphs need to be checked but rather a small subset.

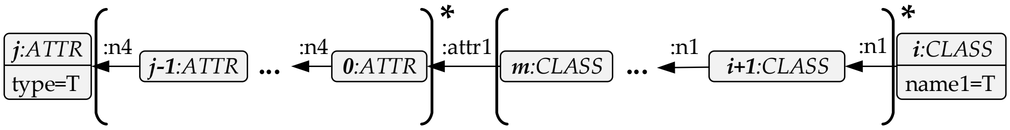


Figure 3.11: Language of Recursive Graph Schemata

3.4 Recursive Graph Constraints

We introduce recursive conditions as a special class of infinite conditions and show that it is sufficient to check the components of the recursions only to verify domain completeness of the whole system. Recursive conditions are infinite conditions that are given by the disjunction over the infinite set of graphs that are obtained from a start (premise) graph by repeating specific graph structures recursively. In particular, this allows the definition of conditions over regular paths in abstract syntax graphs of source code. Fig. 3.11 illustrates such a regular path expression defining that there is some path between an ATTRibute j of type T and the CLASS i with name T. Concretely, a) either j is directly connected to i via edge $:\mathtt{attr1}$, i.e., j is the first attribute of class i, or b) j is the jth attribute of class i and there are $j-1$ other attributes between both via $:\mathtt{n4}$ edges, or c) j is the first attribute of some other class m and there are $m-i$ other classes between i and m via $:\mathtt{n1}$ edges, or d) j is the jth attribute of class m and m is the $m-i$th class defined behind class i. Therefore, we want to describe an infinite set of graphs (paths) where the graph (path) structures that are enclosed by brackets with the Kleene star $*$ may be repeated recursively. This allows the definition of the constraint "Each attribute j of type T is the attribute of some class m and moreover, for each such j there is a class i with name T and either $i=m$ or i is defined somewhere before m." - condensed "For each attribute type that is used there is a corresponding class definition.". Such a constraint is rather an infinite constraint, i.e., the disjunction over all possible paths between j and i. Thus, the motivation behind the notion of recursive conditions (constraints) is a) to have a notation for defining infinite graph conditions with recursively repeating graph structures, b) and whose satisfiability is decidable, c) the ability to involve infinite graph constraints in verifying domain completeness while in general, the verification of domain completeness for infinite constraints does not terminate, and d) in particular, to use recursive graph constraints for specifying constraints over regular paths for the definition of abstract syntax graphs. We introduce the notion of recursive graph schemata for the definition of infinite sets of graphs that are obtained from a start graph by the (recursively repeated) restricted application of productions via pre-defined matches between the productions.

Definition 3.18 (Recursive Graph Schema) A *recursive graph schema* $GS=(GG,M,s_{GS},t_{GS})$ is given by

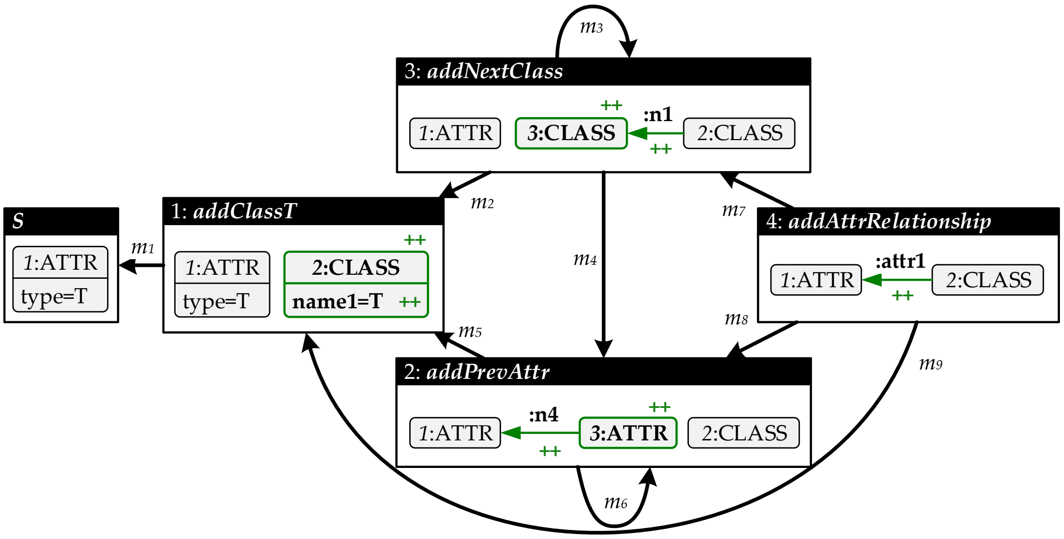
1. a graph grammar $GG=(S,P)$ with start graph S and a set P of productions $p=(L_p \xleftarrow{l_p} K_p \xrightarrow{r_p} R_p)$ with LHS L_p, gluing object K_p, RHS R_p and $l_p, r_p \in \mathcal{M}$,

2. a set M of matches $m \in \mathcal{M}$ where

   (a) $m\colon L_p \to S$ from the LHS L_p of some $p \in P$ to start graph S, or

   (b) $m\colon L_p \to R_{p'}$ from the LHS L_p of some $p \in P$ to the RHS $R_{p'}$ of some $p' \in P$, and

3. source function $s_{GS}\colon M \to P$ and target function $t_{GS}\colon M \to (S \cup P)$ such that for all $m \in M$, $s_{GS}(m\colon L_p \to A) = p$ and furthermore, $t_{GS}(m\colon A \to S) = S$ or $t_{GS}(m\colon A \to R_p) = p$ for some $p = (L_p \leftarrow K_p \to R_p) \in P$. $\triangle$



$$m_1 = (1 \mapsto 1); m_2 = (1 \mapsto 1, 2 \mapsto 2); m_3 = (1 \mapsto 1, 2 \mapsto 3); m_4 = (1 \mapsto 3, 2 \mapsto 2); m_5 = (1 \mapsto 1, 2 \mapsto 2); m_6 = (1 \mapsto 3, 2 \mapsto 2); m_7 = (1 \mapsto 1, 2 \mapsto 3); m_8 = (1 \mapsto 3, 2 \mapsto 2); m_9 = (1 \mapsto 1, 2 \mapsto 2)$$

$$s_{GS}(m_1) = 1 : \mathsf{addClassT}; s_{GS}(m_2) = s_{GS}(m_3) = s_{GS}(m_4) = 3 : \mathsf{addNextClass}; s_{GS}(m_5) = s_{GS}(m_6) = 2 : \mathsf{addPrevAttr}; s_{GS}(m_7) = s_{GS}(m_8) = s_{GS}(m_9) = 4 : \mathsf{addAttrRelationship};$$

$$t_{GS}(m_1) = S; t_{GS}(m_2) = t_{GS}(m_5) = t_{GS}(m_9) = 1 : \mathsf{addClassT}; t_{GS}(m_3) = t_{GS}(m_7) = 3 : \mathsf{addNextClass}; t_{GS}(m_4) = t_{GS}(m_6) = t_{GS}(m_8) = 2 : \mathsf{addPrevAttr};$$

Figure 3.12: Recursive Graph Schema

Remark 3.10 (Recursive Graph Schema) *Note that by function s_{GS}, each match is mapped to exactly one rule as source (analogously, with t_{GS} for the target). However, this does not restrict the expressiveness of recursive graph schemata. Two rules $p_1 : L \leftarrow K_1 \rightarrow R_1, p_2 : L \leftarrow K_2 \rightarrow R_2$ with the same LHS L and the same outgoing match $m : L \rightarrow A$ can be redefined by two rules $p_1, p_2' : L' \leftarrow K_2 \rightarrow R_2$ with distinct LHSs $L \neq L'$ and distinct matches $m, m' : L' \rightarrow A$ by renaming the elements (nodes and edges) of L resulting in L' and without changing the semantics up to isomorphism. This allows to define separate sources for m, m' with $s_{GS}(m) = p_1$ and $s_{GS}(m') = p_2'$.* △

Example 3.13 (Recursive Graph Schema) *Fig. 3.12 illustrates a recursive graph schema with start graph S, productions $\{1 \ldots 4\}$, matches $\{m_1 \ldots m_9\}$ and source (target) function s_{GS} (t_{GS}) which defines the infinite set of graphs that are presented in Fig. 3.11. From start graph S we obtain the graph with the corresponding* CLASS *of name T and edge* : attr1 *by applying rules* 1 *and* 4 *via matches m_1 and m_9 successively. Analogously, we may obtain bigger graphs beginning with S by (repeatedly) applying rules* 2 *and* 3 *via corresponding matches $m_2 \ldots m_6$ before applying rule* 4*. Note that cycles of match morphisms define graph structures that may be repeated recursively. The repeated application of rule* 3 *via match m_3 allows the repeated addition of classes whereas rule* 2 *together with match m_6 allows the repeated addition of attributes.* △

We define the precise semantics of a recursive graph schema *GS* by its graph language which is induced by terminating recursive transformation sequences over the schema starting at start graph *S*. A recursive transformation sequence is given by a path of matches *M* in *GS* and a corresponding sequence of recursive transformation steps where the match of each step is restricted by the co-match of the previous step. The sequence is terminating in the sense that there are no matches defined in M for extending the sequence. A path of matches M in GS is given by a sequence of matches $(m_i \in M)_{i \in I}$ with $s_{GS}(m_i) = t_{GS}(m_{i+1})$, for all $i \in I$.

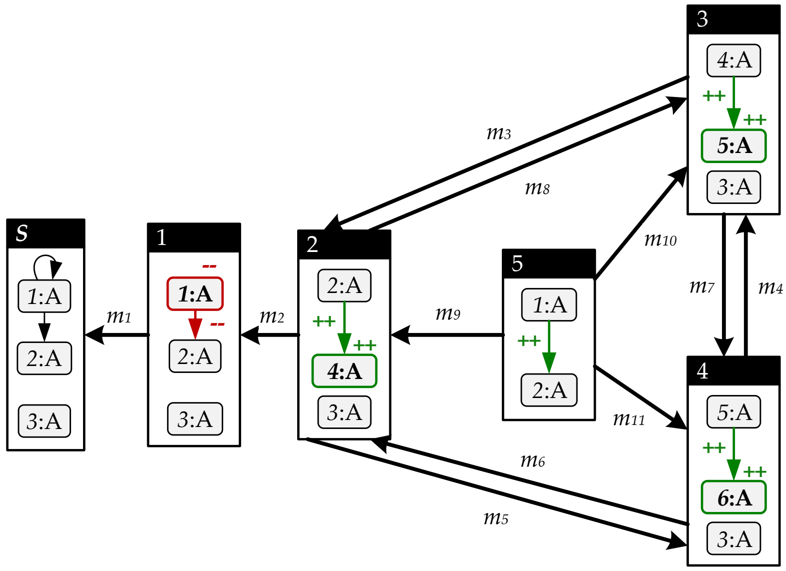
Definition 3.19 (Cyclic & Terminating Match-Path) Let $GS = ((S,P),M,s_{GS},t_{GS})$ be a recursive graph schema. A *match-path in GS* is a sequence of $n > 0$ matches $(m_i \in M)_{i \in I}, I = \{1 \dots n\}$ with $s_{GS}(m_i) = t_{GS}(m_{i+1})$, for all $i \in I$. A match-path with n matches is *terminating*, if there does not exist a match $m \in M$ with $t_{GS}(m) = s_{GS}(m_n)$. A match-path is *acyclic*, if for all $i, j \in I$ it is true that $i \neq j$ implies $s_{GS}(m_i) \neq s_{GS}(m_j)$. Otherwise, the match-path is *cyclic*. A match-path of n matches *starts (ends)* in A, if $t_{GS}(m_1) = A$ $(s_{GS}(m_n) = A)$. With $Paths_A(GS)$ we denote *the set of all match-paths in GS that start in A* and with $Paths_{A,B}(GS)$ we denote *the set of all match-paths in GS that start in A and end B.* A *match-cycle is some match-path path* $\in Paths_{A,A}(GS)$. A *match-cycle path* $\in Paths_{A,A}(GS)$ *is reachable from match-path* $(m_i)_{i \in \{1 \dots n\}}$, if there is some i with $s_{GS}(m_i) = A$ or $t_{GS}(m_i) = A$. △

Example 3.14 (Match-Path) *Given the recursive graph schema GS in Fig. 3.12. For example, $m = (m_1, m_5, m_6, m_8)$ is a match path with four matches - the source of each match coincides with the target of its successive match. Furthermore, m is terminating (there is no match with rule* 4 : *as target), m is cyclic ($s_{GS}(m_5) = s_{GS}(m_6)$) and $m \in Paths_{S,4:}(GS)$. The same path without match m_6 is acyclic. Match-path $(m_6) \in Paths_{2:,2:}(GS)$ is a match-cycle that is reachable from m.* △

Proposition 3.2 ((De)-Composition of Acyclic Match-Paths) *Given two match-paths $path_1 = (m_{1,i})_{i \in \{1 \dots n\}}$ and $path_2 = (m_{2,i})_{i \in \{1 \dots m\}}$ together with their merged match-path $path_3 = (m_{1,1} \dots m_{1,n}, m_{2,1} \dots m_{2,m})$. If $path_3$ is acyclic, then $path_1$ and $path_2$ are acyclic* (Decomposition). *However conversely, if $path_1$ and $path_2$ are acyclic, then $path_3$ is not necessarily acyclic but may be cyclic* (Composition). △

Proof. "Decomposition": Assume that $path_i$ is cyclic for $i \in \{1,2\}$. Then, the composition $path_3$ is cyclic by definition contradicting with the assumption that $path_3$ is acyclic. "Composition": The counterexample is as follows: Let $n = m = 1$ and $m_{1,1} = m_{2,1}$ be a reflexive match, i.e., $s_{GS}(m_{1,1}) = t_{GS}(m_{1,1}) = s_{GS}(m_{2,1}) = t_{GS}(m_{2,1})$. Match-paths $path_1$ and $path_2$ are acyclic, respectively, but the composition $path_3 = (m_{1,1}, m_{2,1})$ is cyclic. □

Definition 3.20 (Recursive Transformation) Let $GS = ((S,P),M,s_{GS},t_{GS})$ be a recursive graph schema. A *recursive transformation step* $G \xRightarrow{(p,m,n)}_{GS,n'} G'$ from G to G' via production $p = (L_p \xleftarrow{l_p} K_p \xrightarrow{r_p} R_p), p \in P$, match $m \colon L_p \to A, m \in M$ and morphism $n \colon A \to G, n \in \mathscr{M}$ is defined by a direct transformation $G \xRightarrow{(p,n \circ m)} G'$ via p and $n \circ m$ with pushouts $(1),(2)$ and co-match $n' \colon R_p \to G' \in \mathscr{M}$. A *recursive transformation sequence* w.r.t. a given match-path $path = (m_i \colon A_i \leftarrow L_{p_i})_{i \in \{1 \dots n\}}$ in GS, in short recursive transformation, is given by a sequence of n recursive transformation steps $(A'_{i-1} \xRightarrow{(s_{GS}(m_i),m_i,n'_{i-1})}_{GS,n'_i} A'_i)_{i \in \{1 \dots n\}}$ from A_1 to A'_n with $A'_0 = A_1$ and $n'_0 = id_{A_1}$, denoted by $A_1 \xRightarrow{path}_{GS,n'_n} A'_n$, in short $A_1 \xRightarrow{path}_{GS} A'_n$, $A_1 \overset{*}{\Rightarrow}_{GS,n'_n} A'_n$ or $A_1 \overset{*}{\Rightarrow}_{GS} A'_n$ if n'_n or $path$ is not relevant, as shown below right. A recursive transformation is *terminating, (a)cyclic or starts (ends) in A*, if the underlying match-path is terminating, (a)cyclic or starts (ends) in A.



$m_1 = (1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3); m_2 = (2 \mapsto 2, 3 \mapsto 3); m_3 = (3 \mapsto 3, 4 \mapsto 4); m_4 = (3 \mapsto 3, 5 \mapsto 5); m_5 = (3 \mapsto 3, 2 \mapsto 6); m_6 = (3 \mapsto 3, 5 \mapsto 4); m_7 = (3 \mapsto 3, 4 \mapsto 6); m_8 = (2 \mapsto 5, 3 \mapsto 3); m_9 = (1 \mapsto 4, 2 \mapsto 3); m_{10} = (1 \mapsto 5, 2 \mapsto 3); m_{11} = (1 \mapsto 6, 2 \mapsto 3)$
$t_{GS}(m_1) = S; s_{GS}(m_1) = t_{GS}(m_2) = 1; s_{GS}(m_2) = s_{GS}(m_5) = s_{GS}(m_8) = t_{GS}(m_3) = t_{GS}(m_6) = t_{GS}(m_9) = 2; s_{GS}(m_3) = s_{GS}(m_7) = t_{GS}(m_4) = t_{GS}(m_8) = t_{GS}(m_{10}) = 3; s_{GS}(m_4) = s_{GS}(m_6) = t_{GS}(m_5) = t_{GS}(m_7) = t_{GS}(m_{11}) = 4; s_{GS}(m_9) = s_{GS}(m_{10}) = s_{GS}(m_{11}) = 5$

Figure 3.13: Recursive Graph Schema with Interweaving Cyclic Match-Paths

$$\begin{tikzcd}
L_p & & K_p \arrow[ll, hook', "l_p"'] \arrow[rr, hook, "r_p"] \arrow[dd, hook, "n''"] & & R_p \arrow[dd, hook, "n'"] \\
A \arrow[from=1-1, hook, "m"'] \arrow[d, hook, "n"'] & (1) & & (2) & \\
G & & O \arrow[ll, hook', "l'_p"'] \arrow[rr, hook, "r'_p"] & & G'
\end{tikzcd}$$

$$\begin{tikzcd}
 & & L_{p_{i+1}} \arrow[d, hook, "m_{i+1}"] & K_{p_{i+1}} \arrow[l, hook', "l_{p_{i+1}}"'] \arrow[r, hook, "r_{p_{i+1}}"] \arrow[ddd, hook] & R_{p_{i+1}} \arrow[ddd, hook, "n'_{i+1}"] \\
L_{p_i} \arrow[d, hook, "m_i"'] & K_{p_i} \arrow[l, hook', "l_{p_i}"'] \arrow[r, hook, "r_{p_i}"] \arrow[dd, hook] & R_{p_i} \arrow[dd, hook, "n'_i"] & & \\
A_i \arrow[d, hook, "n'_{i-1}"'] & & & & \\
A'_{i-1} & O_i \arrow[l, hook', "l'_{p_i}"'] \arrow[r, hook, "r'_{p_i}"] & A'_i & O_{i+1} \arrow[l, hook', "l'_{p_{i+1}}"'] \arrow[r, hook, "r'_{p_{i+1}}"] & A'_{i+1}
\end{tikzcd}$$

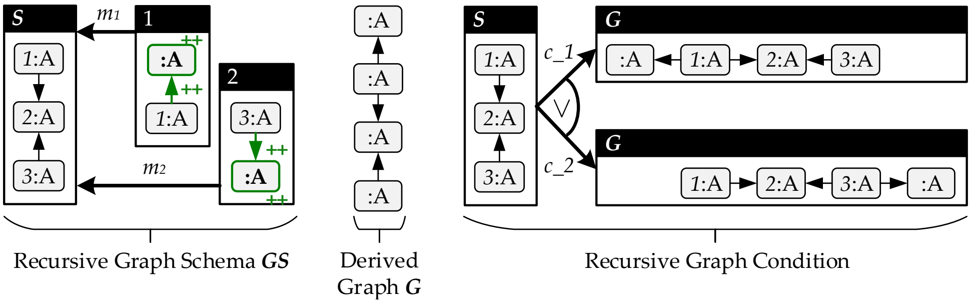
The *derived span* $der(t)$ *of a recursive transformation* $t \colon A_1 \overset{path}{\Longrightarrow}_{GS} A'_n$ is given by the derived span $der(t')$ of the underlying transformation $t' \colon (A'_{i-1} \xRightarrow{(s_{GS}(m_i), n'_{i-1} \circ m_i)} A'_i)_{i \in \{1 \ldots n\}}$. △

Remark 3.11 (Recursive Transformation) *Note that identical (empty) recursive transformations are not defined, since, their definition is based on match-paths which consists of* $n > 0$ *matches leading to a sequence of* $n > 0$ *recursive transformation steps for transformations. Furthermore, in* $\mathcal{M}$*-adhesive categories, the derived co-matches* $n' \colon R_p \to G'$ *in recursive transformation steps are guaranteed to be in* $\mathcal{M}$ *- thus, claiming* $n' \in \mathcal{M}$ *is not a restriction. For Def. 3.20,* (1) *is also a pullback by* $l_p \in \mathcal{M}$ *and pushouts along* $\mathcal{M}$*-morphisms are pullbacks with* $n', n'' \in \mathcal{M}$*, since,* $n \circ m \in \mathcal{M}$ *by* $\mathcal{M}$*-composition and furthermore,* $\mathcal{M}$*-morphsisms are closed under pushouts and pullbacks for pullback* (1) *and pushout* (2)*.* △

Finally, the semantics of a recursive graph schema is given by its graph language over the set of terminating recursive transformation sequences starting at start graph S. Note that for some recursive graph schemata there may not exist a terminating recursive transformation sequence or a recursive transformation sequence at all and therefore, its language may be empty. For example, the recursive graph schema in Fig. 3.13 without rule 5 and adjacent matches m_9 to m_{11} has no terminating match-path and therefore, no terminating recursive transformation sequence. Moreover, for the complete schema in Fig. 3.13 there exists an infinite set of terminating match-paths but all existing match-paths do not lead to a recursive transformation sequence due to violations of the gluing condition at the first match m_1 in each match-path. The deletion of node $1 : \mathsf{A}$ by applying rule 1 via match m_1 would result in a dangling reflexive edge on $1 : \mathsf{A}$. Thus, in general, for a match-path there may not exists a corresponding recursive transformation sequence due to a violation of the gluing condition.

Definition 3.21 (Graph Language of Recursive Graph Schemata) Let $GS = ((S,P),M,s_{GS},t_{GS})$ be a recursive graph schema. The *language* $\mathscr{L}(GS)$ *of* GS is given by $\mathscr{L}(GS) := R(GS)|_{\sim}$ with $R(GS)|_{\sim}$ being the quotient set of $R(GS)$ by $\sim$, $R(GS) := \{der(t) \mid t\colon S \overset{*}{\Longrightarrow}_{GS} G, t \text{ is terminating}, t \text{ starts in } S\}$ being the set of all derived spans that are derivable by terminating recursive transformation sequences starting at start graph S and $\sim$ being the equivalence relation on $R(GS)$ where $g\colon S \to G \sim g'\colon S \to G'$ if and only if $G \cong G'$ with isomorphism $i\colon G \to G'$ and $i \circ g = g'$. △

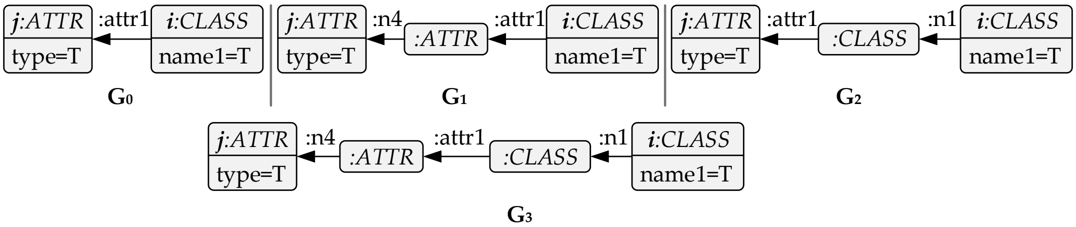
Remark 3.12 (Graph Language of Recursive Graph Schemata) *Note that $\sim$ is indeed an equivalence relation, i.e., 1. reflexivity: $g \sim g$ by isomorphism $i = id_G\colon G \to G$ with $id_G \circ g = g$, 2. symmetry: $g \sim g'$ if and only if $g' \sim g$ by inverse isomorphism $i^{-1}\colon G' \to G$ with $g = id_G \circ g = i^{-1} \circ i \circ g \overset{Def.~3.21}{=} i^{-1} \circ g'$, and 3. transitivity: $g \sim g'$ and $g' \sim g''\colon S \to G''$ implies $g \sim g''$ by isomorphism $i' \circ i\colon G \to G''$ with isomorphism $i'\colon G' \to G''$ and $i' \circ i \circ g \overset{Def.~3.21}{=} i' \circ g' \overset{Def.~3.21}{=} g''$. Moreover in $\mathscr{M}$-adhesive categories, $path = path'$ for $g = der(t), g' = der(t') \in R(GS)$ with $t\colon S \overset{path}{\Longrightarrow}_{GS} G, t'\colon S \overset{path'}{\Longrightarrow}_{GS} G'$ implies $G \cong G'$ with isomorphism $i\colon G \to G'$ and $i \circ g = g'$ (thus, $g \sim g'$) by the uniqueness of pushout complements and pushout objects. Furthermore, language $\mathscr{L}(GS)$ may contain the same graph G several times but each with different derived spans $der(t_i)\colon S \to G$.*



This is important in order to obtain recursive graph constraints over that language with the desired semantics. For example, from the recursive graph schema GS from above, graph G can be derived by two terminating recursive transformation sequences $t_1\colon S \overset{(m_1)}{\Longrightarrow}_{GS} G$ and $t_2\colon S \overset{(m_2)}{\Longrightarrow}_{GS} G$ starting at S, i.e., via the two match-paths (m_1) and (m_2), respectively. The mappings along morphisms $m_1, m_2, c_1 = der(t_1)$ and $c_2 = der(t_2)$ are expressed by identifications of the numbers for each node. Thus, $\mathscr{L}(GS) = \{[c_1],[c_2]\}$ is given by two equivalence classes which contain one of the two derived spans to graph G, respectively, resulting in a recursive graph constraint $(\exists(S \xrightarrow{c_1} G, \textbf{true})) \vee (\exists(S \xrightarrow{c_2} G, \textbf{true}))$ which disjoints both path-options. Therefore, a graph G' generally satisfies the constraint, if for all occurrences of pattern S in G' a) either node $: \mathsf{A}$ is attached to the top node $1 : \mathsf{A}$, or b) to the bottom node $3 : \mathsf{A}$. △

Remark 3.13 (Graph Language of Recursive Graph Schemata) *Note that the language of a recursive graph schema may be empty if no terminating recursive transformations do exist: 1. either due to a violation of the gluing condition (cf. Fig. 3.13), or 2. no terminating match-paths do exist, or 3. the set of productions P is empty. This is true since an empty set P implies that the set of matches M is also empty and furthermore, the language is defined over recursive transformation sequences w.r.t. match paths and empty match paths (empty recursive transformation sequences) are not defined explicitly (cf. Rem. 3.11).* △

Example 3.15 (Graph Language of Recursive Graph Schemata) *For the recursive graph schema in Ex. 3.13, Fig. 3.11 illustrates all graphs that can be derived by terminating recursive transformation sequences with starting at start graph S. This includes graph G_0 from below via recursive transformation $t_0\colon S \xRightarrow{(m_1)}_{GS} G_0$ and all graphs G_1, G_2, G_3 etc. that can be obtained from G_0 by arbitrarily adding a list of attribute (and class) nodes before j (and after i) recursively via corresponding recursive transformations.*



As the schema contains cyclic match-paths, its language is infinite containing equivalence class $[der(t_0)\colon S \to G_0]$ and the infinite set of equivalence classes $[der(t_i)\colon S \to G_i]$ of all other derived spans that can be obtained via terminating recursive transformations $t_i\colon S \overset{*}{\Rightarrow}_{GS} G_i$ starting at S. △

Based on the language of recursive graph schemata with start graph S and productions P, we define recursive (infinite) graph conditions. A recursive graph condition is defined by a disjunction over all derived spans of the language as mappings to the conclusion graphs with S being the premise graph. We assume a restriction of P to productions that are non-deleting only in order to obtain an induced morphism as derived span for each pair of (premise,conclusion) graphs instead of spans of morphisms (cf. Rem. 2.15).

Definition 3.22 (Recursive Graph Condition) Let $GS = ((S,P),M,s_{GS},t_{GS})$ be a recursive graph schema with non-deleting productions P and $\mathscr{L}(GS)$ be the graph language of GS. The *recursive graph condition c_{GS}* w.r.t. GS is defined by $c_{GS} = \vee_{[ac]\in\mathscr{L}(GS)}\exists(ac\colon S \to \_, \mathbf{true})$. △

Recursive graph conditions are in $\mathscr{M}$-normal form which is necessary to be used in the verification of domain completeness in Sec. 3.2 and Thm. 3.3.

Proposition 3.3 (Recursive Graph Conditions in $\mathscr{M}$-normal form) *In $\mathscr{M}$-adhesive categories,* each recursive graph condition is in $\mathscr{M}$-normal form. △

Proof. Technically, non-deleting productions $p = (L_p \xhookleftarrow{l_p} K_p \xrightarrow{r_p} R_p)$ are defined with $l_p = id_{L_p}$ resulting in recursive transformation steps in Def. 3.20 with $l'_p \in \mathscr{M}$, l'_p being an isomorphism, $l'^{-1}_p \in \mathscr{M}$ and l'^{-1}_p being its inverse morphism for $\mathscr{M}$-adhesive categories. This holds, since, the class $\mathscr{M}$ is closed under isomorphisms (therefore, $l_p \in \mathscr{M}$), iso- and $\mathscr{M}$-morphisms are closed under pushouts (therefore, $l'_p \in \mathscr{M}$ and l'_p is isomorphism) and the inverse morphism l'^{-1}_p is in $\mathscr{M}$ by $\mathscr{M}$-decomposition of $l'_p \circ l'^{-1}_p = id_O \in \mathscr{M}$. Furthermore, $r'_p \in \mathscr{M}$, since, $r_p \in$

$\mathscr{M}$ by the definition of productions and $\mathscr{M}$-morphisms are closed under pushouts. Thus, the induced morphisms ac in Def. 3.22 are in $\mathscr{M}$ by $\mathscr{M}$-composition of $r'_p \circ l'^{-1}_p$, i.e., recursive graph conditions are in $\mathscr{M}$-normal form. □

Note that recursive graph conditions are standard nested conditions and therefore, the standard logical operations for forming bigger conditions can be applied, e.g., negation, con- and disjunction with other conditions. Especially the possibility to negate recursive conditions not only allows the definition of graph constraints of the form "For all S there exists some regular path" and "there exists S with some regular path" but also "For all S there does not exist some regular path" and "there exists S without some regular path".

Remark 3.14 (Conjunctive Recursive Graph Conditions) *Note that recursive graph conditions are explicitly defined by a disjunction over the conclusions that is infinite for infinite sets of conclusions. A corresponding notion where the disjunction is replaced by a conjunction would disagree with the intrinsic idea of having an option of (infinite) conclusions. Moreover, a conjunction over an infinite set of recursively growing conclusions as constraint can only be satisfied by infinite graphs.* △

Example 3.16 (Recursive Graph Constraint) *For the recursive graph schema in Ex. 3.13, the corresponding infinite recursive graph constraint is generally satisfied by a graph G, if in G for all attributes A of type T there is a class C_1 with name T defined, and A is the attribute of some class C_2 and either $C_1 = C_2$ or C_1 is defined before C_2. The membership of an attribute to a class is expressed by paths of* : `attr1` *and* : `n4` *edges between both. The fact that a class is defined before some other class is expressed by paths of* : `n1` *edges between both.* △

In [BDK+12], the decidability of reachability and coverability in various types of graph transformation systems with different restrictions were investigated. In the following, we investigate the decidability of partial reachability in graph grammars and recursive graph schemata in order to conclude over the decidability of the satisfaction of recursive graph conditions in Thm. 3.5. In contrast to reachability, given a graph G, partial reachability of G requires that there is at least one transformation sequence to some G' such that there is an $\mathscr{M}$-morphism from G' to G instead of an isomorphism between both. Conversely, coverability requires that there is at least one transformation sequence to some G' such that there is an $\mathscr{M}$-morphism from G to G'. Thus, partial reachability is the inverse of coverability w.r.t. reachability.

While for finite-state graph transformation systems, the partial reachability problem is obviously decidable by iterating over all states and check partial reachability for each, it is different for infinite-state graph transformation systems (states are graphs and transitions between states are transformation steps). We use infinite-state graph transformation systems for specifying infinite graph languages of recursive graph patterns. While graph grammars with deleting rules can be used for specifying such languages where the start graph is the base case of the recursion and further graph patterns are obtained by recursively deleting and adding elements, the partial reachability problem is undecidable for general graph grammars by a reduction from the halting problem over turing machines (cf. Prop. 3.4 and Item 2). However, for recursive graph schemata (with deleting rules), it turns out to be decidable by their monotone nature which is given by the restrictions for matches, since, 1. matches are pre-defined, and 2. each recursive transformation step in a recursive transformation uses the co-match of the previous step to further restrict the match (cf. Prop. 3.4 and Item 3). This leads to the decidability of the satisfaction of recursive graph conditions in Thm. 3.5.

Definition 3.23 (Partial Reachability Problem) Let G be a graph, $GG = (S,P)$ be a graph grammar with start graph S and a set of productions P and $GS = (GG, M, s_{GS}, t_{GS})$ be a recursive graph schema. The *partial reachability problem* for graph grammars (recursive graph schemata) is defined as follow: Is there a (terminating recursive) transformation sequence $S \overset{*}{\Rightarrow} G'$ ($S \overset{*}{\Rightarrow}_{GS} G'$) that starts in S to some graph G' via productions P (in GS) such that there exists an $\mathscr{M}$-morphism $m\colon G' \to G, m \in \mathscr{M}$? △

Proposition 3.4 (Decidability of Partial Reachability) *Let G be a graph, $GG = (S,P)$ be a graph grammar with start graph S and a set of productions P and $GS = (GG, M, s_{GS}, t_{GS})$ be a recursive graph schema.*

1. *For grammars GG with non-deleting productions P only and finite graphs G, the (partial) reachability problem is decidable in $\mathscr{M}$-adhesive categories.*

2. *In general and in particular for grammars GG with finite sets of productions P and finite graphs G in the $\mathscr{M}$-adhesive category $(\mathbf{Graphs}_{TG}, \mathscr{M})$ of typed graphs with(out) application conditions (and $\mathscr{M}$-matches), the (partial) reachability problem is undecidable.*

3. *In $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, for recursive graph schemata GS with a finite set of matches M and finite graphs G, the (partial) reachability problem is decidable.* △

Proof. The proof is presented in appendix A.12. □

Theorem 3.5 (Decidability of Satisfaction of Recursive Graph Conditions) *Let $GS = ((S,P), M, s_{GS}, t_{GS})$ be a recursive graph schema with a finite set of matches M, $p\colon S \to G$ be a morphism (G be a finite graph) and c_{GS} be the recursive graph condition w.r.t. GS. Then,* the problem whether p (G) satisfies c_{GS} is decidable in $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$. △

Proof. By Defs. 2.12 and 3.22, $p \models c_{GS} = \vee_{[der(t)] \in \mathscr{L}(GS)} \exists (der(t)\colon S \to \_, \mathbf{true})$ if and only if $\exists [der(t)\colon S \to G'] \in \mathscr{L}(GS). p \models \exists (der(t), \mathbf{true}) \Leftrightarrow \exists$ terminating recursive transformation t in GS that starts in S with $der(t)\colon S \to G'$ and $p \models \exists (der(t), \mathbf{true}) \Leftrightarrow \exists$ terminating recursive transformation $t\colon S \overset{*}{\Rightarrow}_{GS} G'$ in GS that starts in S such that there exists $q\colon G' \to G \in \mathscr{M}$ with $q \circ der(t) = p$ which is decidable by Prop. 3.4 and Item 3 by checking the commuting property for all existing $q \in \mathscr{M}$ which also terminates with the procedure from the proof of Prop. 3.4 and Item 3 for finite G. For recursive graph constraints c_{GS} over S, listing all occurrences of S in G (all $p\colon S \to G \in \mathscr{M}$) also terminates and results in a finite set for finite G. For each occurrence of S in G we can proceed with c_{GS} as given above. □

Recursive structures in recursive conditions are expressed by cycles of matches M in the underlying recursive graph schema. Thus, a recursive graph schema with cycles in M may lead to an infinite recursive graph constraint which is constructed step-wise by pumping the underlying condition via the repeated iteration of cycles. The verification procedure of domain completeness terminates for finite graph constraints only. Therefore, we present two techniques for deriving finite constraints from infinite recursive graph constraints by making them tighter or weaker. The idea behind tightened recursive graph conditions is to omit all conclusion graphs from an infinite recursive condition c_{GS} that exceed a specific upper bound. The upper bound is defined by a dedicated graph G_u and the tightened condition of c_{GS} is formed over all conclusion graphs of c_{GS} with inclusions to G_u only. Therefore, the set of graphs satisfying a set of tightened constraints is a subset of the set of graphs satisfying the set of the corresponding originial infinite constraints but not necessarily vice versa. However, it holds that a successful verification of domain completeness w.r.t. the tightened constraints and a given TGG implies that all graphs up to the upper bound satisfying the original infinite constraints can be completely transformed via the TGG. In contrast to that, the idea behind weakened recursive graph conditions is to list for each acyclic, terminating recursive transformation sequence $S \xRightarrow{path}_{GS} G$ over GS and for all match-cycles in GS that are reachable from match-path *path* all relevant overlappings of a single iteration of each cycle and G as conclusion graphs. Therefore, the derived conditions cover those graph structures only that may occur when iterating each cycle the last time by omitting those structures that are created by the iterations in between. For positive recursive constraints c_{GS}, the set of graphs satisfying the corresponding weakened constraint is a superset of the set of graphs satisfying c_{GS}. Thus, a successful verification of domain completeness w.r.t. weakened constraints and a given TGG implies domain completeness w.r.t. the original (infinite) constraints c_{GS} and TGG which implies furthermore that all graphs satisfying constraints c_{GS} can be completely transformed via the TGG.

Definition 3.24 (Tightened Graph Language of Recursive Graph Schemata) Let $GS = ((S,P),M,s_{GS},t_{GS})$ be a recursive graph schema. Let $\mathscr{L}(GS)$ be the graph language of GS and graph G_u be an upper bound. The *tightened language* $\mathscr{L}_t(GS,G_u) \subseteq \mathscr{L}(GS)$ of GS and w.r.t. G_u is given by $\mathscr{L}_t(GS,G_u) := \{[ac] \mid [ac\colon S \to G] \in \mathscr{L}(GS), \exists i\colon G \to G_u \in \mathscr{M}\}$. △

Before defining weakened conditions, we define the notions of $\mathscr{M}$-decomposition, initial $\mathscr{M}$-subobject and weakened language of recursive graph schemata at first. An $\mathscr{M}$-decomposition of a morphism $tr\colon A \to B$ is a decomposition of tr into two morphisms $tr_1\colon A \to C, tr_2\colon C \to B$ where both tr_1 and tr_2 are in $\mathscr{M}$. For example in $(\mathbf{Graphs},\mathscr{M})$, $\mathscr{M}$-decompositions are a restriction of general decompositions of morphisms in the sense that graph A is guaranteed to be a sub-graph of C and C is a sub-graph of B whereas in general decompositions C may be "bigger" than B with an non-injective morphism tr_2. Similar to the concept of an $\mathscr{M}$-initial object which is considered as the "smallest" $\mathscr{M}$-subobject of all objects of a category, we introduce the weaker notion of an initial $\mathscr{M}$-subobject which is only the "smallest" $\mathscr{M}$-subobject concerning one object of a category, respectively. While a category may not have an $\mathscr{M}$-initial object, it may have an initial $\mathscr{M}$-subobject for each object.

Definition 3.25 ($\mathscr{M}$-decomposition of Morphisms) *An $\mathscr{M}$-decomposition d of an $\mathscr{M}$-morphism $tr\colon L \to R$*, in short *tr-$\mathscr{M}$*-decomposition, consists of two $\mathscr{M}$-morphisms $d = (L \xrightarrow{tr_1} L' \xrightarrow{tr_2} R)$ with $tr_2 \circ tr_1 = tr$ and $tr_1, tr_2 \in \mathscr{M}$. △

Definition 3.26 (Initial $\mathscr{M}$-subobject) Given an $\mathscr{M}$-adhesive category $(\mathbf{C},\mathscr{M})$ and an object $G \in \mathbf{C}$. Then, *$\mathscr{M}$-subobject $[i_G\colon I_G \to G \in \mathscr{M}]$ of G is initial*, if for each $\mathscr{M}$-subobject $[a\colon A \to G \in \mathscr{M}]$ of G there exists an (unique) $\mathscr{M}$-morphism $i_A\colon I_G \to A \in \mathscr{M}$ with $a \circ i_A = i_G$. An $\mathscr{M}$-adhesive category $(\mathbf{C},\mathscr{M})$ has initial $\mathscr{M}$-subobjects, if for each object $G \in \mathbf{C}$ there exists an initial $\mathscr{M}$-subobject. △

Remark 3.15 (Initial $\mathscr{M}$-subobject) *Note that the uniqueness of morphism i_A is optional as it follows directly from $a \in \mathscr{M}$ with $\mathscr{M}$ being a class of monomorphisms, since, $i_{A,1}, i_{A,2} \in \mathscr{M}$ with $a \circ i_{A,1} = i_G = a \circ i_{A,2}$ implies that $i_{A,1} = i_{A,2}$.* △

Initial $\mathscr{M}$-subobjects can be constructed by the extremal $\mathscr{E}$-$\mathscr{M}$-factorisation of initial morphisms. Furthermore, initial $\mathscr{M}$-subobjects are unique up to isomorphism.

Proposition 3.5 (Construction & Uniqueness of Initial $\mathscr{M}$-subobjects) *Given an $\mathscr{M}$-adhesive category $(\mathbf{C},\mathscr{M})$ with an extremal $\mathscr{E}$-$\mathscr{M}$-factorisation and initial object I. Then,* the initial

$\mathcal{M}$-subobject $[i_{G,2}\colon I_G \to G \in \mathcal{M}]$ for an object $G \in \mathbf{C}$ can be constructed *by the extremal $\mathcal{E}$-$\mathcal{M}$-factorisation $i_{G,2} \circ i_{G,1} = i_G$ of the initial morphism $i_G\colon I \to G$ with $i_{G,1}\colon I \to I_G \in \mathcal{E}, i_{G,2}\colon I_G \to G \in \mathcal{M}$. Furthermore, in $\mathcal{M}$-adhesive categories,* initial $\mathcal{M}$-subobjects are unique up to isomorphism. △

Proof. We have to show that $\mathcal{M}$-subobject $[i_{G,2}\colon I_G \to G \in \mathcal{M}]$ of G is initial.

$$\begin{tikzcd}
 & & I \arrow[lld, bend right, "i_A"'] \arrow[ld, dotted, "i_O"'] \arrow[d, "i_{G,1}"] \arrow[dd, bend left=60, "i_G"] \\
A \arrow[rrd, hook, bend right, "a"'] \arrow[ru, phantom, "(=)"] & O \arrow[l, hook', "o_1"'] \arrow[r, hook, bend left, "o_2"] \arrow[ru, phantom, "(=)"] \arrow[d, phantom, "(1)"] & I_G \arrow[l, dotted, bend left, "o_2^{-1}"] \arrow[d, hook, "i_{G,2}"] \arrow[r, phantom, "(=)"] & {} \\
 & {} & G
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \overline{O} \arrow[ld, "o_1"'] \arrow[rd, "o_2"] & \\
\overline{I}_{G,1} \arrow[ru, dotted, hook, bend left, "o_1'"] \arrow[rd, hook, "\bar{i}_{G,1}"'] & (2) & \overline{I}_{G,2} \arrow[lu, dotted, hook', bend right, "o_2'"'] \arrow[ld, hook', "\bar{i}_{G,2}"] \\
 & \overline{G} &
\end{tikzcd}$$

Let $[a\colon A \to G \in \mathcal{M}]$ be an $\mathcal{M}$-subobject of G and $i_A\colon I \to A$ be the initial morphism to A. We construct pullback (1) (o_1, o_2) over morphisms $(a, i_{G,2})$. By $\mathcal{M}$-morphisms are closed under pullbacks, $o_1, o_2 \in \mathcal{M}$. By the uniqueness of the initial morphism, $i_G = i_{G,2} \circ i_{G,1} = a \circ i_A$. Thus, by the universal property of pullbacks, there exists a unique morphism $i_O\colon I \to O$ with $o_1 \circ i_O = i_A$ and $o_2 \circ i_O = i_{G,1}$. By the definition of class $\mathcal{E}$ with $i_{G,1} \in \mathcal{E}$ and $o_2 \in \mathcal{M}$, o_2 is an isomorphism with inverse isomorphism $o_2^{-1} \in \mathcal{M}$ and $o_2 \circ o_2^{-1} = id_{I_G}$ $(*^1)$, since, class $\mathcal{M}$ is closed under isomorphisms. By $\mathcal{M}$-composition, $o_1 \circ o_2^{-1} \in \mathcal{M}$, $o_1 \circ o_2^{-1}$ is unique by Rem. 3.15 and furthermore, $a \circ o_1 \circ o_2^{-1} \overset{(1)}{=} i_{G,2} \circ o_2 \circ o_2^{-1} \overset{(*^1)}{=} i_{G,2} \circ id_{I_G} = i_{G,2}$. The resulting initial $\mathcal{M}$-subobjects of the presented construction are unique, since, extremal $\mathcal{E}$-$\mathcal{M}$-factorisations are unique up to isomorphism in the given $\mathcal{M}$-adhesive category. In general, apart from the construction, the uniqueness of initial $\mathcal{M}$-subobjects in $\mathcal{M}$-adhesive categories is shown as follows. Given two initial $\mathcal{M}$-subobjects $[\bar{i}_{G,1}\colon \overline{I}_{G,1} \to \overline{G} \in \mathcal{M}]$ and $[\bar{i}_{G,2}\colon \overline{I}_{G,2} \to \overline{G} \in \mathcal{M}]$ of $\overline{G}$. We construct pullback (2) (o_1, o_2) over $(\bar{i}_{G,1}, \bar{i}_{G,2})$ with $o_1, o_2 \in \mathcal{M}$, since, $\mathcal{M}$-morphisms are closed under pullbacks and furthermore, $\bar{i}_{G,1} \circ o_1, \bar{i}_{G,2} \circ o_2 \in \mathcal{M}$ by $\mathcal{M}$-composition. By Def. 3.26, there exists $o_2'\colon \overline{I}_{G,2} \to \overline{O} \in \mathcal{M}$ with $\bar{i}_{G,1} \circ o_1 \circ o_2' = \bar{i}_{G,2}$ $(*^1)$. Analogously, there exists $o_1'\colon \overline{I}_{G,1} \to \overline{O} \in \mathcal{M}$ with $\bar{i}_{G,1} \circ id_{\overline{I}_{G,1}} = \bar{i}_{G,1} = \bar{i}_{G,2} \circ o_2 \circ o_1' \overset{(*^1)}{=} \bar{i}_{G,1} \circ o_1 \circ o_2' \circ o_2 \circ o_1'$. By $\bar{i}_{G,1} \in \mathcal{M}$ and $\mathcal{M}$ being a class of monomorphisms, $o_1 \circ o_2' \circ o_2 \circ o_1' = id_{\overline{I}_{G,1}}$. Conversely, $\bar{i}_{G,2} \circ id_{\overline{I}_{G,2}} = \bar{i}_{G,2} \overset{(*^1)}{=} \bar{i}_{G,1} \circ o_1 \circ o_2' = \bar{i}_{G,2} \circ o_2 \circ o_1' \circ o_1 \circ o_2'$. By $\bar{i}_{G,2} \in \mathcal{M}$ being a monomorphism, $o_2 \circ o_1' \circ o_1 \circ o_2' = id_{\overline{I}_{G,2}}$. Thus, $o_2 \circ o_1'$ is an isomorphism with commuting (2). □

Example 3.17 (Initial $\mathcal{M}$-subobject) *The category $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ has $(\varnothing, T_{DSIG})$ as initial object with $\varnothing$ being the empty graph except for the data nodes and T_{DSIG} being the term algebra of data signature DSIG. The initial $\mathcal{M}$-subobject of an object $(G,D) \in \mathbf{AGraphs}_{ATGI}$ is constructed by the extremal $\mathcal{E}$-$\mathcal{M}$-factorisation of the initial morphism $i_G\colon (\varnothing, T_{DSIG}) \to (G,D)$. Thus, the initial $\mathcal{M}$-subobject of (G,D) with DSIG-algebra D is $[i_G'\colon (\varnothing, D) \to (G,D) \in \mathcal{M}]$ with $\varnothing$ being the empty graph except for the data nodes and $i_G' = (i_{G,G}', i_{G,D}')$ being the empty morphism $i_{G,G}'\colon \varnothing \to G$ on the graph part and identity $i_{G,D}' = id_D\colon D \to D$ on the data part. Note that, $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ has an initial object but no $\mathcal{M}$-initial object, since, $\mathcal{M}$-morphisms are isomorphisms on the data part and this does not hold for all initial morphisms in $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$.* △

The result in Prop. 3.6 we use to prove the well-definedness of the construction of weakened languages of recursive graph schemata in Prop. 3.7.

Proposition 3.6 (Initial $\mathcal{M}$-subobjects along $\mathcal{M}$-Morphisms) *In an $\mathcal{M}$-adhesive category with initial $\mathcal{M}$-subobjects, let $a\colon A \to B \in \mathcal{M}$ be an $\mathcal{M}$-morphism and $[i_A\colon I_A \to A \in \mathcal{M}]$ be the initial $\mathcal{M}$-subobject of A. Then,* $[a \circ i_A\colon I_A \to B]$ is the initial $\mathcal{M}$-subobject of B. △

Proof. By $\mathcal{M}$-composition $a \circ i_A \in \mathcal{M}$, i.e., $[a \circ i_A \in \mathcal{M}]$ is an $\mathcal{M}$-subobject of B. Let $[i_B\colon I_B \to B \in \mathcal{M}]$ be the initial $\mathcal{M}$-subobject of B. Then, there exists an unique morphism $i_{I_A}\colon I_B \to I_A \in \mathcal{M}$ with $a \circ i_A \circ i_{I_A} = i_B$ $(*^1)$. By $\mathcal{M}$-composition, $[i_A \circ i_{I_A} \in \mathcal{M}]$ is an $\mathcal{M}$-subobject of A. Thus from i_A being the initial $\mathcal{M}$-subobject of A by assumption, there is $i_{I_B}\colon I_A \to I_B \in \mathcal{M}$ with $i_A \circ i_{I_A} \circ i_{I_B} = i_A = i_A \circ id_{I_A} \overset{i_A \text{ is Mono}}{\Rightarrow} i_{I_A} \circ i_{I_B} = id_{I_A}$ $(*^2)$. Moreover, $i_B \circ i_{I_B} \overset{(*^1)}{=} a \circ i_A \circ i_{I_A} \circ i_{I_B} \overset{(*^2)}{=} a \circ i_A$ $\Rightarrow i_B \circ i_{I_B} \circ i_{I_A} = a \circ i_A \circ i_{I_A} \overset{(*^1)}{=} i_B = i_B \circ id_{I_B} \overset{i_B \text{ is Mono}}{\Rightarrow} i_{I_B} \circ i_{I_A} = id_{I_B}$ $(*^3)$.

$$\begin{tikzcd}
I_B \arrow[d, hook, dotted, bend left, "i_{I_A}"] \arrow[rrd, hook, "i_B"] & & \\
I_A \arrow[u, hook, dotted, bend left, "i_{I_B}"] \arrow[r, hook, "i_A"] & A \arrow[u, phantom, "(1)"] \arrow[r, hook, "a"] & B
\end{tikzcd}$$

By $(*^1), (*^2)$ and $(*^3)$, i_{I_A} is an isomorphism with (1) commutes and therefore, $[i_B] = [a \circ i_A]$. □

For the construction of weakened graph languages, we restrict productions to be non-deleting in order to avoid conflicts with violations of the gluing condition and introduce the equivalence of match-cycles (cf. Rem. 3.16). We are confident that the construction can be extended to general productions by integrating gluing condition checks.

Definition 3.27 (Equivalence of Match-Cycles) Let $\gg((m_i)_{i\in\{1\ldots n\}}) := \{(m_i, m_{i+1}) \mid 1 \leq i < n\} \cup \{(m_n, m_1)\}$ be an order over the matches of a given match-cycle $(m_i)_{i\in\{1\ldots n\}}$. *Two match-cycles $path_1$ and $path_2$ are equal up to shifting of matches* if and only if $\gg(path_1) = \gg(path_2)$. Let GS be a recursive graph schema and $path$ be a match-cycle in GS. With $\gg_{GS}(path) := \{path' \mid path' \text{ is match-cycle in } GS, \gg(path) = \gg(path')\}$ we denote the set of all match-cycles in GS that are equal to $path$ up to shifting of matches. For match-paths $path$ in GS that are no match-cycles, we define $\gg_{GS}(path) := \varnothing$. Let $Paths_{\_,\_}(GS)$ be the set of all acyclic match-cycles in GS and $\sim$ be the equivalence relation on $Paths_{\_,\_}(GS)$ where $path_1 \sim path_2$ if and only if $\gg_{GS}(path_1) = \gg_{GS}(path_2)$. With $Paths_{\_,\_}(GS)_{|\sim}$ we denote the equivalence classes of all acyclic match-cycles in GS that are equal up to shifting of matches. △

Example 3.18 (Equivalence of Match-Cycles) *For the match-cycle $path = (m_7, m_8, m_6)$ in Fig. 3.13, $\gg_{GS}(path) = \{(m_7, m_8, m_6), (m_6, m_7, m_8), (m_8, m_6, m_7)\}$.* △

Definition 3.28 (Weakened Graph Language of Recursive Graph Schemata) Let $GS = ((S,P), M, s_{GS}, t_{GS})$ be a recursive graph schema with non-deleting productions P. The *weakened language $\mathcal{L}_w(GS) := \underline{\mathcal{L}_w}(GS)_{|\sim}$ of GS is given by the quotient set of $\underline{\mathcal{L}_w}(GS)$ by* $\sim$ with $\sim$ being the equivalence relation from Def. 3.21 and projection $\underline{\mathcal{L}_w}(GS) := \{ac \mid (ac, n) \in \cup_{m \in \underline{Paths_S}(GS)}(\underline{\mathcal{L}_w}(m, 1, id_S, id_S, 1))\}$ with $\underline{Paths_S}(GS) \subseteq Paths_S(GS)$ being the set of all acyclic and terminating match-paths in GS that start in S with construction $\underline{\mathcal{L}_w}$ as given inductively below. △

Construction

$$\underline{\mathscr{L}}_w((m_i)_{i\in\{1\dots j\}}, i, ac, n, k) = \begin{cases} \cup_{(t,n')\in\mathscr{B}}(\underline{\mathscr{L}}_w((m_i)_{i\in\{1\dots j\}}, i+1, & \text{, for } ac\colon \_ \to A \text{ and} \\ der(t)\circ ac, n', 1)) \text{ with } \mathscr{B} = & \text{if } i \leq j \text{ and} \\ \{(t,n') \mid t\colon A \xRightarrow{(s_{GS}(m_i),m_i,n)}_{GS,n'} B\} & (\mathscr{P} = \varnothing \text{ or } k=2) \\ \cup_{(ac',n')\in\mathscr{C}}(\underline{\mathscr{L}}_w((m_i)_{i\in\{1\dots j\}}, i, & \text{, for } ac\colon \_ \to A \text{ and} \\ ac', n', 2)) \text{ with } \mathscr{C} = \{(ac,n)\}\cup & \text{if } i \leq j \text{ and} \\ \cup_{path\in\mathscr{P}}(A \oplus B_{path}) \text{ where} & \mathscr{P} \neq \varnothing \\ B_{path} = \underline{\mathscr{L}}_w(path, 1, id_R, id_R, 2) \text{ for} & \\ path = (m'_k)_{k\in\{1\dots l\}} \text{ and } m'_1\colon L \to R & \\ \{(ac,n)\} & \text{, otherwise} \end{cases}$$

with switch $k \in \{1,2\}$, $\overline{\mathscr{P}} = Paths_{t_{GS}(m_i),t_{GS}(m_i)}(GS)\setminus \gg_{GS} ((m_i)_{i\in\{1\dots j\}})$ being the set of all match-cycles in GS that start and end in $\underline{t_{GS}(m_i)}$ except all paths that are equal up to shifting to the path that is currently being handled, $\mathscr{P} \subseteq \overline{\mathscr{P}}$ being all paths in $\overline{\mathscr{P}}$ that are acyclic and $\oplus$ defined as follows: $A \oplus B_{path} := \cup_{d\in\mathscr{D}}(\{(a'_2 \circ ac, a'_1 \circ \overline{n}') \mid cond\})$ where in contrast to $(ac,n) \in \mathscr{C}$ which represents the case with no iterations of cycles via match-path $path \in \mathscr{P}$, $A \oplus B_{path}$ represents arbitrary iterations by forming all relevant overlappings of the result of a single iteration $it \in B_{path}$ and A where:

$$\begin{tikzcd}
 & R \arrow[d, hook, dotted, "\overline{n}'"] \arrow[dd, bend left=60, "\overline{n}"] \arrow[dd, phantom, shift left=25, "(=)"] \\
L \arrow[r, hook, dotted, "\overline{ac}'"] \arrow[d, hook, "m'_1"] \arrow[dr, phantom, "(PO)"] & L' \arrow[d, hook, dotted, "\overline{m}'_1"] \\
R \arrow[r, hook, "\overline{ac}"] & R'
\end{tikzcd}
\qquad
\begin{tikzcd}
I_L \arrow[r, hook, "i_{L,1}"] & I'_L \arrow[r, hook, "i_{L,2}"] \arrow[dr, bend right, "m"] & L \arrow[r, hook, dotted, "o"] \arrow[rr, bend left, "\overline{ac}'"] \arrow[d, hook, "m'_1"] \arrow[dr, phantom, "(2)"] & O \arrow[dd, hook, dotted, "a''"] \arrow[ddr, phantom, "(5)"] & L' \arrow[ddd, hook, dotted, "a'_1"] \\
 & & R \arrow[ur, hook, dotted, "o'"] \arrow[dr, hook, dotted, "n'"] \arrow[d, hook, "id_R"] \arrow[r, phantom, "(3)"] \arrow[dr, phantom, shift right=15, "(4)"] & {} & \\
 & & R \arrow[r, hook, dotted, "der(t)"] \arrow[d, hook, "n"] & R'' & \\
 & & A \arrow[rr, hook, dotted, bend right, "a'_2"] & & A'
\end{tikzcd}$$

1. $\mathscr{D} = \{d = (I_L \xrightarrow{i_{L,1}} I'_L \xrightarrow{i_{L,2}} L) \mid d$ is i_L-$\mathscr{M}$-decomposition for initial $\mathscr{M}$-subobject $[i_L\colon I_L \to L \in \mathscr{M}]$ of $L\}$, and

2. *cond*:

   (a) $it = (\overline{ac}\colon R \to R', \overline{n}\colon R \to R') \in B_{path}$,

   (b) $(a'_1\colon L' \to A', a'_2\colon A \to A')$ is pushout over $(\overline{ac}' \circ i_{L,2}, n \circ m)$,

   (c) $(\overline{ac}' \in \mathscr{M}, \overline{m}'_1 \in \mathscr{M})$ is pushout complement over $(m'_1, \overline{ac})$ resulting in pushout (PO) with induced morphism $\overline{n}' \in \mathscr{M}$ and $\overline{m}'_1 \circ \overline{n}' = \overline{n}$,

   (d) $m\colon I'_L \to R \in \mathscr{M}$ such that $\exists t\colon R \overset{p'}{\Longrightarrow}_{GS,n'} R''$ with $p' \in \overline{\mathscr{P}}$ or $der(t) = n' = id_R$ such that i. $\exists a''\colon O \to R'' \in \mathscr{M}$ for pushout (o,o') over $(i_{L,2}, m)$ with ii. $(2)+(3)$ commutes, and iii. $(3)+(4)$ commutes. △

Remark 3.16 (Construction of Weakened Graph Languages) *The construction is based on the idea that cyclic, terminating match-paths are obtained from acyclic, terminating paths by possibly pumping the path after each match by adding arbitrary match-cycles. Therefore, the construction* *starts with an acyclic, terminating match-path m in GS that starts in S, id_S for ac as derived span, id_S for n as co-match and switch $k = 1$. The construction passes through m step-wise with matches $m_i \in m$:*

1. *If there is no match-cycle from and to the target $t_{GS}(m_i)$ of m_i which differs from all paths that are equal to the path that is currently being handled up to shifting of matches ($\mathscr{P} = \varnothing$) and that may lead to a cyclic match-path before m_i or if switch $k = 2$, then a recursive transformation step t with co-match n' is performed via production $s_{GS}(m_i)$, match m_i and co-match n. The construction recursively proceeds with the next match in m for m_i, an extended derived span $der(t) \circ ac$ for ac, co-match n' for n and switch $k = 1$.*

2. *If there is a match-cycle from and to $t_{GS}(m_i)$ which differs from all paths that are equal to the path that is currently being handled up to shifting of matches ($\mathscr{P} \neq \varnothing$), then before performing the recursive transformation step via match m_i in Item 1 with switch $k = 2$, all cycles that may pump path m before m_i are considered at first. Therefore, for each path in $\mathscr{P}$ representing a single cycle iteration before m_i, the construction is called recursively and the result B_{path} is overlapped with the current result A by $A \oplus B_{path}$. Beside the case $(ac, n) \in \mathscr{C}$ where no cycles via path are iterated, set $A \oplus B_{path}$ simulates arbitrary cycle iterations by relevant overlappings. The overlappings are formed by $\mathscr{M}$-decompositions $\mathscr{D}$ (cf. Def. 3.28 and Item 1) and pushout constructions over common overlapping object I'_L (cf. Def. 3.28 and Item 2b). An overlapping is relevant, if the overlapping "occurs" in recursive transformations in GS (cf. Prop. 3.7 and Item 3), i.e, the part which is added to A by the overlapping is guaranteed to be added by some existing recursive transformation (cf. Def. 3.28 and Item 2(d)ii) in the given context (cf. Def. 3.28 and Item 2(d)iii).*

Thus, for acyclic, terminating recursive transformations $t\colon S \xRightarrow{path}_{GS} G$, if there does not exist match-cycles in GS that are reachable from match-path path, then the construction behaves equivalent to t. Otherwise, if match-cycles in GS exist that are reachable from path, then the result of the single iteration of each cycle is overlapped with intermediate results of t, therefore, simulating the last single iteration of each cycle in all contexts that occur in existing recursive transformations based on t. In order to ensure that only a single iteration of each cycle c is considered in recursive calls of the construction while an infinite number of iterations exists, the construction neglects cycles that are equal to c up to shifting of matches while passing stepwise through c. Otherwise, for match-cycle $c = (m_1 \dots m_n)$ at match m_2, the construction would additionally consider cycle $(m_2 \dots m_n, m_1)$ resulting in a two-time iteration of c, etc.. △

Example 3.19 (Weakened Graph Language of Recursive Graph Schemata) *Fig. 3.14 illustrates the weakened graph language of the recursive graph schema GS in Fig. 3.12. The language consists of 16 equivalence classes of morphisms $[ac_i\colon S \to G_i]_{i\in\{0\dots15\}}$ where the mapping ac_i of the* :ATTR *node in S to G_i is given by node name 1 in Fig. 3.14, respectively. While graphs G_0, G_1, G_4 and G_7 are obtained by the four existing acyclic, terminating match-paths in GS that start in S, the other graphs are obtained by additional relevant overlappings with results of single match-cycle iterations that together represent all the "last" cycle iterations in all contexts that occur in cyclic, terminating recursive transformations in GS that start in S. In detail, G_0, G_1, G_4 and G_7 are obtained by stepping through acyclic paths $(m_1, m_9), (m_1, m_2, m_7), (m_1, m_5, m_8)$ and (m_1, m_5, m_4, m_7), respectively. In contrast, G_2 is obtained by path (m_1, m_2) followed by an overlapping with the result of iterating cycle (m_3) once and the succeeding path (m_7). The overlapping is given by match m_3 itself with $m = m'_1 = m_3$ and $i_{L,2} = id_L$ in Def. 3.28 and Item 1 and $der(t) = n' = id_R$ in Def. 3.28 and Item 2d. Thus, the overlapping is relevant in the sense that it does occur in the recursive transformation w.r.t. path (m_1, m_2, m_3, m_7) leading to G_2, i.e.,*

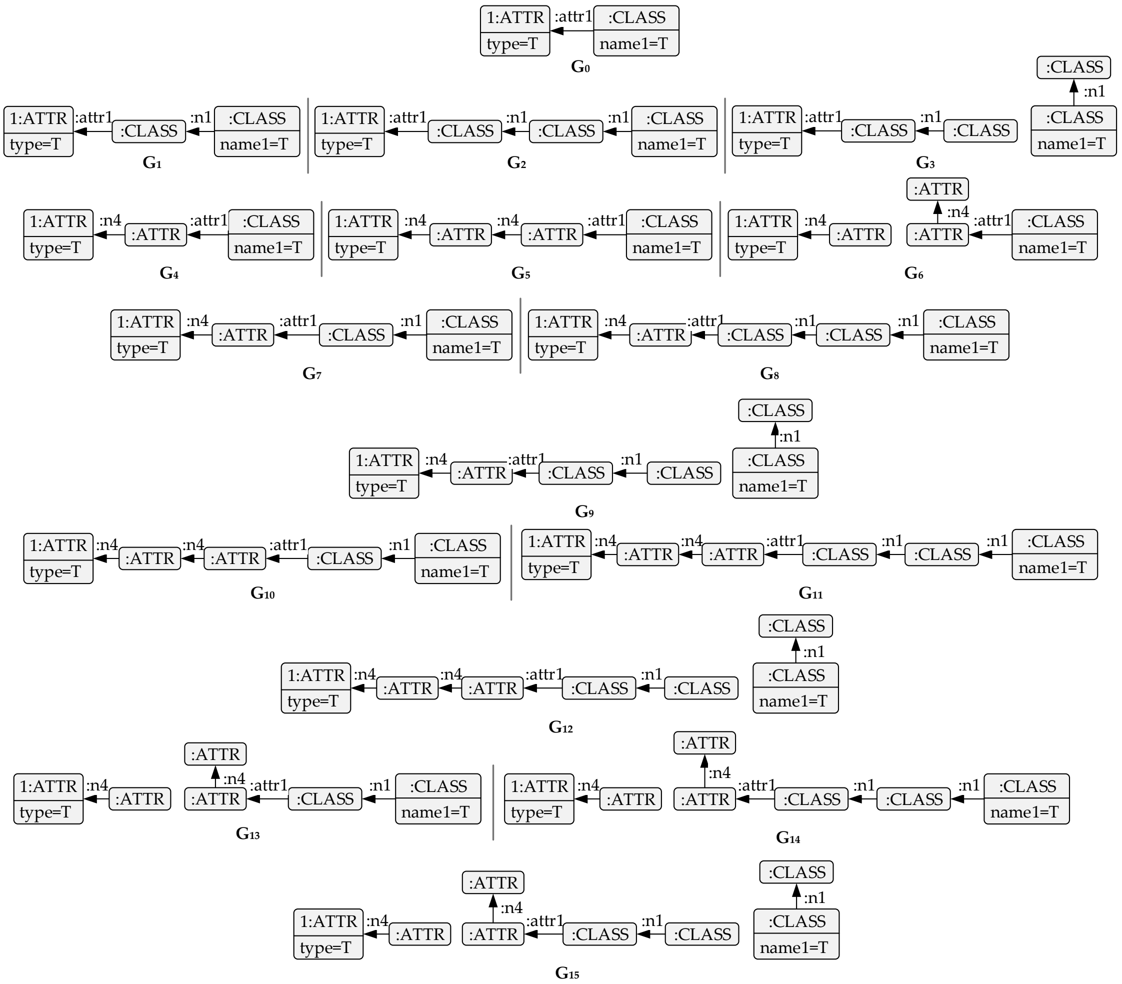


Figure 3.14: Weakened Graph Language

acyclic path (m_1,m_2,m_7) is pumped to a cyclic path (m_1,m_2,m_3,m_7) via additional cycle (m_3). Analogously to G_2, G_3 is obtained by (m_1,m_2) followed by an overlapping via cycle (m_3) and succeeding (m_7). However this time, the overlapping is not exactly given by match m_3 itself but by $m'_1 = m_3$ in Def. 3.28 and Item 1 and furthermore for rule 3 : addNextClass *as source and target of m_3: Graph I'_L consists of node* 1 : ATTR *only and morphisms $m = i_{L,2} = (1 \mapsto 1)$. Therefore, the overlapping is given by a gluing (pushout) of the graph parts of the graph after (m_1,m_2) and the RHS of rule* 3 : addNextClass *via common node* 1 : ATTR. *Note that, there is a recursive transformation t in Def. 3.28 and Item 2d via path $p' = (m_3)$ such that a'' exists and $(2)+(3)$ and $(3)+(4)$ commute. Thus, the overlapping is relevant in the sense that it does occur in recursive transformations w.r.t. paths of the form $(m_1,m_2,m_3,\ldots,m_3,m_7)$, i.e., acyclic path (m_1,m_2,m_7) is pumped to cyclic paths $(m_1,m_2,m_3,\ldots,m_3,m_7)$ by adding an arbitrary number of additional cycles (m_3). Four other overlappings via $m'_1 = m_3$ and i_L-$\mathcal{M}$-decompositions in Def. 3.28 and Item 1 technically exist but all violate the conditions in Def. 3.28 and Item 2d and therefore, they are not considered for language construction: 1. We assume that morphism $i_{L,2} = id_L$ and $m = (1 \mapsto 1, 2 \mapsto 2)$. There are morphisms a'' but without commuting $(2)+(3)$ and $(3)+(4)$. 2. The three remaining overlappings each add a second* : ATTR *node and therefore, in all three cases there does not exists an injective morphism $a'' \in \mathcal{M}$, since, production* 3 : addNextClass *does not create* : ATTR *nodes.* △

In the following, we show that the construction of weakened graph languages is well-defined in the sense that it leads to the desired set $\mathscr{L}_w(GS)$ of equivalence classes of morphisms for a given recursive graph schema GS with start graph S. In particular, 1. for each acyclic, terminating recursive transformation $t\colon S \overset{*}{\Rightarrow}_{GS} G$ which starts in S, there is a class $[der(t)] \in \mathscr{L}_w(GS)$ that coincides with the derived span $der(t)$ of t, 2. for each cyclic, terminating recursive transformation $t\colon S \overset{*}{\Rightarrow}_{GS} G$ which starts in S, there is a class $[ac] \in \mathscr{L}_w(GS)$ that may not coincide with but partially reflect $der(t)$, and 3. each $[ac] \in \mathscr{L}_w(GS)$ partially reflects the derived span $der(t)$ of some terminating recursive transformation $t\colon S \overset{*}{\Rightarrow}_{GS} G$ which starts in S. We use Items 1 to 3 for showing that the language over general recursive graph constraints is a subset of the language over weakened recursive graph constraints in Prop. 3.8 such that domain completeness w.r.t. weakened constraints implies domain completeness w.r.t. general constraints (cf. Thm. 3.7). Furthermore, Item 3 illustrates the fact that the weakened language contains (weakened constraints cover) correct cases only which increases the accuracy of the domain completeness verification w.r.t. weakened constraints by potentially omitting the verification of faulty cases.

Proposition 3.7 (Well-Definedness of Construction of Weakened Graph Languages) *Let $GS = ((S,P),M,s_{GS},t_{GS})$ be a recursive graph schema with non-deleting productions P, $\mathscr{L}(GS)$ be the language of GS and $\mathscr{L}_w(GS)$ be the weakened language of GS. For the construction of weakened graph languages, the following holds in $\mathscr{M}$-adhesive categories with effective pushouts and initial $\mathscr{M}$-subobjects:*

1. *For each terminating, acyclic $t\colon S \overset{*}{\Rightarrow}_{GS} G$ which starts in S there is $[der(t)] \in \mathscr{L}_w(GS)$,*
2. *For each terminating, cyclic $t\colon S \overset{*}{\Rightarrow}_{GS} G$ which starts in S there is $[ac\colon S \to G'] \in \mathscr{L}_w(GS)$ such that there is an $\mathscr{M}$-morphism $i\colon G' \to G$ with $i \circ ac = der(t)$, and*
3. *For each $[ac\colon S \to G'] \in \mathscr{L}_w(GS)$ there is terminating $t\colon S \overset{*}{\Rightarrow}_{GS} G$ which starts in S and $\mathscr{M}$-morphism $i\colon G' \to G$ such that $i \circ ac = der(t)$.* △

Proof. The proof is presented in appendix A.13. □

Based on the notions of tightened and weakened graph languages of recursive graph schemata, we define tightened and weakened recursive graph conditions according to Def. 3.22.

Definition 3.29 (Tightened & Weakened Recursive Graph Condition) Let GS be a recursive graph schema. The recursive graph condition c_{GS} w.r.t. GS is *(tightened w.r.t. a given upper bound G_u) weakened*, if it is formed over the (tightened) weakened graph language of GS (and w.r.t. G_u). △

Example 3.20 (Tightened & Weakened Recursive Graph Constraint) *The weakened recursive graph condition w.r.t. the recursive graph schema GS in Ex. 3.13 is given by the disjunction over all morphisms of the corresponding weakened graph language of GS in Ex. 3.19. Note that the weakened recursive graph constraint is only an approximation to the recursive graph constraint in Ex. 3.16. More precisely, it is not required that for all attributes of type T there is a class of name T such that there is a path between both but the path may also be interrupted as illustrated by graphs $G_3, G_6, G_9, G_{12}, G_{13}, G_{14}$ and G_{15} in Fig. 3.14. In contrast to that, the tightened recursive graph constraint w.r.t. GS is equivalent to the recursive graph constraint in Ex. 3.16 but only for graphs up to the given upper bound.* △

Analogously to Prop. 3.3, in the following we show that tightened and weakened recursive graph conditions are in $\mathcal{M}$-normal form which is necessary to be used in the verification of domain completeness in Sec. 3.2 and Thm. 3.3. Furthermore, we investigate the relationship between graph languages over recursive graph constraints in Prop. 3.8 in order to conclude over the verification of domain completeness w.r.t. infinite graph constraints via Sec. 3.2 and Thm. 3.3 in Thm. 3.7. More precisely, the verification of domain completeness can be performed w.r.t. tightened & weakened recursive graph constraints. While for weakened recursive graph constraints we obtain a general result for domain completeness in Thm. 3.7 and Item 2, for tightened recursive graph constraints we obtain a result for domain completeness up to a given upper bound only in Thm. 3.7 and Item 1.

Theorem 3.6 (Tightened & Weakened Recursive Graph Conditions in $\mathcal{M}$-normal form) *In $\mathcal{M}$-adhesive categories with effective pushouts and initial $\mathcal{M}$-subobjects,* each tightened & weakened recursive graph condition is in $\mathcal{M}$-normal form. △

Proof. Let GS be a recursive graph schema, c_{GS} be the recursive graph condition, $c_{t,GS}$ be the tightened recursive graph condition and $c_{w,GS}$ be the weakened recursive graph condition w.r.t. GS. By Defs. 3.24 and 3.29 it follows that $c_{t,GS}$ is built up by a subset of morphisms of c_{GS}. Therefore by Prop. 3.3 it follows that $c_{t,GS}$ is in $\mathcal{M}$-normal form. By Prop. 3.7 and Item 3, for each morphism $ac\colon S \to G'$ in $c_{w,GS}$, there is a morphism $der(t)\colon S \to G$ in c_{GS} (cf. Defs. 3.21 and 3.22) and morphism $i \in \mathcal{M}$ such that $i \circ ac = der(t)$. By Prop. 3.3, $der(t) \in \mathcal{M}$ implying further that $ac \in \mathcal{M}$ by $\mathcal{M}$-decomposition, i.e., $c_{w,GS}$ is in $\mathcal{M}$-normal form. □

Proposition 3.8 (Relationship between Languages over Recursive Graph Constraints) *Let GS be a recursive graph schema with non-deleting productions. Furthermore, let c_{GS} be the recursive graph constraint, and $c_{w,GS}$ be the weakened recursive graph constraint w.r.t. GS. Moreover, let $c_{t,GS}$ be the tightened recursive graph constraint w.r.t. GS and upper bound G_u. Then, the following holds in $\mathcal{M}$-adhesive categories with effective pushouts and initial $\mathcal{M}$-subobjects:*

1. *$\mathcal{L}_I(\{c_{GS}\})_{G_u} \subseteq \mathcal{L}_I(\{c_{t,GS}\})$ and $\mathcal{L}_I(\{c_{GS}\}) \subseteq \mathcal{L}_I(\{c_{w,GS}\})$, and*
2. *$\mathcal{L}(\{c_{GS}\})_{G_u} \subseteq \mathcal{L}(\{c_{t,GS}\})$ and $\mathcal{L}(\{c_{GS}\}) \subseteq \mathcal{L}(\{c_{w,GS}\})$*

where languages $\mathcal{L}_I(\{c_{GS}\})_{G_u}$ and $\mathcal{L}(\{c_{GS}\})_{G_u}$ are defined accordingly to Sec. 3.2 and Def. 3.13. △

Proof. Let $c_{GS}, c_{t,GS}, c_{w,GS}$ be the corresponding recursive graph constraints over S.
Case ($c_{w,GS}$): Let $G \in \mathcal{L}_I(\{c_{GS}\})$ or $G \in \mathcal{L}(\{c_{GS}\})$, respectively, and $p\colon S \to G \in \mathcal{M}$ be a corresponding morphism. By Sec. 2.2.3 and Rem. 2.10 and Sec. 3.1 and Def. 3.1, $p \models c_{GS} \overset{Sec.\ 2.2.3 and\ Def.\ 2.12}{\Leftrightarrow} \exists q\colon A \to G \in \mathcal{M}$ for some $ac\colon S \to A$ in c_{GS} such that $q \circ ac = p \overset{Defs.\ 3.21 and\ 3.22}{\Leftrightarrow} \exists q\colon A \to G \in \mathcal{M}$ for some $der(t) = ac\colon S \to A$ with $t\colon S \overset{*}{\Rightarrow}_{GS} A$ being terminating and starting in S such that $q \circ der(t) = p \overset{Prop.\ 3.7 and\ Items\ 1 and\ 2\text{ and }Def.\ 3.29}{\Rightarrow} \exists q\colon A \to G \in \mathcal{M}, ac'\colon S \to A'$ in $c_{w,GS}$, and $i\colon A' \to A \in \mathcal{M}$ such that $i \circ ac' = der(t)$ implying further that $\exists q \circ i\colon A' \to G \in \mathcal{M}$ for $ac'\colon S \to A'$ in $c_{w,GS}$ by $\mathcal{M}$-composition $\overset{Sec.\ 2.2.3 and\ Def.\ 2.12}{\Leftrightarrow} p \models c_{w,GS}$. Therefore by Sec. 2.2.3 and Rem. 2.10, $G \models_I c_{w,GS}$ or $G \models c_{w,GS}$, respectively, implying further that $G \in \mathcal{L}_I(\{c_{w,GS}\})$ or $G \in \mathcal{L}(\{c_{w,GS}\})$, respectively, by Sec. 3.1 and Def. 3.1.
Case ($c_{t,GS}$): Let $G \in \mathcal{L}_I(\{c_{GS}\})_{G_u}$ or $G \in \mathcal{L}(\{c_{GS}\})_{G_u}$, respectively, and $p\colon S \to G \in \mathcal{M}$ be a corresponding morphism. By Sec. 3.2 and Def. 3.13, $G \in \mathcal{L}_I(\{c_{GS}\})$ or $G \in \mathcal{L}(\{c_{GS}\})$, respectively, and furthermore, there is $i\colon G \to G_u \in \mathcal{M}$. By Sec. 2.2.3 and Rem. 2.10 and Sec. 3.1 and Def. 3.1, $p \models c_{GS} \overset{Sec.\ 2.2.3 and\ Def.\ 2.12}{\Leftrightarrow} \exists q\colon A \to G \in \mathcal{M}$ for some $ac\colon S \to A$ in c_{GS} such that

$q \circ ac = p$. Thus, by $\mathscr{M}$-composition there is $i \circ q \colon A \to G_u \in \mathscr{M}$ implying further that ac is a morphism in $c_{t,GS}$ by Defs. 3.24 and 3.29, i.e., $p \models c_{t,GS}$. Therefore by Sec. 2.2.3 and Rem. 2.10, $G \overset{I}{\models} c_{t,GS}$ or $G \models c_{t,GS}$, respectively, implying further that $G \in \mathscr{L}_I(\{c_{t,GS}\})$ or $G \in \mathscr{L}(\{c_{t,GS}\})$, respectively, by Sec. 3.1 and Def. 3.1. □

Theorem 3.7 (Verification of Domain Completeness w.r.t. Recursive Graph Constraints) *Let GG be a grammar, $\overline{GS}$ be a set of recursive graph schemata with non-deleting productions, $C_{\overline{GS}}$ be the set of recursive graph constraints, and $C_{w,\overline{GS}}$ be the set of weakened recursive graph constraints w.r.t. $\overline{GS}$. Furthermore, let $C_{t,\overline{GS}}$ be the set of tightened recursive graph constraints w.r.t. $\overline{GS}$ and a common upper bound G_u. Then, the following holds in $\mathscr{M}$-adhesive categories with effective pushouts and initial $\mathscr{M}$-subobjects for a given set of constraints C:*

1. *$\mathscr{L}(C \cup C_{t,\overline{GS}}) \subseteq \mathscr{L}(GG)$ implies domain completeness up to upper bound G_u: $\mathscr{L}(C \cup C_{\overline{GS}})_{G_u} \subseteq \mathscr{L}(GG)$, and*

2. *$\mathscr{L}(C \cup C_{w,\overline{GS}}) \subseteq \mathscr{L}(GG)$ implies domain completeness: $\mathscr{L}(C \cup C_{\overline{GS}}) \subseteq \mathscr{L}(GG)$.* △

Proof. 1. We assume $\mathscr{L}(C \cup C_{t,\overline{GS}}) \subseteq \mathscr{L}(GG)$. Let $G \in \mathscr{L}(C \cup C_{\overline{GS}})_{G_u}$ with $C = C_I \cup C_G$ and $C_{\overline{GS}} = C_{I,\overline{GS}} \cup C_{G,\overline{GS}}$ where constraints $C_I, C_{I,\overline{GS}}$ are designated for initial satisfaction and $C_G, C_{G,\overline{GS}}$ are designated for general satisfaction $\overset{Sec.\ 3.2 and\ Def.\ 3.13}{\Leftrightarrow}$ $G \in \mathscr{L}(C \cup C_{\overline{GS}})$ and $\exists i \colon G \to G_u \in \mathscr{M}$ $\overset{Sec.\ 3.1 and\ Def.\ 3.1}{\Leftrightarrow}$ $G \in \mathscr{L}_I(C_I \cup C_{I,\overline{GS}})$ and $G \in \mathscr{L}(C_G \cup C_{G,\overline{GS}})$, i.e., $G \overset{I}{\models} C_I \cup C_{I,\overline{GS}}$ and $G \models C_G \cup C_{G,\overline{GS}}$ and $\exists i \colon G \to G_u \in \mathscr{M}$ $\overset{Sec.\ 2.2.3 and\ Def.\ 2.15 \text{ and } Sec.\ 3.1 and\ Def.\ 3.1}{\Rightarrow}$ $G \in \mathscr{L}_I(C_I)$, $\forall c_{GS} \in C_{I,\overline{GS}}.G \in \mathscr{L}_I(\{c_{GS}\})$,$G \in \mathscr{L}(C_G)$, and $\forall c_{GS} \in C_{G,\overline{GS}}.G \in \mathscr{L}(\{c_{GS}\})$ and $\exists i \colon G \to G_u \in \mathscr{M}$ $\overset{Sec.\ 3.2 and\ Def.\ 3.13}{\Rightarrow}$ $G \in \mathscr{L}_I(C_I)$, $\forall c_{GS} \in C_{I,\overline{GS}}.G \in \mathscr{L}_I(\{c_{GS}\})_{G_u}$,$G \in \mathscr{L}(C_G)$, and $\forall c_{GS} \in C_{G,\overline{GS}}.G \in \mathscr{L}(\{c_{GS}\})_{G_u}$ $\overset{Prop.\ 3.8}{\Rightarrow}$ $G \in \mathscr{L}_I(C_I)$, $\forall c_{GS} \in C_{I,\overline{GS}}.G \in \mathscr{L}_I(\{c_{t,GS}\})$,$G \in \mathscr{L}(C_G)$, and $\forall c_{GS} \in C_{G,\overline{GS}}.G \in \mathscr{L}(\{c_{t,GS}\})$ with $c_{t,GS} \in C_{t,\overline{GS}}$ $\overset{Sec.\ 2.2.3 and\ Def.\ 2.15 \text{ and } Sec.\ 3.1 and\ Def.\ 3.1}{\Rightarrow}$ $G \in \mathscr{L}(C)$ and $G \in \mathscr{L}(C_{t,\overline{GS}}) \Rightarrow G \in \mathscr{L}(C \cup C_{t,\overline{GS}}) \overset{Assumption}{\Rightarrow} G \in \mathscr{L}(GG)$.

2. We assume $\mathscr{L}(C \cup C_{w,\overline{GS}}) \subseteq \mathscr{L}(GG)$. Let $G \in \mathscr{L}(C \cup C_{\overline{GS}})$ with $C = C_I \cup C_G$ and $C_{\overline{GS}} = C_{I,\overline{GS}} \cup C_{G,\overline{GS}}$ where constraints $C_I, C_{I,\overline{GS}}$ are designated for initial satisfaction and $C_G, C_{G,\overline{GS}}$ are designated for general satisfaction $\overset{Sec.\ 3.1 and\ Def.\ 3.1}{\Leftrightarrow}$ $G \in \mathscr{L}_I(C_I \cup C_{I,\overline{GS}})$ and $G \in \mathscr{L}(C_G \cup C_{G,\overline{GS}})$, i.e., $G \overset{I}{\models} C_I \cup C_{I,\overline{GS}}$ and $G \models C_G \cup C_{G,\overline{GS}}$ $\overset{Sec.\ 2.2.3 and\ Def.\ 2.15 \text{ and } Sec.\ 3.1 and\ Def.\ 3.1}{\Rightarrow}$ $G \in \mathscr{L}_I(C_I)$, $\forall c_{GS} \in C_{I,\overline{GS}}.G \in \mathscr{L}_I(\{c_{GS}\})$,$G \in \mathscr{L}(C_G)$, and $\forall c_{GS} \in C_{G,\overline{GS}}.G \in \mathscr{L}(\{c_{GS}\})$ $\overset{Prop.\ 3.8}{\Rightarrow}$ $G \in \mathscr{L}_I(C_I)$, $\forall c_{GS} \in C_{I,\overline{GS}}.G \in \mathscr{L}_I(\{c_{w,GS}\})$,$G \in \mathscr{L}(C_G)$, and $\forall c_{GS} \in C_{G,\overline{GS}}.G \in \mathscr{L}(\{c_{w,GS}\})$ with $c_{w,GS} \in C_{w,\overline{GS}}$ $\overset{Sec.\ 2.2.3 and\ Def.\ 2.15 \text{ and } Sec.\ 3.1 and\ Def.\ 3.1}{\Rightarrow}$ $G \in \mathscr{L}(C)$ and $G \in \mathscr{L}(C_{w,\overline{GS}}) \Rightarrow G \in \mathscr{L}(C \cup C_{w,\overline{GS}})$ $\overset{Assumption}{\Rightarrow}$ $G \in \mathscr{L}(GG)$.

□

Finally, we show that tightened and weakened recursive graph constraints can effectively be used for verifying domain completeness. This means that the constraints are actually finite under certain conditions such that the verification terminates (cf. Thms. 3.4 and 3.8). Note that the

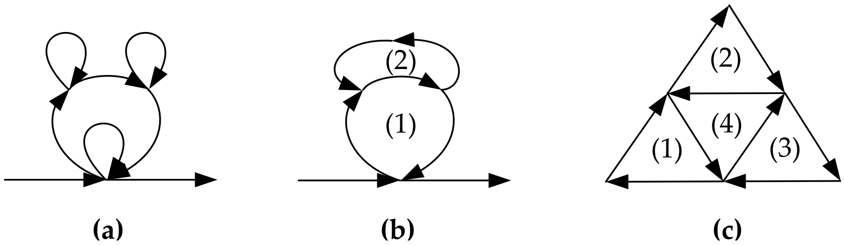


Figure 3.15: Simple Recursive Graph Schema

number of nestings of recursive graph constraints is finite by constructions Defs. 3.22 and 3.29. For ensuring the finiteness of weakened recursive graph constraints, we introduce the notion of simple recursive graph schemata in Def. 3.30. A recursive graph schema is simple if all acyclic match-cycles that are not equal up to shifting of matches are linked via at most one production as depicted in Fig. 3.15 (a). The schemata in Fig. 3.15 (b) and (c) are not simple. Fig. 3.15 (b) contains two acyclic match-cycles (1) and (2) that are linked via two productions. Fig. 3.15 (c) contains a cycle of acyclic match-cycles (1), (2) and (3) that are linked to each other via one production, respectively. However, this situation always leads to an acyclic match-cycle (4) which is linked to other match-cycles via more than one production. Therefore, a cycle of acyclic match-cycles is also forbidden for simple recursive graph schemata.

Definition 3.30 (Simple Recursive Graph Schema) Let GS be a recursive graph schema and $Paths_{\_,\_}(GS)|_{\sim}$ be the equivalence classes of all acyclic match-cycles in GS that are equal up to shifting of matches as defined in Def. 3.27. Then GS is simple, if for all $[path_1], [path_2] \in Paths_{\_,\_}(GS)|_{\sim}$ and for all $m_i, m_{i'} \in path_1$ and $m_j, m_{j'} \in path_2$ it holds that $s_{GS}(m_i) = s_{GS}(m_j)$ and $s_{GS}(m_{i'}) = s_{GS}(m_{j'})$ implies that $m_i = m_{i'}$ and $m_j = m_{j'}$. △

Example 3.21 (Simple Recursive Graph Schema) *The recursive graph schema in Fig. 3.12 is simple while the schema in Fig. 3.13 is not simple.* △

Theorem 3.8 (Finiteness of Recursive Graph Constraints) *Let* $GS = ((S,P),M,s_{GS},t_{GS})$ *be a recursive graph schema.*

1. *The recursive graph constraint* c_{GS} *w.r.t. GS is finite, if:*
   (a) *The set* $Paths_S(GS)$ *of acyclic, terminating match-paths in GS that start in S is finite, and*
   (b) *For all match-paths path* $\in Paths_S(GS)$ *there is no match-cycles in GS that is reachable from path.*
2. *In* $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$*, the tightened recursive graph constraint* $c_{t,GS}$ *w.r.t. GS and upper bound* G_u *is finite, if:*
   (a) *Recursive graph schema GS contains non-deleting productions P only,*
   (b) *Graph* G_u *is finite, and*
   (c) *The set of matches M is finite.*
3. *In* $(\mathbf{AGraphs}_{ATGI,fin}, \mathcal{M}_{fin})$*, the weakened recursive graph constraint* $c_{w,GS}$ *w.r.t. GS is finite, if:*
   (a) *The set of matches M is finite, and*

(b) *Recursive graph schema GS is simple.* △

Proof. By Sec. 2.2.3 and Def. 2.12, for constraint c is finite we have to show that the index set I of every disjunction $\vee_{i \in I}$ in c is finite.

1. By assumptions Thm. 3.8 and Items 1a and 1b and Def. 3.21, language $\mathscr{L}(GS)$ of GS is the finite set of derived spans of the finite set of terminating recursive transformation sequences starting at S. Thus, by construction Def. 3.22, c_{GS} is finite.

2. W.l.o.g. and by assumption Thm. 3.8 and Item 2a we assume that each recursive transformation step via a production $p \in P$ and match $m \in M$ creates at least one graph element while preserving the remaining elements. By M being finite by assumption Thm. 3.8 and Item 2c, we have finitely many possibilities at each step to continue with the next step until we have exceeded finite upper bound G_u (cf. assumption Thm. 3.8 and Item 2b). Therefore, there are finitely many recursive transformation sequences up to upper bound G_u leading to finitely many derived spans in tightened language $\mathscr{L}_t(GS, G_u)$ of GS and w.r.t. G_u (cf. Def. 3.24). Thus, by construction Def. 3.29, $c_{t,GS}$ is finite.

3. We focus on the construction of weakened languages in Def. 3.28. There are at most $|M|!$ acyclic match-paths in GS, each consisting of at most $|M|$ matches, since, having a match $m \in M$ two-times in a path yields a cyclic path. Therefore by assumption Thm. 3.8 and Item 3a of M being finite, the set of acyclic match-paths in GS is finite and furthermore, each acyclic match-path consists of a finite set of matches. Thus, the set $\underline{Paths}_S(GS)$ of acyclic, terminating match-paths in GS that start in S is finite. Analogously, the set $\mathscr{P}$ of acyclic match-cycles in GS that are equal up to shifting of matches is finite. Therefore, we can iterate over all paths $m \in \underline{Paths}_S(GS)$ and call $\underline{\mathscr{L}_w}$, respectively. In each call for $\underline{\mathscr{L}_w}(m, 1, id, id, 1)$ we can iterate over the finite set of matches $m_i \in m$ if $\mathscr{P} = \varnothing$ for each $m_i \in m$. If for some $m_i \in m$, $\mathscr{P} \neq \varnothing$, then we can iterate over finite $\mathscr{P}$ and the finite set of overlappings $(A \oplus B_{path})$ in $\mathscr{C}$ with recursive calls of $\underline{\mathscr{L}_w}(path, 1, id, id, 2)$ for each $path \in \mathscr{P}$, since, we are in category $(\mathbf{AGraphs}_{ATGI,fin}, \mathscr{M}_{fin})$ of finite graphs and therefore, the set of possible overlappings is finite. Analogously, we conclude for all recursive calls and furthermore, the recursion is guaranteed to end at depth $|M|!$ for each case, since, GS is simple by assumption Thm. 3.8 and Item 3b. Thus, set $\mathscr{C}$ is finite in each case. In conclusion, the weakened language $\mathscr{L}_w(GS)$ of GS is a finite set of morphisms implying that the weakened recursive graph constraint $c_{w,GS}$ w.r.t. GS is finite by Def. 3.29.

□

Note that weakened constraints involve additional approximations – A graph up to an upper bound satisfies the original constraint if and only if it satisfies the tightened constraint. However, a graph satisfies the weakened constraint if it satisfies the original constraint but not necessarily vice versa. Thus, verifying domain completeness against weakened constraints my involve more graphs than verifying against the original constraints. Therefore, verifying domain completeness w.r.t. tightened constraints may be more accurate and lead to less false negatives in comparison to a verification w.r.t. weakened constraints when having an upper bound for the size of graphs. On the other hand, verifying domain completeness w.r.t. weakened constraints yields a more general result without an upper bound and may be more efficient, since, not all possibilities of cyclic match-paths up to a certain upper bound are listed in weakened constraints and therefore, must not be checked.

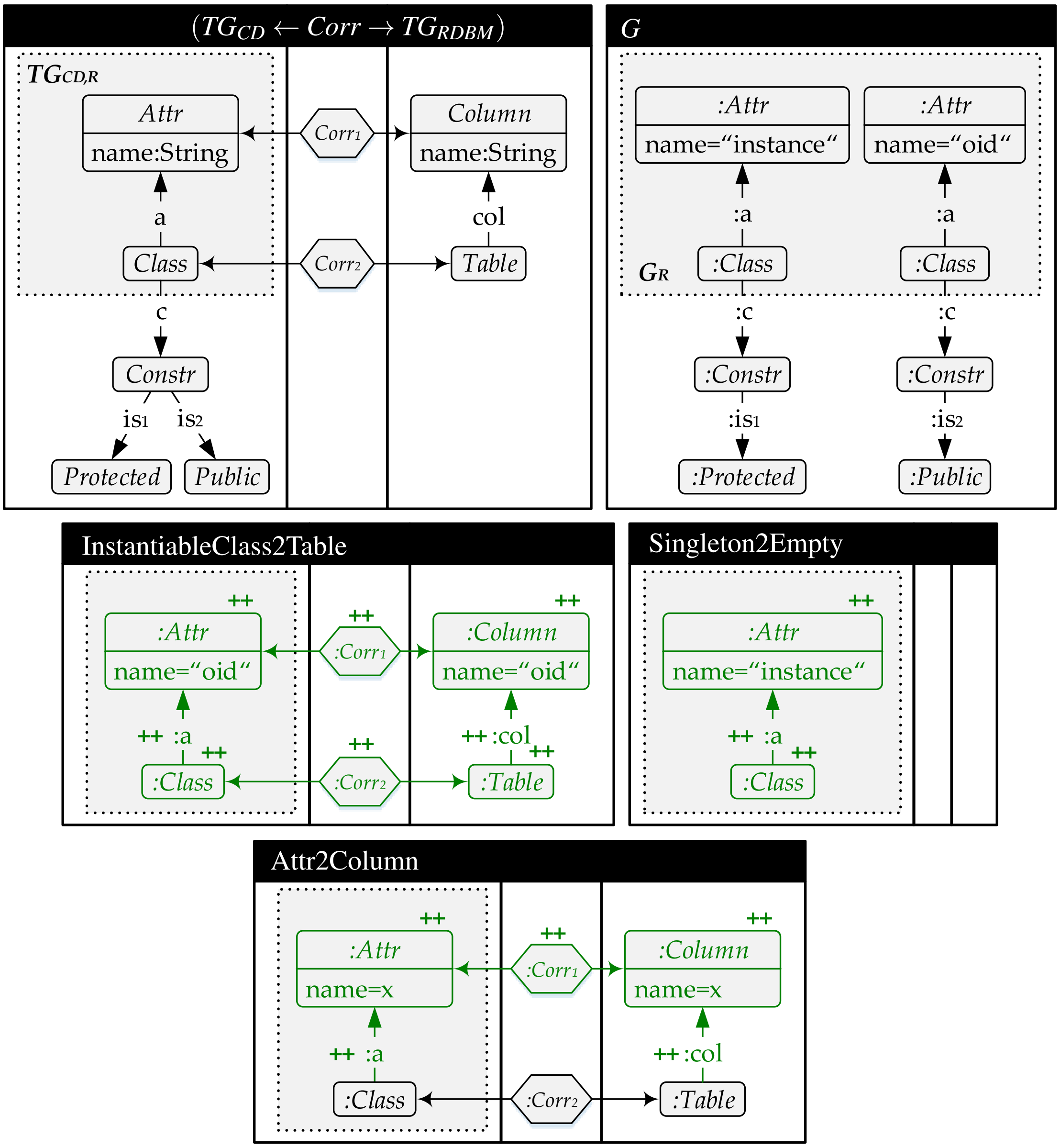


Figure 3.16: Triple Type Graph $(TG_{CD} \leftarrow Corr \rightarrow TG_{RDBM})$ with Domain Restriction $TG_{CD,R}$ (top,left), Graph G with Resitriction G_R (top,right) and Triple Rules (bottom)

3.5 Domain Restrictions

According to Chap. 3, we assume that the graphs (models) in the domain of discourse are defined by a domain type graph together with a set C of domain graph constraints. When verifying domain completeness w.r.t. a given graph grammar GG, we may not be interested in a complete coverage of language $\mathscr{L}(C)$ by language $\mathscr{L}(GG)$ but only in the coverage of certain elements of each graph in $\mathscr{L}(C)$ by $\mathscr{L}(GG)$ which we call domain completeness under restrictions. In Fig. 3.16 we adress the translation of UML class diagrams (CDs) into relational database models (RDBMs) based on a triple graph grammar (TGG) (cf. Sec. 1.2). Therefore, the translation (TGG) only covers those elements $TG_{CD,R}$ in class diagrams that are related to RDBMs, i.e., classes and their attributes, while neglecting other details concerning class constructors and their

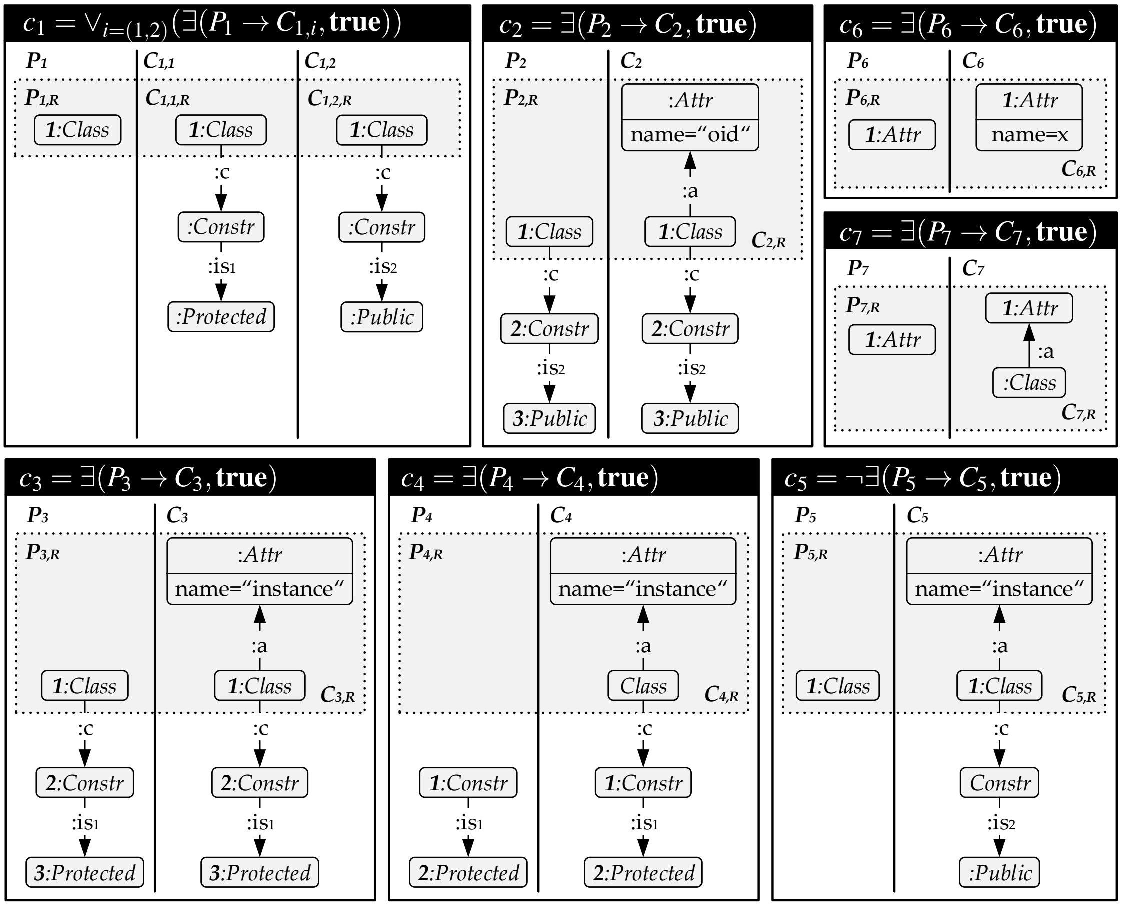


Figure 3.17: Domain Graph Constraints with Restrictions

visibilities. However, the graph constraints restricting the structures of class diagrams may not only cover RDBM related elements but also other irrelevant elements for the translation. This is problematic when verifying domain completeness under restrictions based on the techniques for verifying domain completeness in Sec. 3.2, since, the extension of graphs via the constraints may lead to graphs with irrelevant elements that cannot be covered by the TGG causing the verification of C-extension completeness in Def. 3.8 to fail. In Thm. 3.9, we give sufficient conditions under which the constraints can be restricted to relevant elements such that domain completeness under restrictions w.r.t. the original constraints can be verified based on the techniques for verifying domain completeness w.r.t. the restricted constraints in the sense of Sec. 3.2.

General Assumption: Initial and general satisfaction of constraints are defined over an initial object (cf. Def. 2.15). Therefore, we assume that the results of this Section are applied in the context of $\mathcal{M}$-adhesive categories with initial object. Moreover, for the extension of constraints in Thm. 3.10, we assume $\mathcal{M}$-adhesive categories with effective pushouts.

For example, the triple type graph $(TG_{CD} \leftarrow Corr \to TG_{RDBM})$ in Fig. 3.16 (top,left) defines the model elements of class diagrams and database models. Class diagrams (type graph TG_{CD}) may contain several Classes where each class may have an arbitrary number of Attributes, each with a specific name of type String. Furthermore, each class may have a Constructor with visibility Protected or Public. Classes with protected constructors are singleton classes, i.e., there exists exactly one instance of each class. In contrast, classes with public constructors are instantiable classes, i.e., there may exist an arbitrary number of instances of each class. On the other hand, RDBMs (type graph TG_{RDBM}) may contain several Tables, each with an arbitrary number of Columns with a specific name of type String. Moreover, classes (attributes) in CDs correspond to tables (columns) in RDBMs via $Corr_2$ ($Corr_1$) (cf. type graph $Corr$).

Furthermore, the constraints in Fig. 3.17 state that 1. each Class has a Constructor of visibility Protected or Public ($c_1 = \vee_{i=(1,2)}(\exists(P_1 \to C_{1,i}, \mathbf{true}))$), 2. each instantiable class has an Attribute "oid" (object id) which contains the unique id for each instantiated object of the class ($c_2 = \exists(P_2 \to C_2, \mathbf{true})$), 3. each singleton class has an attribute "instance" which contains the singleton instance object of the class ($c_3 = \exists(P_3 \to C_3, \mathbf{true})$), 4. each protected constructor is the constructor of some class with attribute "instance" ($c_4 = \exists(P_4 \to C_4, \mathbf{true})$ - Analogously, for public constructors a similar constraint is defined with attribute "oid"), 5. each class is either singleton or instantiable, i.e., for each class there does not exist a public constructor and an attribute "instance" at the same time ($c_5 = \neg\exists(P_5 \to C_5, \mathbf{true})$ - Analogously, a similar constraint is defined for the combination of protected constructor and attribute "oid"), 6. each attribute has a name ($c_6 = \exists(P_6 \to C_6, \mathbf{true})$), and 7. each attribute is assigned to a class ($c_7 = \exists(P_7 \to C_7, \mathbf{true})$). Moreover, a constraint may be defined that prohibits classes to have protected and public constructors at the same time.

The triple rule InstantiableClass2Table in Fig. 3.16 (bottom) defines the translation from CDs into RDBMs by mapping instantiable Classes with Attribute oid in CDs to corresponding Tables with Column oid in RDBMs. Triple rule Singleton2Empty maps singleton Classes with Attribute instance in CDs to nothing in RDBMs, i.e., only instantiable classes are translated to corresponding tables whereas singleton classes are omitted. Finally, triple rule Attr2Column translates general Attributes with name "x" of Classes in CDs into Columns with the same name "x" of corresponding Tables in RDBMs.

Therefore, the translation is restricted to (only covers) classes and attributes in CDs while neglecting constructors and their visibilities. According to [SEM$^+$12], the restriction is formalised by an $\mathscr{M}$-type morphism $t\colon TG_{CD,R} \to TG_{CD} \in \mathscr{M}$ from the restricted type graph $TG_{CD,R}$ which only contains classes and their attributes to the complete domain type graph TG_{CD} of class diagrams (cf. Fig. 3.16 (top,left)). Therefore, each graph and graph constraint can be restricted along the type morphism while both may also contain elements outside the restriction, in general. For example, graph G_R in Fig. 3.16 (top,right) is the restriction of graph G along type morphism t and constraints $c_i[P_i \mapsto P_{i,R}, C_i \mapsto C_{i,R}]$ in Fig. 3.17 where premise (conclusion) P_i (C_i) is substituted by $P_{i,R}$ ($C_{i,R}$) in each constraint are the restrictions of constraints $(c_i)_{i=(1..6)}$ along t. The domain completeness problem under restrictions w.r.t. a given set of domain graph constraints C and a graph grammar GG is as follows: Given a restriction of the domain type graph, does it hold for each graph G in $\mathscr{L}(C)$ that the restriction of G which comprises relevant elements only (i.e., elements that are typed over the restricted type graph) is covered by (contained in) grammar GG (language $\mathscr{L}(GG)$)?

In the following, we first recall the definitions for rectrictions of graphs and positive (nested) graph constraints along type morphisms from [SEM$^+$12] and then we define the domain completeness problem under restrictions in accordance with the general domain completeness problem from Def. 3.2.

$$\begin{tikzcd}
TG & G \arrow[l, "t_G"'] & G' \arrow[l, "a"'] \\
TG_R \arrow[u, "t"] \arrow[ur, phantom, "(1)"] & G_R \arrow[l, "t_{G_R}"] \arrow[u, "t'"] \arrow[ur, phantom, "(2)"] & G'_R \arrow[l, "a_R"] \arrow[u, "t''"']
\end{tikzcd}$$

Definition 3.31 (Restriction along Type Morphism [Def. 3.1 [SEM+12])] Given an object (G,t_G) typed over TG via $t_G\colon G \to TG$ and type morphism $t\colon TG_R \to TG \in \mathscr{M}$, then *$TG_R$ is called a restriction of TG* and *(G_R,t_{G_R}) is a restriction of (G,t_G) along t* with induced morphism $t'\colon G_R \to G$, written $G_R = Restr_t(G)$, if (1) is a pullback. Given morphism $a\colon G' \to G$, then *$a_R\colon G'_R \to G_R$ is a restriction of a along t*, written $a_R = Restr_t(a)$, if additionally (2) is a pullback. △

Remark 3.17 (Restriction of Objects) *Note that by pullback composition with $(1)+(2)$ being a pullback, $(G'_R,t_{G_R} \circ a_R)$ is also a restriction of $(G',t_G \circ a)$ along t with induced morphism $t''\colon G'_R \to G'$, written $G'_R = Restr_t(G')$.* △

Definition 3.32 (Restriction of Nested Conditions [Def. 3.2 [SEM+12])] Given a nested condition ac_P typed over TG and restriction TG_R of TG with type morphism $t\colon TG_R \to TG \in \mathscr{M}$. Then, *the restriction $Restr_t(ac_P)$ of ac_P along t* is defined as follows:

1. $Restr_t(ac_P) := \textbf{true}$ for $ac_P = \textbf{true}$,
2. $Restr_t(ac_P) := \exists(Restr_t(a), Restr_t(ac_C))$ for $ac_P = \exists(a\colon P \to C, ac_C)$,
3. $Restr_t(ac_P) := \neg Restr_t(ac'_P)$ for $ac_P = \neg ac'_P$, and
4. $Restr_t(ac_P) := (\vee_{i\in I}) \wedge_{i\in I} (Restr_t(ac_{P,i}))$ for $ac_P = (\vee_{i\in I}) \wedge_{i\in I} (ac_{P,i})$. △

Definition 3.33 (Domain Completeness (Problem) under Restrictions) Given the language $\mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ over domain graph constraints $C = C_I \cup C_G$ and domain type graph TG with conditions C_I (C_G) that are designated for initial (general) satisfaction. Furthermore, let $t\colon TG_R \to TG \in \mathscr{M}$ be a type morphism such that TG_R is a restriction of TG and let $\mathscr{L}(GG)$ be the language over graph grammer GG and restricted type graph TG_R. *Domain completeness under restrictions* holds if for all $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ it is true that $Restr_t(G) \in \mathscr{L}(GG)$. Thus, the *domain completeness problem under restrictions* is defined as follows: Does it hold that for all $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ it is true that $Restr_t(G) \in \mathscr{L}(GG)$? △

As already discussed at the beginning of this Section, in order to apply the results from Sec. 3.2 to the restricted domain type graph for verifying domain completeness under restrictions, the domain constraints need to be restricted at first. However, the restriction of constraints may lead to constraints with a shifted meaning. For example, when assuming general (initial) satisfaction for constraints, the restriction of constraints c_2, c_3, c_4 and c_5 in Fig. 3.17 lead to constraints with a meaning that was not prevailing before the restriction: 1. The restriction of c_2 claims that each (there exists a) Class (that) has an Attribute oid, 2. the restriction of c_3 claims that each (there exists a) Class (that) additionally has an Attribute instance, 3. the restriction of c_4 claims that there exists a Class with Attribute instance, and 4. the restriction of c_5 claims that each (there exists a) Class (which) does not have an Attribute instance. For general satisfaction, the restrictions of c_2 and c_3 lead to a contradiction in view of the unrestricted constraints, since, each class should be either singleton or instantiable by constraints c_1 and c_5. Furthermore, the restriction of c_4 (c_5) prohibits class diagrams that only contain instantiable classes (or that contain singleton classes). Similarly for initial satisfaction, the restrictions prohibit class diagrams that only contain instantiable or singleton classes. Analogously, we obtain inconsistent results for the restrictions of negations $\neg c_2, \neg c_3$ and $\neg c_4$. Thus, negations and constraints ac_P with elements in the premise graph P that are outside the restriction may lead to inconsistent results when being restricted. This is due to the fact that the context of elements in the premise (or conclusion) that is outside the restriction is lost after the restriction such that the restricted premise (or conclusion) may match to a wider range of graph patterns.

In the following, we show for the (general) initial satisfaction of positive graph constraints (which only contain restricted elements in their premises) that they can be restricted such that the restriction leads to a consistent result in the sense that each graph which satisfies the original constraint does also satisfy the restricted one as stated in Prop. 3.9. This extends the results in [SEM^{+}12] from initial to general satisfaction which is the usual interpretation of graph constraints, in particular in view of domain completeness (cf. Sec. 3.3). In Lem. 3.16, we first prove that a graph G initially (generally) satisfies a restricted condition $Restr_t(ac_P)$ if and only if its restriction $Restr_t(G)$ initially (generally) satisfies $Restr_t(ac_P)$ which is used in the proof of Prop. 3.9.

Lemma 3.16 (Restriction of Objects and Satisfaction) *Let $t\colon TG_R \to TG \in \mathscr{M}$ be a type morphism and TG_R be the restriction of type graph TG. Let ac_P be a nested condition typed over TG, $Restr_t(ac_P)$ be its restriction along t, $(G, t_G\colon G \to TG)$ be an object typed over TG via t_G and $Restr_t(G)$ be its restriction along t. Then, $G \overset{I}{\models} Restr_t(ac_P)$ if and only if $Restr_t(G) \overset{I}{\models} Restr_t(ac_P)$. Furthermore, $G \models Restr_t(ac_P)$ if and only if $Restr_t(G) \models Restr_t(ac_P)$.* △

Proof. The proof is presented in appendix A.14. □

In Lem. 3.18, we show that if a graph G generally satisfies a positive constraint ac_P where premise P only contains elements within the restriction, then restricted graph $Restr_t(G)$ also generally satisfies the restricted constraint $Restr_t(ac_P)$ which is used in the proof of Prop. 3.9. This extends the result for initial satisfaction from Cor. 5.2 in [SEM^{+}12] to general satisfaction. In contrast to the result for initial satisfaction, Lem. 3.18 does not hold for positive constraints with elements in the premise that are outside the restriction, in general, as shown by the following counter-example. Consider constraint c_3 in Fig. 3.17. Given a class diagram with instantiable classes only that generally satisfies c_3, then the restriction of the class diagram according to $TG_{CD,R}$ in Fig. 3.16 may not generally satisfy the restricted constraint $Restr_t(c_3)$, since, there may be classes without an attribute of name instance. Furthermore, similarly to the result for initial satisfaction, Lem. 3.18 does not hold for non-positive constraints even if the premise only contains elements within the restriction, in general (cf. constraint c_5 in Fig. 3.17). Lem. 3.18 holds for constraints c_1, c_6 and c_7 in Fig. 3.17, since, these constraints are positive and only contain elements in their premises P_1, P_6 and P_7 that are within the restriction. Before proving Lem. 3.18, we prove Lem. 3.17 which is used in the proof of Lem. 3.18.

Lemma 3.17 (Compatibility of Restriction and General Satisfaction I) *Let $t\colon TG_R \to TG \in \mathscr{M}$ be a type morphism and TG_R be the restriction of type graph TG. Let ac_P be a positive nested condition over premise P and typed over TG and $Restr_t(ac_P)$ be its restriction over $Restr_t(P)$ along t with induced morphism $i\colon Restr_t(P) \to P$. Given an object $(G, t_G\colon G \to TG)$ typed over TG via t_G and its restriction $Restr_t(G)$ along t with induced morphism $t'\colon Restr_t(G) \to G$. Then, it holds that if there is $p\colon P \to G \in \mathscr{M}$ with $p \models ac_P$, then there is a unique $p_R\colon Restr_t(P) \to Restr_t(G) \in \mathscr{M}$ such that $t' \circ p_R = p \circ i$. Furthermore, it holds that $p_R \models Restr_t(ac_P)$.* △

Proof. Let $p \in \mathscr{M}$ with $p \models ac_P$. By construction of $Restr_t(G)$, (3) is a pullback (PB) with $t' \in \mathscr{M}$ (cf. Def. 3.31), since, $\mathscr{M}$-morphisms $t \in \mathscr{M}$ are closed under pullbacks. By induction over the structure of nested conditions: **Basis.** Let $ac_P = \mathbf{true} = Restr_t(ac_P)$. By construction of $Restr_t(P)$, $(1)+(2)$ is a pullback. By p being a morphism and G, P both being typed over

TG via t_G and $t_C \circ a$ it follows that $t_G \circ p = t_C \circ a$ $^{(*^1)}$. Thus, $t_G \circ p \circ i \overset{(*^1)}{=} t_C \circ a \circ i \overset{PB(1)+(2)}{=} t \circ t_{C_R} \circ a_R$. By the universal property of pullback (3), there is a unique p_R with $t' \circ p_R = p \circ i$ and $p_R \models \mathbf{true} = Restr_t(ac_P)$. By $\mathscr{M}$-morphisms $t \in \mathscr{M}$ are closed under pullbacks $(1)+(2)$, $i \in \mathscr{M}$. By $\mathscr{M}$-composition, $p \circ i \in \mathscr{M}$ and therefore, by $\mathscr{M}$-decomposition with $t' \in \mathscr{M}$, $p_R \in \mathscr{M}$.

$$\begin{tikzcd}
& & \overset{ac_C}{\triangledown} & \\
G \arrow[r, "t_G"] & TG & C \arrow[l, "t_C"'] \arrow[ll, bend right, "q"'] & P \arrow[l, "a"'] \arrow[lll, bend right=40, "p"'] \arrow[d, bend left, "i^{-1}"] \\
Restr_t(G) \arrow[u, "t'"] \arrow[r, "t_{G_R}"] \arrow[ur, phantom, "(3)"] & TG_R \arrow[u, "t"] \arrow[ur, phantom, "(1)"] & Restr_t(C) \arrow[u, "t''"] \arrow[l, "t_{C_R}"'] \arrow[ll, dotted, bend left, "q_R"] \arrow[ur, phantom, "(2)"] & Restr_t(P) \arrow[l, "a_R"'] \arrow[u, bend left, "i"] \arrow[lll, dotted, bend left=40, "p_R"] \\
& & \underset{Restr_t(ac_C)}{\triangle} &
\end{tikzcd}$$

Hypothesis. The assumption holds for positive conditions $ac_C, ac_{P,i}, i \in I$ and their restrictions $Restr_t(ac_C), Restr_t(ac_{P,i})$. **Step.** Let $ac_P = \exists(a\colon P \to C, ac_C)$ and $Restr_t(ac_P) = \exists(a_R\colon Restr_t(P) \to Restr_t(C), Restr_t(ac_C))$ with pullbacks (1) and (2) by the construction of restrictions of conditions (cf. Def. 3.32). Pullback composition implies that $(1)+(2)$ is a pullback. Analogously to the induction basis, there is a unique $p_R \in \mathscr{M}$ with $t' \circ p_R = p \circ i$ $^{(*^3)}$. By assumption $p \models ac_P$, there is $q \in \mathscr{M}$ with $q \circ a = p$ and $q \models ac_C$ $^{(*^4)}$. By q being a morphism and G, C being typed over TG via t_G, t_C, $t_G \circ q = t_C$ $^{(*^2)}$. Thus, $t_g \circ q \circ t'' \overset{(*^2)}{=} t_C \circ t'' \overset{PB(1)}{=} t \circ t_{C_R}$. By the universal property of pullback (3), there is a unique q_R with $t' \circ q_R = q \circ t''$ $^{(*^5)}$. Furthermore, $q_R \in \mathscr{M}$, since, $t'' \in \mathscr{M}$ by $\mathscr{M}$-morphisms $t \in \mathscr{M}$ are closed under pullbacks (1), $q \circ t'' \in \mathscr{M}$ by $\mathscr{M}$-composition and $q_R \in \mathscr{M}$ by $\mathscr{M}$-decomposition with $t' \in \mathscr{M}$ and $(*^5)$. Thus, $t' \circ q_R \circ a_R \overset{(*^5)}{=} q \circ t'' \circ a_R \overset{PB(2)}{=} q \circ a \circ i \overset{(*^4)}{=} p \circ i \overset{(*^3)}{=} t' \circ p_R$. By $t' \in \mathscr{M}$ being a monomorphism, $q_R \circ a_R = p_R$. Furthermore, by induction hypothesis, $q_R \models Restr_t(ac_C)$. Therefore, $p_R \models Restr_t(ac_P)$.

For conditions $ac_P = \wedge_{i \in I}(ac_{P,i})$ we conclude as follows. Assumption $p \models ac_P$ implies that $p \models ac_{P,i}$ for all $i \in I$. Therefore, there is a unique $p_R \in \mathscr{M}$ with $t' \circ p_R = p \circ i$ and $p_R \models Restr_t(ac_{P,i})$ for all $i \in I$ implying further that $p_R \models \wedge_{i \in I}(Restr_t(ac_{P,i})) = Restr_t(ac_P)$. Analogously, we prove the assumption for conditions $ac_P = \vee_{i \in I}(ac_{P,i})$. □

Lemma 3.18 (Compatibility of Restriction and General Satisfaction II) *Let $t\colon TG_R \to TG \in \mathscr{M}$ be a type morphism and TG_R be the restriction of type graph TG. Let ac_P be a positive nested condition over premise P and typed over TG with $P \cong Restr_t(P)$ and $Restr_t(ac_P)$ be its restriction over $Restr_t(P)$ along t. Given an object $(G, t_G\colon G \to TG)$ typed over TG via t_G and its restriction $Restr_t(G)$ along t with induced morphism $t'\colon Restr_t(G) \to G$, then it holds that $G \models ac_P \implies Restr_t(G) \models Restr_t(ac_P)$.* △

Proof. Based on the figure in the proof of Lem. 3.17, by induction over the structure of nested conditions: **Basis.** For $ac_P = \mathbf{true} = Restr_t(ac_P)$, for all $p_R\colon Restr_t(P) \to Restr_t(G)$ it holds that $p_R \models Restr_t(ac_P)$, i.e., $Restr_t(G) \models Restr_t(ac_P)$.

We show for $ac_P = \exists(a\colon P \to C, ac_C)$ and its restriction $Restr_t(ac_P) = \exists(a_R\colon Restr_t(P) \to Restr_t(C), Restr_t(ac_C))$ that if $G \models ac_P$, then for all $p_R\colon Restr_t(P) \to Restr_t(G) \in \mathscr{M}$ there is

$q_R\colon \mathit{Restr}_t(C) \to \mathit{Restr}_t(G) \in \mathscr{M}$ with $q_R \circ a_R = p_R$ and $q_R \models \mathit{Restr}_t(ac_C)$. This directly implies that $\mathit{Restr}_t(G) \models \mathit{Restr}_t(ac_P)$. Let $P \cong \mathit{Restr}_t(P)$ with (inverse) isomorphism $i\colon \mathit{Restr}_t(P) \to P$ ($i^{-1}\colon P \to \mathit{Restr}_t(P)$) and $i, i^{-1} \in \mathscr{M}$, since, class $\mathscr{M}$ contains all isomorphisms. Let $p_R \in \mathscr{M}$. There is $p = t' \circ p_R \circ i^{-1} \in \mathscr{M}$ by $\mathscr{M}$-composition and $t', t'' \in \mathscr{M}$, since, pullbacks (PBs) (3) and (1) preserves $\mathscr{M}$-morphisms $t \in \mathscr{M} \overset{G \models ac_P}{\Rightarrow}$ There is $q\colon C \to G \in \mathscr{M}$ with $q \models ac_C$ and $q \circ a = p = t' \circ p_R \circ i^{-1}\ {}^{(*^1)}$. By q being a morphism, $t_G \circ q = t_C \Rightarrow t_G \circ q \circ t'' = t_C \circ t'' \overset{PB(1)}{=} t \circ t_{C_R}$. By the universal property of pullback (3), there is a unique $q_R\colon \mathit{Restr}_t(C) \to \mathit{Restr}_t(G)$ with $t' \circ q_R = q \circ t''\ {}^{(*^2)}$. Furthermore, $q_R \in \mathscr{M}$ by first $\mathscr{M}$-composition with $t' \circ q_R = q \circ t'' \in \mathscr{M}$ and then, $\mathscr{M}$-decomposition with additionally $t' \in \mathscr{M}$. Thus, $t' \circ p_R \overset{i \text{ is isomorphism}}{=} p \circ i \overset{(*^1)}{=} q \circ a \circ i \overset{PB(2)}{=} q \circ t'' \circ a_R \overset{(*^2)}{=} t' \circ q_R \circ a_R \overset{t' \in \mathscr{M} \text{ is a monomorphism}}{\Rightarrow} q_R \circ a_R = p_R$. Furthermore, by Lem. 3.17, $q_R \models \mathit{Restr}_t(ac_C)$. Thus, $p_R \models \mathit{Restr}_t(ac_P)$ and furthermore, $\mathit{Restr}_t(G) \models \mathit{Restr}_t(ac_P)$. **Hypothesis.** The assumption holds for conditions $ac_{P,i}, i \in I$ and their restrictions $\mathit{Restr}_t(ac_{P,i})$. **Step.** Let condition $ac_P = \wedge_{i\in I}(ac_{P,i})$. Assumption $G \models ac_P$ implies that $G \models ac_{P,i}$ for all $i \in I$. By induction hypothesis, $\mathit{Restr}_t(G) \models \mathit{Restr}_t(ac_{P,i})$ for all $i \in I$. Therefore, $\mathit{Restr}_t(G) \models \wedge_{i\in I}(\mathit{Restr}_t(ac_{P,i})) = \mathit{Restr}_t(ac_P)$. For conditions $ac_P = \vee_{i\in I}(ac_{P,i})$ we conclude analogously. □

In Prop. 3.9, we give sufficient conditions under which constraints can be restricted such that each graph which initially (generally) satisfies the original constraint does also initially (generally) satisfy the restricted one. Therefore, the domain constraints can be restricted such that all graphs that satisfy the original domain constraints do also satisfy the restricted domain constraints. Thus, (a successful verification of) domain completeness under restrictions w.r.t. the restricted constraints implies domain completeness under restrictions w.r.t. the original constraints which can be verified based on the techniques from Sec. 3.2 for verifying "standard" domain completeness w.r.t. the restricted constraints as stated in Thm. 3.9.

Proposition 3.9 (Compatibility of Restriction and Satisfaction) *Let $t\colon TG_R \to TG \in \mathscr{M}$ be a type morphism and TG_R be the restriction of type graph TG. Let ac_P be a positive nested condition over premise P and typed over TG and $\mathit{Restr}_t(ac_P)$ be its restriction along t. Given an object $(G, t_G\colon G \to TG)$ typed over TG via t_G, then it holds that $G \overset{I}{\models} ac_P \implies G \overset{I}{\models} \mathit{Restr}_t(ac_P)$. Furthermore, if $P \cong \mathit{Restr}_t(P)$, then $G \models ac_P \implies G \models \mathit{Restr}_t(ac_P)$. For a set $C = C_I \cup C_G$ of positive nested conditions typed over TG with conditions C_I (C_G) that are designated for initial (general) satisfaction and furthermore, with $P \cong \mathit{Restr}_t(P)$ for all $ac_P \in C_G$ it holds that $\mathscr{L}_I(C_I) \cap \mathscr{L}(C_G) \subseteq \mathscr{L}_I(\mathit{Restr}_t(C_I)) \cap \mathscr{L}(\mathit{Restr}_t(C_G))$ where $\mathit{Restr}_t(C) := \{\mathit{Restr}_t(c) \mid c \in C\}$.* △

Proof. We first show the result for initial satisfaction and for general satisfaction afterwards.

"$\overset{I}{\models}$" Let $G \overset{I}{\models} ac_P$, i.e., there exists $p\colon P \to G \in \mathscr{M}$ with $p \models ac_P$. By Fact 3.4 in [SEM+12], there exists $p_R\colon \mathit{Restr}_t(P) \to \mathit{Restr}_t(G) \in \mathscr{M}$ with $p_R \models \mathit{Restr}_t(ac_P)$. By the uniquness of initial morphisms, for initial object I and initial morphisms $i_1\colon I \to \mathit{Restr}_t(P)$, $i_2\colon I \to \mathit{Restr}_t(G)$ we obtain that $p_R \circ i_1 = i_2$ and therefore, $\mathit{Restr}_t(G) \overset{I}{\models} \mathit{Restr}_t(ac_P)$. By Lem. 3.16, $G \overset{I}{\models} \mathit{Restr}_t(ac_P)$.

"$\models$" $G \models ac_P \overset{Lem.\ 3.18}{\Rightarrow} \mathit{Restr}_t(G) \models \mathit{Restr}_t(ac_P) \overset{Lem.\ 3.16}{\Rightarrow} G \models \mathit{Restr}_t(ac_P)$.

Finally, $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ implies that $G \overset{I}{\models} c_I$ and $G \models c_G$ for all $c_I \in C_I, c_G \in C_G$ implying further that $G \overset{I}{\models} \mathit{Restr}_t(c_I)$ and $G \models \mathit{Restr}_t(c_G)$ for all $c_I \in C_I, c_G \in C_G$. Therefore, $G \in$

$\mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$, i.e., $\mathscr{L}_I(C_I) \cap \mathscr{L}(C_G) \subseteq \mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$. □

Beside conditions with premises that only contain elements within the restrictions, we define conditions that are already purely restricted not only for the premise but also for all conclusions, called purely restricted conditions. Lem. 3.19 and Prop. 3.10 state that a purely restricted condition and its restriction are semantically equivalent. While in Prop. 3.9 we restrict to positive conditions only, in Prop. 3.10 we restrict to possibly non-positive conditions with negations that are purely restricted and show that these conditions can be directly used for verifying domain completeness under restrictions as stated in Thm. 3.9. For example, conditions c_6 and c_7 in Fig. 3.17 is purely restricted while condition c_1 is not. Although c_1 only contains elements in its premise that are within the restriction, it is not purely restricted, since, conclusions $C_{1,1}$ and $C_{1,2}$ also contain elements that are outside the restriction.

Definition 3.34 ((Purely) Restricted Condition) Let $t\colon TG_R \to TG \in \mathscr{M}$ be a type morphism and TG_R be the restriction of type graph TG. Let ac_P be a nested condition over premise P and typed over TG. *Condition ac_P is purely restricted along t*, if for all morphisms $a\colon P' \to C'$ in ac_P and their restrictions $Restr_t(a)\colon Restr_t(P') \to Restr_t(C')$ it holds that the induced morphisms in Def. 3.31 are isomorphisms, i.e., $P' \cong Restr_t(P')$ and $C' \cong Restr_t(C')$. *Condition ac_P is restricted along t*, if $P \cong Restr_t(P)$. △

Lemma 3.19 (Compatibility of Pure Restrictions and General Satisfaction) *Let $t\colon TG_R \to TG \in \mathscr{M}$ be a type morphism and TG_R be the restriction of type graph TG. Let ac_P be a purely restricted condition along t over premise P and typed over TG and $Restr_t(ac_P)$ be its restriction over $Restr_t(P)$ along t with induced morphism $i\colon Restr_t(P) \to P$. Given an object $(G, t_G\colon G \to TG)$ typed over TG via t_G. Then, it holds that:*

1. *If there is $p\colon P \to G \in \mathscr{M}$ with $p \models ac_P$, then there is a unique morphism $p_R\colon Restr_t(P) \to G \in \mathscr{M}$ such that $p_R = p \circ i$. Furthermore, $p_R \models Restr_t(ac_P)$.*

2. *If there is $p_R\colon Restr_t(P) \to G \in \mathscr{M}$ with $p_R \models Restr_t(ac_P)$, then there is a unique morphism $p\colon P \to G \in \mathscr{M}$ such that $p_R = p \circ i$. Furthermore, $p \models ac_P$.*

△

Proof. By the construction of restrictions, (1) and (2) are pullbacks (cf. Def. 3.31). Furthermore, by assumption ac_P is purely restricted, induced morphisms t'' and i are isomorphisms with inverse isomorphisms t''^{-1} and i^{-1}. Moreover, $t'', i \in \mathscr{M}$, since, $\mathscr{M}$-morphisms $t \in \mathscr{M}$ are closed under pullbacks $(1), (2)$. We first show the result for positive conditions and for negations afterwards.

1. Let $p_R = p \circ i$. The uniqueness of p_R directly follows by definition - Assume p'_R such that $p'_R = p \circ i$, then $p'_R = p \circ i = p_R$. Furthermore, $p_R \in \mathscr{M}$ by $\mathscr{M}$-composition of $p, i \in \mathscr{M}$. By induction over the structure of nested conditions we prove that $p_R \models Restr_t(ac_P)$. **Basis.** For $ac_P = \textbf{true}$, $p_R \models Restr_t(ac_P) = \textbf{true}$. **Hypothesis.** The assumption holds for conditions ac_C and $ac_{P,i}, i \in I$ and their restrictions $Restr_t(ac_C)$ and $Restr_t(ac_{P,i})$. **Step.** For $ac_P = \exists(a\colon P \to C, ac_C)$ and its restriction $Restr_t(ac_P) = \exists(a_R\colon Restr_t(P) \to Restr_t(C), Restr_t(ac_C))$, $p \models ac_P$ implies that there is $q \in \mathscr{M}$ with $q \circ a = p$. Let $q_R = q \circ t''$ with $q_R \in \mathscr{M}$ by $\mathscr{M}$-composition of $q, t'' \in \mathscr{M}$. Thus, $p_R = p \circ i = q \circ a \circ i \overset{PB(2)}{=} q \circ t'' \circ a_R = q_R \circ a_R$. By induction hypothesis, $q_R \models Restr_t(ac_C)$. Thus, $p_R \models Restr_t(ac_P)$. For $ac_P = \wedge_{i \in I}(ac_{P,i})$, $p \models ac_P$ implies that $p \models ac_{P,i}$ for all $i \in I$. Therefore, there is a

unique $p_R \in \mathcal{M}$ such that $p_R = p \circ i$ and $p_R \models Restr_t(ac_{P,i})$ for all $i \in I$ implying further that $p_R \models Restr_t(ac_P) = \wedge_{i\in I}(Restr_t(ac_{P,i}))$. For $ac_P = \vee_{i\in I}(ac_{P,i})$ we conclude analogously.

2. Let $p = p_R \circ i^{-1}$. By class $\mathcal{M}$ is closed under isomorphisms, $i^{-1} \in \mathcal{M}$ and therefore, by $\mathcal{M}$-composition with assumption $p_R \in \mathcal{M}$ it follows that $p \in \mathcal{M}$. Furthermore, $p \circ i = p_R \circ i^{-1} \circ i \overset{i \text{ is iso}}{=} p_R \circ id_{Restr_t(P)} = p_R$. The uniqueness of p concludes as follows. Assume $p' \in \mathcal{M}$ with $p_R = p' \circ i$. Then, $p' = p' \circ id_P \overset{i \text{ is iso}}{=} p' \circ i \circ i^{-1} = p_R \circ i^{-1} = p \circ i \circ i^{-1} \overset{i \text{ is iso}}{=} p \circ id_P = p$. By induction over the structure of nested conditions we prove that $p \models ac_P$. **Basis.** For $ac_P =$ **true**, $p \models ac_P$. **Hypothesis.** The assumption holds for conditions ac_C and $ac_{P,i}, i \in I$ and their restrictions $Restr_t(ac_C)$ and $Restr_t(ac_{P,i})$. **Step.** For $ac_P = \exists(a\colon P \to C, ac_C)$ and its restriction $Restr_t(ac_P) = \exists(a_R\colon Restr_t(P) \to Restr_t(C), Restr_t(ac_C))$, $p_R \models Restr_t(ac_P)$ implies that there is $q_R \in \mathcal{M}$ with $q_R \circ a_R = p_R$ and $q_R \models Restr_t(ac_C)$. Let $q = q_R \circ t''^{-1}$ with $q \in \mathcal{M}$ by $\mathcal{M}$-composition of $q_R, t''^{-1} \in \mathcal{M}$ where $t''^{-1} \in \mathcal{M}$, since, class $\mathcal{M}$ is closed under isomorphisms. Furthermore, $t''^{-1} \circ a = t''^{-1} \circ a \circ id_P \overset{i \text{ is iso}}{=} t''^{-1} \circ a \circ i \circ i^{-1} \overset{PB(2)}{=} t''^{-1} \circ t'' \circ a_R \circ i^{-1} \overset{t'' \text{ is iso}}{=} id_{Restr_t(C)} \circ a_R \circ i^{-1} = a_R \circ i^{-1}$ $(*^1)$. Thus, $p = p_R \circ i^{-1} = q_R \circ a_R \circ i^{-1} = \overset{(*^1)}{=} q_R \circ t''^{-1} \circ a = q \circ a$. Moreover, $q \circ t'' = q_R \circ t''^{-1} \circ t'' \overset{t'' \text{ is iso}}{=} q_R \circ id_{Restr_t(C)} = q_R$. By induction hypothesis, $q \models ac_C$ and $p \models ac_P$. For $ac_P = \wedge_{i\in I}(ac_{P,i})$ and its restriction $Restr_t(ac_P) = \wedge_{i\in I}(Restr_t(ac_{P,i}))$, $p_R \models Restr_t(ac_P)$ implies that $p_R \models Restr_t(ac_{P,i})$ for all $i \in I$. For each $i \in I$, there is $p_i \in \mathcal{M}$ with $p_i \models ac_{P,i}$ and $p_R = p_i \circ i$. By the uniqueness of p_i, it follows that $p_1 = \ldots = p_n$ and therefore, there is $p = p_1 \in \mathcal{M}$ with $p \models ac_P$. For $ac_P = \vee_{i\in I}(ac_{P,i})$ we conclude analogously.

$$\begin{tikzcd}
 & & ac_C & \\
 & TG & C \arrow[l, "t_C"'] \arrow[dd, bend left, "t''^{-1}"] \arrow[lld, bend right, "q"'] & P \arrow[l, "a"'] \arrow[dd, bend left, "i^{-1}"] \arrow[llld, bend right=40, "p"'] \\
G \arrow[ru, "t_G"] & (1) & & (2) \\
 & TG_R \arrow[uu, "t"] & Restr_t(C) \arrow[l, "t_{C_R}"] \arrow[uu, bend left, "t''"] \arrow[llu, bend left, "q_R"] & Restr_t(P) \arrow[l, "a_R"] \arrow[uu, bend left, "i"] \arrow[lllu, bend left=40, "p_R"] \\
 & & Restr_t(ac_C) &
\end{tikzcd}$$

For $ac_P = \neg ac'_P$ we conclude as follows.

1. Assumption $p \models ac_P$ implies that $\neg(p \models ac'_P)$. Assume that there is $p_R \in \mathcal{M}$ with $p_R \models Restr_t(ac'_P)$ and $p_R = p \circ i$. Then by the case for positive conditions, $p \models ac'_P$ contradicting with the assumption. Thus, p_R can be constructed as shown before and $\neg(p_R \models Restr_t(ac'_P))$, i.e., $p_R \models Restr_t(ac_P) = Restr_t(\neg ac'_P) = \neg Restr_t(ac'_P)$.

2. Assumption $p_R \models Restr_t(ac_P)$ implies that $\neg(p_R \models Restr_t(ac'_P))$. Assume that there is $p \in \mathcal{M}$ with $p \models ac'_P$ and $p_R = p \circ i$. Then by the case for positive conditions, $p_R \models Restr_t(ac'_P)$ contradicting with the assumption. Thus, p can be constructed as shown before and $\neg(p \models ac'_P)$, i.e., $p \models ac_P = \neg ac'_P$.

□

Proposition 3.10 (Compatibility of Pure Restrictions and Satisfaction) *Let $t\colon TG_R \to TG \in \mathcal{M}$ be a type morphism and TG_R be the restriction of type graph TG. Let ac_P be a purely restricted condition over premise P and typed over TG and $Restr_t(ac_P)$ be its restriction along t. Given an object $(G, t_G\colon G \to TG)$ typed over TG via t_G, then it holds that $G \overset{I}{\models} ac_P$ if and only if $G \overset{I}{\models} Restr_t(ac_P)$ and furthermore, $G \models ac_P$ if and only if $G \models Restr_t(ac_P)$. For a set $C = C_I \cup C_G$ of purely restricted conditions along t and typed over TG with conditions C_I (C_G) that are designated for initial (general) satisfaction it holds that $\mathcal{L}_I(C_I) \cap \mathcal{L}(C_G) = \mathcal{L}_I(Restr_t(C_I)) \cap \mathcal{L}(Restr_t(C_G))$ where $Restr_t(C) := \{Restr_t(c) \mid c \in C\}$.* △

Proof. We first show the result for initial satisfaction and for general satisfaction afterwards. Let condition $Restr_t(ac_P)$ over $Restr_t(P)$ be the restriction of ac_P.

"$\overset{I}{\models}$" $G \overset{I}{\models} ac_P \overset{Rem.~2.10}{\Leftrightarrow} \exists p\colon P \to G \in \mathcal{M}, p \models ac_P \overset{Lem.~3.19}{\Leftrightarrow} \exists p_R\colon Restr_t(P) \to G \in \mathcal{M}, p_R \models Restr_t(ac_P) \overset{Rem.~2.10}{\Leftrightarrow} G \overset{I}{\models} Restr_t(ac_P)$.

"$\models$" $G \models ac_P \overset{Rem.~2.10}{\Leftrightarrow} \forall p\colon P \to G \in \mathcal{M}, p \models ac_P \overset{(*^1)}{\Leftrightarrow} \forall p_R\colon Restr_t(P) \to G \in \mathcal{M}, p_R \models Restr_t(ac_P) \overset{Rem.~2.10}{\Leftrightarrow} G \models Restr_t(ac_P)$. It remains to prove $(*^1)$. "$\Rightarrow$" Assume that there exists $p_R \in \mathcal{M}$ with $\neg(p_R \models Restr_t(ac_P))$, i.e., $p_R \models \neg(Restr_t(ac_P)) = Restr_t(\neg ac_P)$. By Lem. 3.19 and Item 2, there is $p\colon P \to G \in \mathcal{M}$ with $p \models \neg ac_P$, i.e., $\neg(p \models ac_P)$, which contradicts with the assumption. "$\Leftarrow$" Conversely, assume that there exists $p \in \mathcal{M}$ with $\neg(p \models ac_P)$, i.e., $p \models \neg ac_P$. By Lem. 3.19 and Item 1, there is $p_R\colon Restr_t(P) \to G \in \mathcal{M}$ with $p_R \models Restr_t(\neg ac_P) = \neg Restr_t(ac_P)$, i.e., $\neg(p_R \models Restr_t(ac_P))$, which contradicts with the assumption.

It remains to prove that $\mathcal{L}_I(C_I) \cap \mathcal{L}(C_G) \subseteq \mathcal{L}_I(Restr_t(C_I)) \cap \mathcal{L}(Restr_t(C_G))$ and conversely, $\mathcal{L}_I(Restr_t(C_I)) \cap \mathcal{L}(Restr_t(C_G)) \subseteq \mathcal{L}_I(C_I) \cap \mathcal{L}(C_G)$. $G \in \mathcal{L}_I(C_I) \cap \mathcal{L}(C_G) \Leftrightarrow G \overset{I}{\models} c_I, \forall c_I \in$ C$_I$ and $G \models c_G, \forall c_G \in C_G \Leftrightarrow G \overset{I}{\models} Restr_t(c_I), \forall c_I \in$ C$_I$ and $G \models Restr_t(c_G), \forall c_G \in C_G \Leftrightarrow G \in \mathcal{L}_I(Restr_t(C_I)) \cap \mathcal{L}(Restr_t(C_G))$. □

Finally, Props. 3.9 and 3.10 lead to the main result in Thm. 3.9 concerning the verification of domain completeness under restrictions. Given a set of domain constraints C, then all non-positive constraints with negations that are not purely restricted and all positive constraints that are designated for general satisfaction but are unrestricted are neglected for the verification. All other domain constraints can be restricted such that (a successful verification of) domain completeness w.r.t. the restricted constraints implies domain completeness under restrictions w.r.t. the original domain constraints C which enables the application of the verification techniques in Sec. 3.2 for verifying domain completeness under restrictions. The main result is used in Sec. 5.1 for verifying the completeness of the translation of programs written in an object-oriented programming language into UML class diagrams where the TGG is only defined on the class definitions in programs while all other syntactic aspects of programs are neglected and it must be ensured that all class definitions an all possible programs over the programming language can be translated into corresponding classes, attributes and inheritance relations in class diagrams. We are confident that the approach can be extended in future work such that the neglected constraints are also be used in the verification, e.g., in the definition of C-inconsistent graphs in Def. 3.4.

Theorem 3.9 (Domain Completeness under Restrictions) *Let $t\colon TG_R \to TG \in \mathcal{M}$ be a type morphism and TG_R be the restriction of type graph TG. Let $C = C_I \cup C_G$ be* *a set of nested conditions (domain constraints) typed over TG with conditions C_I (C_G) that are designated for initial (general) satisfaction and let $Restr_t(C_I) := \{Restr_t(c) \mid c \in C_I, c \text{ is purely restricted along } t \text{ OR } c \text{ is positive}\}$ and $Restr_t(C_G) := \{Restr_t(c) \mid c \in C_G, c \text{ is purely restricted along } t \text{ OR } c \text{ is positive and restricted along } t\}$ be the corresponding sets of restricted conditions along t. Let GG be a graph grammar typed over TG_R. Then, it holds that:*

1. *Domain completeness under restrictions w.r.t. $\mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$ and $\mathscr{L}(GG)$ implies domain completeness under restrictions w.r.t. $\mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ and $\mathscr{L}(GG)$ in the sense of Def. 3.33.*

2. *Domain completeness w.r.t. $\mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$, $\mathscr{L}(GG)$ and TG_R in the sense of Def. 3.2 implies domain completeness under restrictions w.r.t. $\mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$ and $\mathscr{L}(GG)$.* △

Proof.

1. Let $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$. Thus, $G \overset{I}{\models} c_I$ for all $c_I \in C_I$ and $G \models c_G$ for all $c_G \in C_G$. Let $C_I^P = \{c \mid c \in \mathrm{C}_I, c \text{ is positive}\}$ be the set of positive conditions for initial satisfaction, $C_I^N = \{c \mid c \in \mathrm{C}_I, c \text{ is non-positive AND purely restricted along } t\}$ be the set of non-positive conditions for initial satisfaction and analogously for general satisfaction $C_G^P = \{c \mid c \in C_G, c \text{ is positive and restricted along } t\}$ and $C_G^N = \{c \mid c \in C_G, c \text{ is non-positive AND purely restricted along } t\}$. Thus, $C_I^P \cup C_I^N \subseteq C_I$ and $C_G^P \cup C_G^N \subseteq C_G$. Therefore, $G \overset{I}{\models} c_I$ for all $c_I \in C_I^P \cup C_I^N$ and $G \models c_G$ for all $c_G \in C_G^P \cup C_G^N$, i.e., $G \in \mathscr{L}_I(C_I^P) \cap \mathscr{L}(C_G^P)$ and $G \in \mathscr{L}_I(C_I^N) \cap \mathscr{L}(C_G^N)$. By Prop. 3.9, $G \in \mathscr{L}_I(Restr_t(C_I^P)) \cap \mathscr{L}(Restr_t(C_G^P))$. Furthermore by Prop. 3.10, $G \in \mathscr{L}_I(Restr_t(C_I^N)) \cap \mathscr{L}(Restr_t(C_G^N))$. Therefore, $G \in \mathscr{L}_I(Restr_t(C_I^P) \cup Restr_t(C_I^N)) \cap \mathscr{L}(Restr_t(C_G^P) \cup Restr_t(C_G^N)) = \mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$. By the assumption that domain completeness under restrictions w.r.t. $\mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$ and $\mathscr{L}(GG)$ holds, it follows that $Restr_t(G) \in \mathscr{L}(GG)$, i.e., domain completeness under restrictions w.r.t. $\mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ and $\mathscr{L}(GG)$ holds.

2. Let $G \in \mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$, i.e., $G \overset{I}{\models} Restr_t(c_I)$ for all $Restr_t(c_I) \in Restr_t(C_I)$ and $G \models Restr_t(c_G)$ for all $Restr_t(c_G) \in Restr_t(C_G)$. By Lem. 3.16, $Restr_t(G) \overset{I}{\models} Restr_t(c_I)$ for all $Restr_t(c_I) \in Restr_t(C_I)$ and $Restr_t(G) \models Restr_t(c_G)$ for all $Restr_t(c_G) \in Restr_t(C_G)$. Thus, $Restr_t(G) \in \mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$. By the assumption that domain completeness w.r.t. $\mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$ and $\mathscr{L}(GG)$ holds, it follows that $Restr_t(G) \in \mathscr{L}(GG)$, i.e., domain completeness under restrictions w.r.t. $\mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$ and $\mathscr{L}(GG)$ holds.

□

In Sec. 4.1 and Cor. 4.1, we instantiate the result from Thm. 3.9 to the domain completeness of model transformations under restrictions and show in Sec. 4.1 and Ex. 4.1 that the transformation as defined by the *TGG* in Fig. 3.16 is domain complete under restrictions if constraint c_8 from Fig. 3.18 is added to the domain constraints in Fig. 3.17. This is necessary in order to obtain *C*-extension completeness for the example according to Sec. 3.2 and Def. 3.8, i.e., to obtain extensions that can be created via the rules of the given *TGG*. Constraint c_8 is not contained in

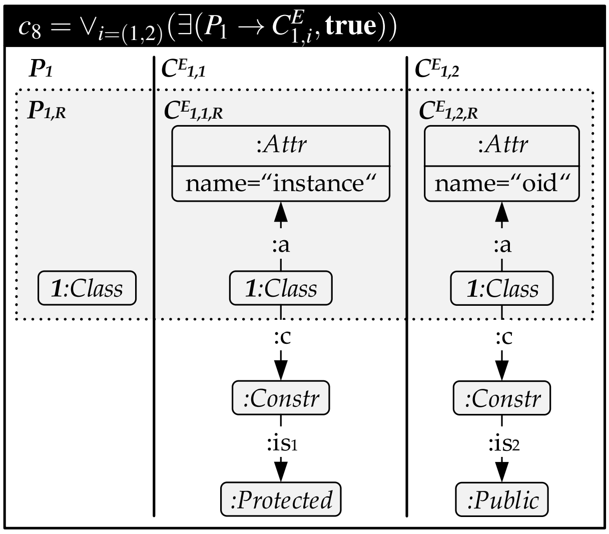


Figure 3.18: Extended Domain Graph Constraint with Restriction

the initial set of domain constraints in Fig. 3.17. However, the constraint can be inferred from the initial set by performing C-extensions of the conclusion graphs of initial domain constraint c_1 via constraints c_2 and c_3 according to Sec. 3.2 and Def. 3.5. As in the example, the extension of domain constraints is useful for the verification of domain completeness under restrictions if the verification is based on the verification of domain completeness w.r.t. restricted domain constraints according to Thm. 3.9 and C-extension completeness cannot be successfully verified based on the initial set of restricted domain constraints but based on an extended set that additionally contains constraints that are inferred from initial constraints via elements outside the restriction, as it is the case for constraint c_8 where class constructors and their visibilities are taken into account for the inference but which are omitted in the restricted constraints. Therefore, we propose to extend the initial set of domain constraints at first and then to use the refined set for verifying domain completeness under restrictions. In Def. 3.35, we define the step-wise extension of conditions and show in Thm. 3.10 that a graph satisfies the initial constraints if and only if it does also satisfy the extended constraints implying further that the language over the initial set of domain constraints equals to the language over the refined set. Therefore, the refined set of constraints can be used instead of the initial set when verifying domain completeness. Note that we restrict to extensions of plain conditions without multiple nestings. However, we are confident that the approach can be extended to conditions with arbitrary nestings such that Thm. 3.10 holds, in future work.

Definition 3.35 (C-Extensions of Conditions) Let ac_P be a plain condition and C be a set of conditions. *The extensions $Extensions(ac_P, C)$ of ac_P via C* is inductively defined as follows based on the C-extensions of conclusion graphs according to Sec. 3.2 and Def. 3.5:
$Extensions(ac_P, C) = \{\mathbf{true}\}$, for $ac_P = \mathbf{true}$,
$Extensions(ac_P, C) = \{\vee_{i\in\{1,\dots,n\}}(\exists(P \xrightarrow{e_i \circ a} E_i, \mathbf{true})) \mid$, for $ac_P = \exists(a\colon P \to C', \mathbf{true})$,
$E = \{E_1, \dots, E_n\} \in Extensions(C', C)\}$ where $e_i\colon C' \to E_i$
is the induced morphism according to Sec. 3.2 and Def. 3.5
$Extensions(ac_P, C) = \{\vee_{i\in I}(e_{P,i}) \mid e_{P,i} \in Extensions(ac_{P,i}, C)\}$, for $ac_P = \vee_{i\in I}(ac_{P,i})$,
$Extensions(ac_P, C) = \{\wedge_{i\in I}(e_{P,i}) \mid e_{P,i} \in Extensions(ac_{P,i}, C)\}$, for $ac_P = \wedge_{i\in I}(ac_{P,i}), and$
$Extensions(ac_P, C) = \{\neg e_P \mid e_P \in Extensions(ac'_P, C)\}$, for $ac_P = \neg ac'_P$.

△

Example 3.22 (C-Extensions of Conditions) *The condition in Fig. 3.18 is an extension of condition c_1 in Fig. 3.17 via conditions c_2 and c_3.* △

Note that in Thm. 3.10, conditions are extended via conditions that are designated for general satisfaction only. If we allow the extension of conditions via conditions that are designated for initial satisfaction, then Thm. 3.10 may not hold. For example, consider two constraints "Each node : A is connected to two : B nodes" and "There exists a : B node connected to a : C node". The extension of the first constraint via the second constraint may lead to a constraint "Each node : A is connected to two : B nodes and furthermore, each of the two : B nodes is connected to a : C node". Therefore, the existential characteristic of the : B node in the second constraint switches to a universal characteristic in the extended constraint. Thus, a graph which satisfies the first initial constraint may not satisfy the extended constraint.

Theorem 3.10 (Equivalence of Languages over Constraints & Extended Constraints) *In an $\mathscr{M}$-adhesive category with effective pushouts, let $C_I \cup C_G$ be a set of constraints where constraints C_I are designated for initial satisfaction and constraints C_G are designated for general satisfaction. Furthermore, let C'_I (C'_G) be a set of extensions of conditions C_I (C_G) via C_G. Then, it holds that $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ if and only if $G \in \mathscr{L}_I(C_I \cup C'_I) \cap \mathscr{L}(C_G \cup C'_G)$. Therefore, $\mathscr{L}_I(C_I) \cap \mathscr{L}(C_G) = \mathscr{L}_I(C_I \cup C'_I) \cap \mathscr{L}(C_G \cup C'_G)$.* △

Proof. The proof is presented in appendix A.15. □

Chapter 4

Domain Completeness of Model Transformations & Model Synchronisations

4.1 Domain Completeness of Model Transformations

Given a model transformation $MT\colon \mathscr{L}(C^S) \Rrightarrow \mathscr{L}(D_2)$ from source DSL $\mathscr{L}(C^S)$ that is restricted by source domain graph constraints C^S to target DSL $\mathscr{L}(D_2)$ in target domain D_2, then MT is domain complete if each source model $G^S \in \mathscr{L}(C^S)$ can be completely translated via a model transformation sequence in the sense that all elements of the graph are translated exactly once by changing their translation attributes from **F** to **T**.

General Assumption: As with the results for verifying domain completeness in Sec. 3.2, we assume that the results of this Chapter are applied in the context $(\mathbf{ATrGraphs}_{ATGI}, \mathscr{M})$ of typed, attributed triple graphs with node type inheritance and common triple type graph $ATGI$. This is due to the fact that the node and edge attributes are used as markings (translation attributes) for checking conflict-freeness of rules. Moreover, translation attributes are used for an intuitive definition of domain completeness of model transformations in Def. 4.1. Note that according to Sec. 3.1 and Def. 3.1, we write C short for a set of constraints $C = C_I \cup C_G$ that is composed of constraints C_I that are designated for initial satisfaction and constraints C_G that are designated for general satisfaction. Analogously, we write $\mathscr{L}(C)$ short for $\mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$.

Definition 4.1 (Domain Completeness of Model Transformations) Let C^S be the set of source domain graph constraints and $\mathscr{L}(C^S)$ be the source domain-specific language of graphs. Furthermore, let TGG be a triple graph grammar that specifies the translation $MT\colon \mathscr{L}(C^S) \Rrightarrow \mathscr{L}(D_2)$ of graphs in $\mathscr{L}(C^S)$ into graphs in the target domain-specific language $\mathscr{L}(D_2)$. *The model transformation MT is domain complete*, if for each graph $G^S \in \mathscr{L}(C^S)$ there is a model transformation sequence $(G^S, G_0 \xRightarrow{tr_{FT}^*} G_n, G^T)$ based on the forward translation rules of TGG with $G_0 = (Att^{\mathbf{F}}(G^S) \leftarrow \varnothing \rightarrow \varnothing)$ and $G_n = (Att^{\mathbf{T}}(G^S) \leftarrow G^C \rightarrow G^T)$. △

Based on the "classical" syntactical completeness and correctness of model transformations by TGGs based on forward translation rules (cf. Sec. 2.4 and Rem. 2.20), the domain completeness of model transformations from Def. 4.1 can be redefined as follows. While Def. 4.1

reflects the intuitive meaning behind complete transformations, Thm. 4.1 expresses completeness in terms of a language inclusion which can be verified by using the verification techniques for domain completeness. Therefore, both formulations of completeness in Def. 4.1 and Thm. 4.1 are equivalent.

Theorem 4.1 (Domain Completeness of Model Transformations) *Let $MT\colon \mathscr{L}(C^S) \Rrightarrow \mathscr{L}(D_2)$ be a model transformation based on forward translation rules of a given TGG. Transformation MT is domain complete according to Def. 4.1 if and only if domain completeness w.r.t. $\mathscr{L}(C^S)$ and $\mathscr{L}(TGG)^S$ holds in the sense of Sec. 3.1 and Def. 3.2, i.e., $\mathscr{L}(C^S) \subseteq \mathscr{L}(TGG)^S$.* △

Proof. **"⇒"** For each $G^S \in \mathscr{L}(C^S)$ with corresponding model transformation sequence based on forward translation rules the "classical" correctness implies that $G^S \in \mathscr{L}(TGG)^S$ (cf. Sec. 2.4 and Rem. 2.20).

"⇐" By "classical" completeness, for each $G^S \in \mathscr{L}(C^S) \cap \mathscr{L}(TGG)^S$ there is a model transformation sequence $(G^S, G_0 \xRightarrow{tr^*_{FT}} G_n, G^T)$ based on forward translation rules with $G_0 = (Att^{\mathbf{F}}(G^S) \leftarrow \varnothing \rightarrow \varnothing)$ and $G_n = (Att^{\mathbf{T}}(G^S) \leftarrow G^C \rightarrow G^T)$ (cf. Sec. 2.4 and Rem. 2.20). □

Similarly to the verification of domain completeness for flat graphs in Sec. 3.2 and Thm. 3.3, we verify the language inclusion $\mathscr{L}(C^S) \subseteq \mathscr{L}(TGG)^S$ by verifying C^S-conflict-freeness from Thm. 3.3 w.r.t. forward translation rules TR_{FT}, since, we only need to check for possible overlappings of extensions in the source domain. Therefore, similarly to Sec. 3.2 and Def. 3.12, we define the C^S-conflict-freeness of forward translation rules in Def. 4.2. Note that in C^S-conflict-freeness of forward translation rules, only critical pairs need to be considered where the source component of the conflict triple graph O occurs in graphs of $\mathscr{L}(C^S)$ (cf. Def. 4.2 and Item 1) and conflict triple graph O can be embedded in a triple graph O' that can be created by a forward translation sequence (cf. Def. 4.2 and Item 2).

Definition 4.2 (C^S-Conflict-Freeness of Forward Translation Rules) Let C^S be the source domain constraints and TR_{FT} be the forward translation rules of triple rules TR. Then, TR_{FT} *is C^S-conflict-free*, if for each critical pair $(K_1 \xLeftarrow{(tr_{FT,1},o_1)} O = (O^S \leftarrow O^C \rightarrow O^T) \xRightarrow{(tr_{FT,2},o_2)} K_2)$ with $tr_{FT,1}, tr_{FT,2} \in TR_{FT}$ where

1. O^S is significant w.r.t. $\mathscr{L}(C^S)$ (or not C^S-inconsistent), and
2. there is $O \rightarrow O' \in \mathscr{M}$ and forward translation sequence $(Att^{\mathbf{F}}(\overline{O}) \leftarrow \varnothing \rightarrow \varnothing) \xRightarrow{tr^*_{FT}} O'$ via TR_{FT},

it is true that the rules and matches are the same ($tr_{FT,1} = tr_{FT,2}, o_1 = o_2$) (or it is true that the critical pair is strictly confluent). △

Theorem 4.2 (Domain Completeness of Model Transformations) *Let C^S be the source domain constraints in $\mathscr{M}$-normal form, $\mathscr{L}(C^S)$ be the source domain-specific language over C^S and $\mathscr{L}(TGG)$ be the language over a non-deleting triple graph grammar $TGG = (\varnothing, TR)$ with empty triple start graph $\varnothing$, all triple productions TR being non-trivial and where all application conditions are in $\mathscr{M}$-normal form. Let TR_{FT} be the derived set of forward translation rules from TGG. Furthermore, let MT be a model transformation based on forward translation rules TR_{FT}. If the rules TR_{FT} are C^S-conflict-free and $\mathscr{L}(TGG')^S$ is C^S-extension complete where $TGG' = (\varnothing', TR)$ with $\varnothing'$ being the empty triple start graph with DSIG-term algebra $T_{DSIG}(X)$,* *then domain completeness w.r.t.* $\mathscr{L}(C^S)$ *and* $\mathscr{L}(TGG)^S$ *holds for almost injective matches, i.e., it holds that* $\mathscr{L}(C^S) \subseteq \mathscr{L}(TGG)^S$. △

Proof. The proof is basically identical to the proof of Thm. 3.3. Let $G \in \mathscr{L}(C^S)$, $i\colon G_A \to G$ be an instance morphism and $A = (a_i)_{i\in\{1,\dots,n\}} \in Atoms(G_A)$ be the atoms of G_A. Analogously to Thm. 3.3, by $\forall a \in A.a \in EAtoms(C^S)$ and $\mathscr{L}(TGG')^S$ is C^S-extension complete it follows that $\forall a \in A.\exists S_a \in Extensions(a, C^S_G)$ such that $S_a \subseteq \mathscr{L}(TGG')^S$ by Def. 3.8. Thus, $\forall a \in A.\exists S_a \in Extensions(a, C^S_G)$ such that $\forall s \in S_a.\exists$ triple graph $(s \leftarrow s^C \to s^T) \in \mathscr{L}(TGG')$ and model transformation sequence $(s, (Att^{\mathbf{F}}(s) \leftarrow \varnothing \to \varnothing) \xRightarrow{tr^*_{FT}} (Att^{\mathbf{T}}(s) \leftarrow s^C \to s^T), s^T)$ based on forward translation rules TR_{FT} by model transformation MT and Sec. 2.4 and Rem. 2.20. Note that for each such model transformation sequence and a given injective embedding $f\colon s \to s' \in \mathscr{M}$ there is a forward translation sequence $(Att^{\mathbf{F}}(s') \leftarrow \varnothing \to \varnothing) \xRightarrow{tr^*_{FT}} (s' \oplus Att^{\mathbf{T}}_s \oplus Att^{\mathbf{F}}_{s'\setminus s} \leftarrow s^C \to s^T)$ via forward translation rules TR_{FT} analogously to Sec. 3.2 and Lem. 3.2 $(^{*A})$, since, all relevant elements of application conditions in rules TR_{FT} are marked with $\mathbf{T}$ whereas the marking of all elements in $s' \setminus s$ remain $\mathbf{F}$ for the translation sequence and therefore, cannot be matched by the application conditions. Analogously to the proof of Thm. 3.3, there is a function f_{a_E} with $f_{a_E}(a) = s \in S_a, \forall a \in A = (a_i)_{i\in\{1,\dots,n\}}$ such that there exist graphs $(G^E_j)_{1\le j\le n-1}$ and pushouts $(PO^E_k +_{G^E_k} f_{a_E}(a_{k+1}) = PO^E_{k+1})_{k\in\{1,\dots,n-1\}}$ with pushout objects PO^E_{k+1}, $PO^E_1 = f_{a_E}(a_1)$ and injective embeddings $i_{k,1}\colon PO^E_k \to PO^E_{k+1}$ and $i_{k,2}\colon f_{a_E}(a_{k+1}) \to PO^E_{k+1} \in \mathscr{M}$ where $PO^E_n = G_A$ by Sec. 3.2 and Lem. 3.7. For pushout $k = 1$ we conclude as follows. By $(*^A)$, there exists forward translation sequences $t_1\colon (Att^{\mathbf{F}}(PO^E_2) \leftarrow \varnothing \to \varnothing) \xRightarrow{tr^*_{FT}} (PO^E_2 \oplus Att^{\mathbf{T}}_{f_{a_E}(a_1)} \oplus Att^{\mathbf{F}}_{PO^E_2\setminus f_{a_E}(a_1)} \leftarrow f_{a_E}(a_1)^C \to f_{a_E}(a_1)^T)$ and $t_2\colon (Att^{\mathbf{F}}(PO^E_2) \leftarrow \varnothing \to \varnothing) \xRightarrow{tr^*_{FT}} (PO^E_2 \oplus Att^{\mathbf{T}}_{f_{a_E}(a_2)} \oplus Att^{\mathbf{F}}_{PO^E_2\setminus f_{a_E}(a_2)} \leftarrow f_{a_E}(a_2)^C \to f_{a_E}(a_2)^T)$ via forward translation rules TR_{FT}. Analogously to the proof of Thm. 3.3, assumption TR_{FT} is C^S-conflict-free implies that transformation system TR_{FT} is confluent and therefore, there is a complete forward translation sequence $(Att^{\mathbf{F}}(PO^E_2) \leftarrow \varnothing \to \varnothing) \xRightarrow{tr^*_{FT}} (Att^{\mathbf{T}}(PO^E_2) \leftarrow C_2 \to T_2)$, i.e., a model transformation sequence $(PO^E_2, (Att^{\mathbf{F}}(PO^E_2) \leftarrow \varnothing \to \varnothing) \xRightarrow{tr^*_{FT}} (Att^{\mathbf{T}}(PO^E_2) \leftarrow C_2 \to T_2), T_2)$ based on forward translation rules TR_{FT} for pushout object PO^E_2. Note that the restriction to a subset of all critical pairs in C^S-conflict-freeness matches the situation, since, $G^A \in \mathscr{L}(C^S)$ and $\mathscr{M}$-composition of embeddings $i_{k,1}, i_{k,2}$ imply that the source component of all conflict triple graphs of conflicts that may occur is significant w.r.t. $\mathscr{L}(C^S)$ (cf. Def. 4.2 and Item 1) and furthermore, all such conflict triple graphs of conflicts that may occur are embedded in triple graphs that are created by forward translation sequences t_1 and t_2 (cf. Def. 4.2 and Item 2). Analogously, we iterate over all pushouts for $k = (1, \dots, n-1)$ and obtain a model transformation sequence $(PO^E_n, (Att^{\mathbf{F}}(PO^E_n) \leftarrow \varnothing \to \varnothing) \xRightarrow{tr^*_{FT}} (Att^{\mathbf{T}}(PO^E_n) \leftarrow C_n \to T_n), T_n)$ based on forward translation rules TR_{FT} and almost injective matches. By Sec. 2.4 and Rem. 2.20, $PO^E_n = G_A \in \mathscr{L}(TGG')^S$, i.e., there is $\varnothing' \overset{*}{\Rightarrow} (G_A \leftarrow C_A \to T_A)$ via TR. We extend instance morphism $i\colon G_A \to G$ to a triple graph morphism $i'\colon (G_A \leftarrow C_A \to T_A) \to (G \leftarrow C \to T)$ that is an instance morphism componentwise for source, correspondence and target. Analogously to the proof of Thm. 3.3 by Lem. 3.15 componentwise with instance morphism i' and all application conditions in TR being in $\mathscr{M}$-normal form there is a transformation $\varnothing \overset{*}{\Rightarrow} (G \leftarrow C \to T)$ via TR and almost injective matches, i.e., $G \in \mathscr{L}(TGG)^S$. Therefore, $\mathscr{L}(C^S) \subseteq \mathscr{L}(TGG)^S$. □

According to domain completeness under restrictions in Sec. 3.5, we define domain completeness of model transformations under restrictions. This is important for model transformations that only cover specific elements of each graph in the source DSL. For example, the model transformation in Sec. 3.5 from class diagrams (CDs) to relational database models (RDBMs) only covers the elements of the restricted type graph $TG_{CD,R}$ in Fig. 3.16, i.e., classes and their attributes in class diagrams, while neglecting class constructors and their visibilities. In contrast to "full" domain completeness of the transformation which requires that all class diagrams can be completely translated into RDBMs, domain completeness under restrictions only requires that all classes and their attributes in each class diagram can be completely translated into corresponding tables and columns.

Definition 4.3 (Domain Completeness of Model Transformations under Restrictions) Let $t\colon TG^S_R \to TG^S \in \mathscr{M}$ be a type morphism and TG^S_R be the restriction of source type graph TG^S. Let $TG = (TG^S_R \leftarrow TG^C \rightarrow TG^T)$ be a triple type graph. Let C^S be the set of source domain graph constraints typed over TG^S and $\mathscr{L}(C^S)$ be the source domain-specific language of graphs. Furthermore, let TGG be a triple graph grammar typed over TG that specifies the translation $MT\colon \mathscr{L}(C^S) \Rrightarrow \mathscr{L}(D_2)$ of graphs in $\mathscr{L}(C^S)$ into graphs in the target domain-specific language $\mathscr{L}(D_2)$. *The model transformation MT is domain complete under restrictions*, if for the restriction $Restr_t(G^S)$ of each graph $G^S \in \mathscr{L}(C^S)$ along t there is a model transformation sequence $(Restr_t(G^S), G_0 \xRightarrow{tr^*_{FT}} G_n, G^T)$ based on the forward translation rules of TGG with $G_0 = (Att^{\mathbf{F}}(Restr_t(G^S)) \leftarrow \varnothing \rightarrow \varnothing)$ and $G_n = (Att^{\mathbf{T}}(Restr_t(G^S)) \leftarrow G^C \rightarrow G^T)$. △

Similarly to Thm. 4.1, in Thm. 4.3 we reformulate domain completeness of model transformations under restrictions by domain completeness under restrictions in the sense of Def. 3.33 which can be verified by verifying domain completeness w.r.t. restricted constraints as stated in Cor. 4.1. In turn, domain completeness can be verified based on Thm. 4.2.

Theorem 4.3 (Domain Completeness of Model Transformations under Restrictions) *Let $MT\colon \mathscr{L}(C^S) \Rrightarrow \mathscr{L}(D_2)$ be a model transformation based on forward translation rules of a given TGG. Transformation MT is domain complete under restrictions according to Def. 4.3 if and only if domain completeness under restrictions w.r.t. $\mathscr{L}(C^S)$ and $\mathscr{L}(TGG)^S$ holds in the sense of Def. 3.33.* △

Proof. Let $G^S \in \mathscr{L}(C^S)$ and $Restr_t(G^S)$ be its restriction along t.

"⇒" By assumption, there is a model transformation sequence $(Restr_t(G^S), G_0 \xRightarrow{tr^*_{FT}} G_n, G^T)$ based on forward translation rules. By "classical" syntactical correctness of model transformations based on foward translation rules, $Restr_t(G^S) \in \mathscr{L}(TGG)^S$.

"⇐" By assumption, $Restr_t(G^S) \in \mathscr{L}(TGG)^S$. By "classical" syntactical completeness of model transformations based on forward translation rules, there is a model transformation sequence $(Restr_t(G^S), G_0 \xRightarrow{tr^*_{FT}} G_n, G^T)$ based on forward translation rules.

□

Corollary 4.1 (Domain Completeness of Model Transformations under Restrictions) *In the context of Thm. 4.3, domain completeness under restrictions w.r.t. $\mathscr{L}(C^S)$ and $\mathscr{L}(TGG)^S$ holds if according to Thm. 3.9 domain completeness w.r.t. $\mathscr{L}_I(Restr_t(C^S_I)) \cap \mathscr{L}(Restr_t(C^S_G))$, $\mathscr{L}(TGG)^S$ and TG^S_R holds with $C^S = C^S_I \cup C^S_G$.* △

Proof. This follows directly by Thm. 3.9 and Items 1 and 2. □

Based on the reformulations of domain completeness of model transformations (under restrictions) in Thms. 4.1 and 4.3 and the result from Cor. 4.1, the following relationship between the notions of domain completeness and domain completeness under restrictions does exist.

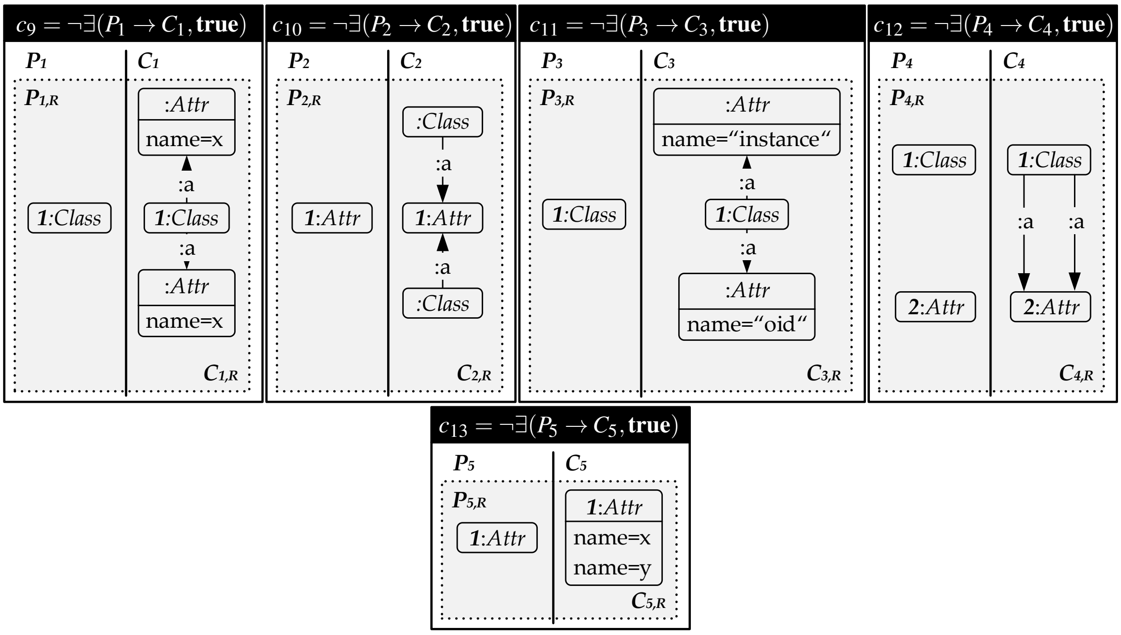
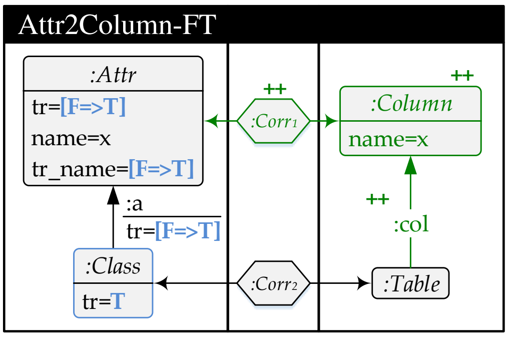
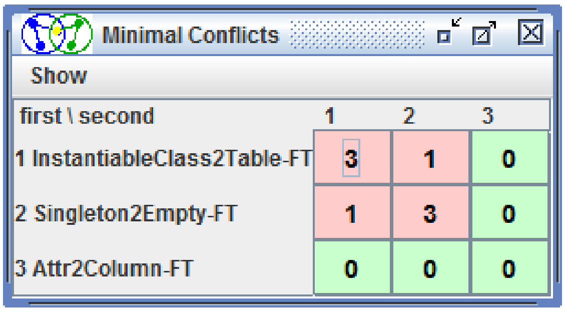

$c_9 = \neg\exists(P_1 \to C_1, \mathbf{true})$
P1
C1
P1,R
:Attr
name=x
:a
1:Class
1:Class
:a
:Attr
name=x
C1,R
$c_{10} = \neg\exists(P_2 \to C_2, \mathbf{true})$
P2
C2
P2,R
:Class
:a
1:Attr
1:Attr
:a
:Class
C2,R
$c_{11} = \neg\exists(P_3 \to C_3, \mathbf{true})$
P3
C3
P3,R
:Attr
name="instance"
:a
1:Class
1:Class
:a
:Attr
name="oid"
C3,R
$c_{12} = \neg\exists(P_4 \to C_4, \mathbf{true})$
P4
C4
P4,R
1:Class
1:Class
:a
:a
2:Attr
2:Attr
C4,R
$c_{13} = \neg\exists(P_5 \to C_5, \mathbf{true})$
P5
C5
P5,R
1:Attr
1:Attr
name=x
name=y
C5,R

Figure 4.1: Additional Domain Graph Constraints with Restrictions





first \ second	1	2	3
1 InstantiableClass2Table-FT	3	1	0
2 Singleton2Empty-FT	1	3	0
3 Attr2Column-FT	0	0	0

Figure 4.2: Foward Translation Rule Attr2Column – FT (left) & Conflict Analysis of Forward Translation Rules with AGG (right)

Corollary 4.2 (Relationship between Domain Completeness of Model Transformations & Domain Completeness under Restrictions) *Let $MT\colon \mathscr{L}(C) \Rrightarrow \mathscr{L}(D_2)$ and $MT_R\colon \mathscr{L}(Restr_t(C)) \Rrightarrow \mathscr{L}(D_2)$ be two model transformations defined by the same TGG where according to Thm. 3.9, $\mathscr{L}(Restr_t(C)) = \mathscr{L}_I(Restr_t(C_I)) \cap \mathscr{L}(Restr_t(C_G))$ for $C = C_I \cup C_G$. If MT_R is domain complete according to Def. 4.1, then MT is domain complete under restrictions according to Def. 4.3.* △

Proof. By Thm. 4.1, domain completeness of MT_R implies domain completeness w.r.t. $\mathscr{L}(Restr_t(C))$, $\mathscr{L}(TGG)^S$ and TG^S_R. By Cor. 4.1, it follows domain completeness under restrictions w.r.t. $\mathscr{L}(C)$ and $\mathscr{L}(TGG)^S$ implying further that MT is domain complete under restrictions by Thm. 4.3. □

Example 4.1 (Domain Completeness of Model Transformations under Restrictions) *The model*

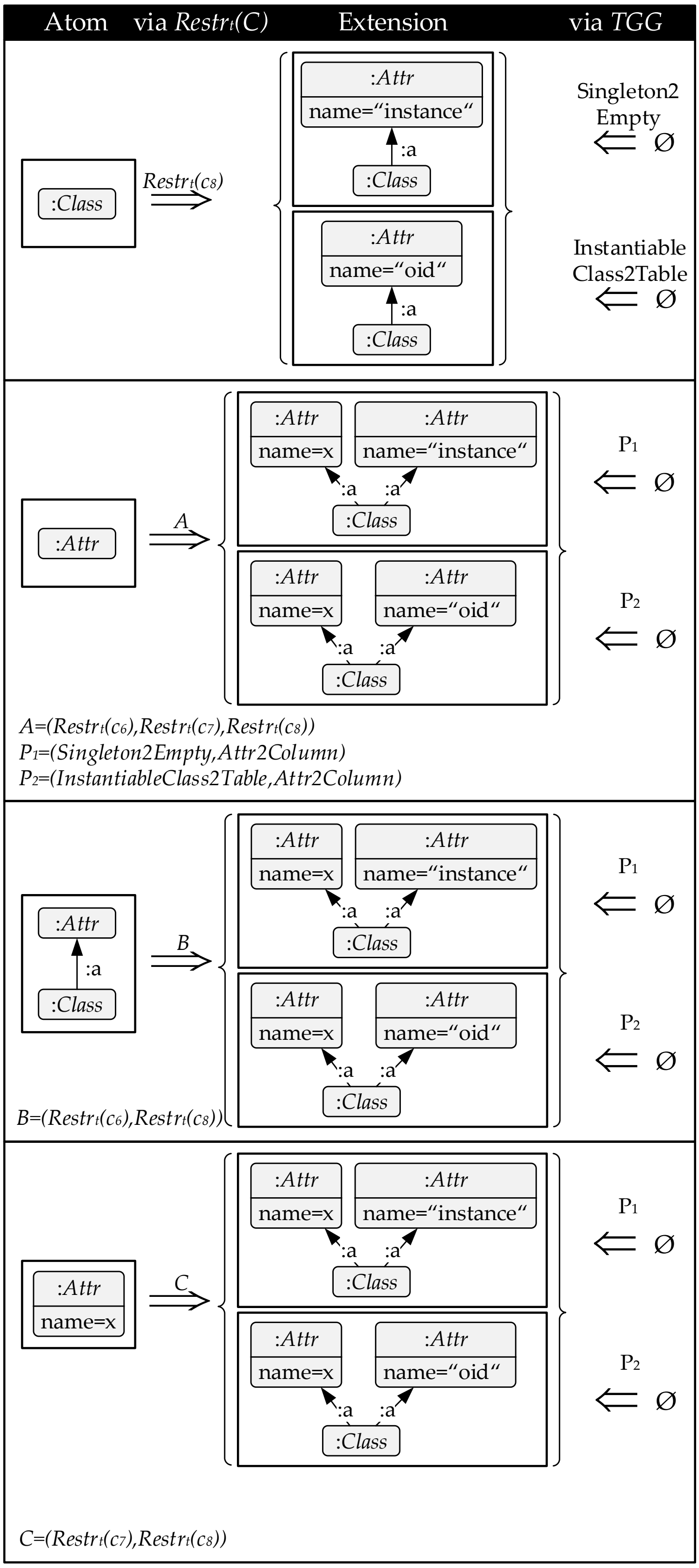
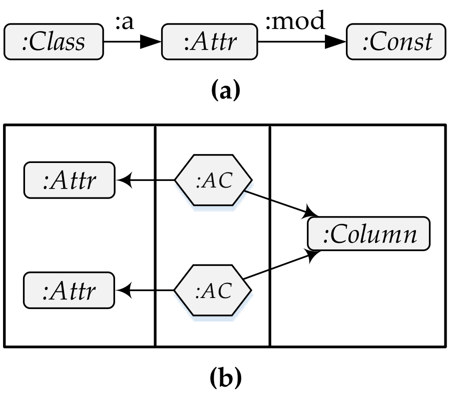
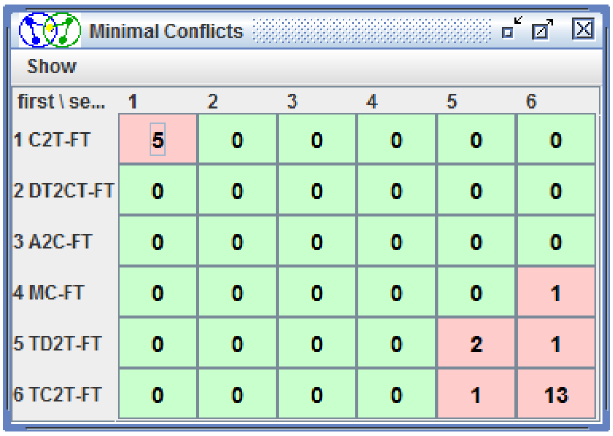

Atom
via $Restr_t(C)$
Extension
via TGG
:Class
$Restr_t(c_8)$
:Attr
name="instance"
:a
:Class
:Attr
name="oid"
Singleton2 Empty
Ø
Instantiable Class2Table
Ø
:Attr
A
name=x
P1
P2
$A=(Restr_t(c_6),Restr_t(c_7),Restr_t(c_8))$
P_1=(Singleton2Empty,Attr2Column)
P_2=(InstantiableClass2Table,Attr2Column)
B
$B=(Restr_t(c_6),Restr_t(c_8))$
C
$C=(Restr_t(c_7),Restr_t(c_8))$

Figure 4.3: Verification of $Restr_t(C)$-Extension Completeness of $\mathscr{L}(TGG)^S$

transformation $MT\colon \mathscr{L}(C) \Rrightarrow \mathscr{L}(TG_{RDBM})$ from UML class diagrams (CDs) to relational database models (RDBMs) as defined by the TGG with empty start graph and triple rules without application conditions and typed over $TG_{CD,R}$ from Sec. 3.5 and Fig. 3.16 is not domain complete w.r.t. the domain constraints $C = \{c_1, \ldots, c_{13}\}$ in Figs. 3.17, 3.18 and 4.1, since, class constructors together with their visibilities are not covered by the given TGG. We assume that all constraints $c \in C$ are designated for general satisfaction, i.e., $C = C_I \cup C_G$ with $C_I = \varnothing$ and $C_G = C$. However, the model transformation is domain complete under restrictions w.r.t. domain constraints C, the given TGG, type morphism $t\colon TG_{CD,R} \to TG_{CD} \in \mathscr{M}$ and restriction $TG_{CD,R}$. Thus, all Classes *and* Attributes *in each CD can be translated to corresponding* Tables *and* Columns *in RDBMs. In order to show this, by Cor. 4.1, we have to verify domain completeness w.r.t. the restricted domain constraints $Restr_t(C)$, the given TGG and restricted type graph $TG_{CD,R}$, i.e., by Thm. 4.1 domain completeness of model transformation $MT_R\colon \mathscr{L}(Restr_t(C)) \Rrightarrow \mathscr{L}(TG_{RDBM})$ from class diagrams that are typed over $TG_{CD,R}$ and satisfy the restricted constraints $Restr_t(C)$ to RDBMs that are typed over TG_{RDBM}. By Thm. 4.2, we have to check that the forward translation rules of TGG are $Restr_t(C)$-conflict-free and furthermore, $\mathscr{L}(TGG)^S$ is $Restr_t(C)$-extension complete. According to Sec. 3.5 and Thm. 3.9, $Restr_t(C) = \{Restr_t(c_1), Restr_t(c_6), Restr_t(c_7), Restr_t(c_8), Restr_t(c_9), Restr_t(c_{10}), Restr_t(c_{11}), Restr_t(c_{12}), Restr_t(c_{13})\}$ while constraints c_2 to c_5 are neglected for the verification, as, they are positive but unrestricted along t (c_2 to c_4) or negative but not purely restricted along t (c_5). The restricted constraints are highlighted by grey boxes in Figs. 3.17, 3.18 and 4.1 with premises $P_{\_,R}$ and conclusions $C^{[E]}_{\_,R}$. Fig. 4.2 (left) shows the forward translation rule* Attr2Column − FT *of triple rule* Attr2Column *in Fig. 3.16. Analogously, we derive forward translation rules* InstantiableClass2Table − FT *and* Singleton2Empty − FT *for triple rules* InstantiableClass2Table *and* Singleton2Empty*. Fig. 4.2 (right) depicts the result of the conflict analysis of the forward translation rules with AGG [AGG16] while omitting conflicts of the same rule and same match. Altogether, there are eight conflicts that are reflected by constraints c_9 to c_{12} in Fig. 4.1, i.e., each conclusion represents a conflict graph where the conflict occurs in the premise part of the conclusion, respectively. Note that the conflicts are actually caused by updates of translation attributes but the translation attributes are not depicted explicitly in Fig. 4.1. In more detail, between rule* InstantiableClass2Table − FT *itself and* Singleton2Empty − FT *itself there are three conflicts, respectively, as reflected by conclusions C_1, C_2 and C_4. This is due to the fact that both rules may translate 1. the same* Class *but not the same* Attribute*, or 2. the same attribute but not the same class, or 3. the same class and attribute but not the same edge* : a *the assigns the attribute to the class. Therefore, we forbid the following obvious patterns in class diagrams by constraints c_9, c_{10} and c_{12}: 1. Constraint c_9 claims that each class does not have two or more attributes of the same name. 2. Constraint c_{10} claims that each attribute is assigned to at most one class. 3. Constraint c_{12} claims that for each two classes and attributes there is at most one assignment edge* : a *between both. Between rules* InstantiableClass2Table − FT *and* Singleton2Empty − FT *and vice versa there is one conflict, respectively, as reflected by conclusion C_3. This is due to the fact that both rules may translate the same class while translating different attributes* “instance″ *and* “oid″ *of that class. Therefore, we forbid the pattern of a class that has attributes* “instance″ *and* “oid″ *at the same time in class diagrams by constraint c_{11}, i.e., class diagrams are not allowed to contain classes that are both instantiable and singleton at the same time. Thus, by adding constraints c_9 to c_{12} to domain constraints c_1 to c_8, the forward translation rules are $Restr_t(C)$-conflict-free. It remains to verify that $\mathscr{L}(TGG)^S$ is $Restr_t(C)$-extension complete. The successful verification is depicted in Fig. 4.3. Each atom over restricted type graph $TG_{CD,R}$ can be extended via restricted constraints $Restr_t(C)$ such that the extension can be created by applying the triple rules of the given TGG with starting at the empty start graph*





first \ se...	1	2	3	4	5	6
1 C2T-FT	5	0	0	0	0	0
2 DT2CT-FT	0	0	0	0	0	0
3 A2C-FT	0	0	0	0	0	0
4 MC-FT	0	0	0	0	0	1
5 TD2T-FT	0	0	0	0	2	1
6 TC2T-FT	0	0	0	0	1	13

Figure 4.4: Graph Pattern of Conflict Graphs (a) and (b) & Conflict Analysis of Forward Translation Rules with AGG (right)

$\varnothing$:

1. *The atom with a single* : Class *node can be extended via $Restr_t(c_8)$ to two graphs, i.e., one graph with additional* "instance" *attribute and one with additional* "oid" *attribute, and both graphs can be created via rule* Singleton2Empty *or* InstantiableClass2Table*, respectively.*

2. *The atom with a single* : Attribute *node can be extended via restricted constraints $Restr_t(c_6)$,$Restr_t(c_7)$ and $Restr_t(c_8)$ successively to two graphs, i.e., one graph with two additional* "instance" *and* x *attributes and one with two additional* "oid" *and* x *attributes, and both graphs can be created via rules* Singleton2Empty *and* Attr2Column *OR* InstantiableClass2Table *and* Attr2Column *successively.*

3. *The atom with an* : Attribute *node,* : Class *node and* : assignment *edge in between can be extended via $Restr_t(c_6)$ and $Restr_t(c_8)$ successively to the same two graphs as in Item 2.*

4. *The same is true for the atom with a single* : Attribute *node and node attribute* name *via $Restr_t(c_7)$ and $Restr_t(c_8)$ successively. Note that constraint c_{13} is essential for building the extensions, e.g., the second extension step in Item 4 via $Restr_t(c_8)$ may also lead to an overlapping resulting in an extension graph G_E where node* : Attr *has two* name *attributes which cannot be created via the rules of the TGG. However, due to constraint c_{13}, graph G_E does not occur in extensions, since, G_E is $Restr_t(C)$-inconsistent according to Def. 3.5.*

Therefore, the model transformation MT_R from class diagrams typed over $TG_{CD,R}$ that satisfy the restricted domain constraints $Restr_t(C)$ to RDBMs is domain complete implying further according to Cor. 4.2 that the model transformation MT from class diagrams typed over TG_{CD} that satisfy the original domain constraints C to RDBMs is domain complete under restrictions w.r.t. the given TGG, type morphism t and restricted type graph $TG_{CD,R}$. △

Example 4.2 (Domain Completeness of Model Transformations with Application Conditions) *Given the model transformation CD2RDBM from UML class diagrams to relational database models in Sec. 2.3.2 and Ex. 2.6 based on the forward translation rules TR_{FT} of the triple rules TR with application conditions in Sec. 2.2.4 and Fig. 2.8 with triple graph grammar $CD2RDBM = (\varnothing, TR)$ in Sec. 2.3.1 and Ex. 2.5 that is typed over triple type graph $TG =$*

$(TG_{CD} \leftarrow TG_C \rightarrow TG_{RDBM})$ in Sec. 2.2.1 and Fig. 2.4. Furthermore, given the source domain constraints C^S for UML class diagrams in Sec. 2.2.3 and Ex. 2.3 that are typed over type graph TG_{CD}. Fig. 4.4 (right) depicts the result of the conflict analysis of forward translation rules TR_{FT} via AGG [AGG16] where all critical pairs that are directly strict confluent and all critical pairs with conflict graphs that do not satisfy the multiplicity constraints in TG_{CD} in the source component, respectively, are already omitted. Note that critical pairs with conflict graphs whose source component violates a negative constraint in C^S is C^S-inconsistent according to Def. 4.2 and Item 1 and therefore also not significant w.r.t. $\mathscr{L}(C^S)$ by Sec. 3.2 and Rem. 3.2 and thus, can be omitted by Def. 4.2 and Item 1. In total there are the following 23 critical pairs remaining:

1. *For forward translation rule* C2T − FT, *the source component of the conflict graph O of each of the five critical pairs contains two* Classes *having the same* name *which is forbidden by constraint* 12. *Therefore, the five critical pairs can be omitted by Def. 4.2 and Item 1.*

2. *For the critical pair of rules* MC − FT *and* TC2T − FT, *the source component of the conflict graph contains graph pattern Fig. 4.4 (a) which is forbidden by constraints* 9 *and* 15. *Therefore, this critical pair can be omitted by Def. 4.2 and Item 1.*

3. *The conflict graphs of the remaining critical pairs contain triple graph pattern Fig. 4.4 (b), respectively, which cannot be created by forward translation sequences via TR_{FT}. Thus, these critical pairs can also be omitted by Def. 4.2 and Item 2.*

Therefore, the forward translation rules TR_{FT} are C^S-conflict-free. C^S-extension completeness of $\mathscr{L}(CD2RDBM')^S$ can be successfully verified analogously to the verification of C-extension completeness in Sec. 3.2 and Ex. 3.6 but this time without a projection to the source domain only. Therefore, $\mathscr{L}(C^S) \subseteq \mathscr{L}(CD2RDBM)^S$ by Thm. 4.2. Thus, model transformation CD2RDBM is domain complete by Thm. 4.1. Therefore, all (infinitely many) UML class diagrams that satisfy the source domain constraints C^S can be transformed to relational database models. $\triangle$

4.2 Domain Completeness of Model Synchronisations

A model synchronisation is domain complete if each update on a source graph $G^S \in \mathscr{L}(C^S)$ leading to a graph G'^S can be completely propagated to the target domain in the sense that all elements of G'^S are in correspondence with elements in the target domain, i.e., all elements in G'^S are translated to elements in the target domain.

Definition 4.4 (Domain Completeness of Model Synchronisations) Let C^S be the set of source domain constraints and $\mathscr{L}(C^S)$ be the source domain language of graphs. Furthermore, let TGG be a triple graph grammar that specifies the translation of graphs in $\mathscr{L}(C^S)$ into graphs of the target domain. Let $u\colon M^S \rightarrow M'^S$ with $M^S, M'^S \in \mathscr{L}(C^S)$ be a model update from a model M^S to a model M'^S both in the source domain. *The synchronisation is domain complete*, if for each such update u and triple graph $M = (M^S \leftarrow M^C \rightarrow M^T)$ the forward propagation operation $fPpg(M,u) = (M',u')$ leads to an update $u'\colon M^T \rightarrow M'^T$ in the target domain and integrated model $M' = (M'^S \leftarrow M'^C \rightarrow M'^T)$ such that there is a model transformation sequence $(M'^S, M_0 \xRightarrow{tr^*_{FT}} M_n, M'^T)$ based on the forward translation rules of TGG with $M_0 = (Att^{\mathbf{F}}(M'^S) \leftarrow \varnothing \rightarrow \varnothing)$ and $M_n = (Att^{\mathbf{T}}(M'^S) \leftarrow M'^C \rightarrow M'^T)$. $\triangle$

Based on the "classical" syntactical completeness and correctness of model transformations and synchronisations by triple graph grammars based on forward translation rules (cf. Cor. 8.5

& Thm 9.25 in [EEGH15]) and the decomposition property of TGGs, the domain completeness of model synchronisations can be reformulated as follows. While Def. 4.4 reflects the intuitive meaning behind complete synchronisations, Thm. 4.4 expresses completeness in terms of a language inclusion which can be verified by using the verification techniques for domain completeness in Sec. 4.1 and Thm. 4.2. Therefore, both formulations of completeness in Def. 4.4 and Thm. 4.4 are equivalent.

Theorem 4.4 (Domain Completeness of Model Synchronisations) *Let $Synch(TGG)$ be the derived TGG synchronisation framework with forward propagation operation fPpg such that the sets of operational translation rules derived from TGG are kernel-grounded and deterministic. Then, the synchronisation via fPpg is domain complete according to Def. 4.4 if and only if $\mathscr{L}(C^S) \subseteq \mathscr{L}(TGG)^S$.* △

Proof. **"⇒"** Let $u\colon M^S \xrightarrow{id} M^S$ be the identical update with $M^S \in \mathscr{L}(C^S)$ and $M = (M^S \leftarrow M^C \rightarrow M^T)$. Furthermore, by "classical" correctness of model synchronisations $fPpg(M,u) = (M,u)$ and there is a model transformation sequence $(M^S, M_0 \xRightarrow{tr^*_{FT}} M_n, M^T)$ based on forward translation rules of *TGG*. By "classical" correctness of model transformations, there is $(M^S \leftarrow M'^C \rightarrow M^T) \in \mathscr{L}(TGG)$ implying further that $M^S \in \mathscr{L}(TGG)^S$ (cf. Def. 8.3 in [EEGH15]). Thus, $\mathscr{L}(C^S) \subseteq \mathscr{L}(TGG)^S$.

"⇐" Let $u\colon M^S \rightarrow M'^S$ with $M^S, M'^S \in \mathscr{L}(C^S)$ implying that $M^S, M'^S \in \mathscr{L}(TGG)^S$. Furthermore, let $M = (M^S \leftarrow M^C \rightarrow M^T)$. By "classical" completeness of model synchronisations $fPpg(M,u) = (M',u')$ with $M' = (M'^S \leftarrow M'^C \rightarrow M'^T)$ and $u'\colon M^T \rightarrow M'^T$. By "classical" correctness of model synchronisations $M' \in \mathscr{L}(TGG)$, i.e., there is $\varnothing \xRightarrow{tr^*} M'$ via rules in *TGG*. By the decomposition property of TGGs, there is a corresponding match-consistent triple transformation sequence $M_{0,0} = \varnothing \xRightarrow{tr_{1,S}} M_{1,0} \Rightarrow \ldots \xRightarrow{tr_{n,S}} M_{n,0} \xRightarrow{tr_{1,F}} M_{n,1} \overset{\ldots}{\Longrightarrow} \xRightarrow{tr_{n,F}} M'$ with $M_{n,0} = (M'^S \leftarrow \varnothing \rightarrow \varnothing)$ (cf. Def. 7.21 in [EEGH15]). By definition there is there is a source consistent triple transformation $M_{n,0} \xRightarrow{tr^*_F} M'$ (cf. Def. 7.18 in [EEGH15]). Therefore by Def. 7.23 in [EEGH15], there is a model transformation sequence $(M'^S, M_{n,0} \xRightarrow{tr^*_F} M', M'^T)$. By Fact 7.36 in [EEGH15], there is a model transformation sequence $(M'^S, M'_0 \xRightarrow{tr^*_{FT}} M'_n, M'^T)$ based on forward translation rules of *TGG* with $M'_0 = (Att^{\mathbf{F}}(M'^S) \leftarrow \varnothing \rightarrow \varnothing)$ and $M'_n = (Att^{\mathbf{T}}(M'^S) \leftarrow M'^C \rightarrow M'^T)$.

□

Example 4.3 (Domain Completeness of Model Synchronisations) *For the source domain constraints C^S of UML class diagrams and triple graph grammar CD2RDBM for translating UML class diagrams into relational database models, in Sec. 4.1 and Ex. 4.2 we have already shown that $\mathscr{L}(C^S) \subseteq \mathscr{L}(CD2RDBM)^S$. Therefore, the model synchronisation for propagating updates from UML class diagrams to relational database models is domain complete by Thm. 4.4. Thus, each update on UML class diagrams that respects domain constraints C^S can be propagated to a corresponding update on interlinked relational database models.* △

Chapter 5

Further Applications

5.1 Completeness of Software Transformations

5.1.1 Introduction to Software Transformations

Software translations are the intrinsic idea of compilers where source code written in a (domain-specific) programming language is translated into formulations of a mostly more low-level intermediate or directly machine-readable language [ASU06, SW13]. Software translations are also demanded in scenarios where programs written in a diverse set of programming languages may be translated into a unified language for unified maintainability and verification [HGN$^+$14]. Apart from such text-to-text translations, a much older tradition is to translate textual representations into visual models which reflect the cognitive concepts we had in mind when preparing and writing the texts from different domains (text-to-model translations) [Str08] and vice versa (model-to-text translations) [SC12]. In the software world this trend was enforced by model-driven engineering (MDE) approaches [WHR14] where source code is represented by diagram models in a diverse set of different domain specific visual modelling languages (e.g. UML) and vice versa source code is generated from models not only in the development phase but also for maintenance after the systems have been deployed (cf. Fig. 5.1).

We do not concentrate on the reasons for translations (e.g., reducing or creating abstractions) but on the well-known formal concept of graph grammars [EEPT06] from Sec. 2.2.4 as a generalisation of word grammars for specifying model transformations [EEGH15, MG06, CH03] in general (cf. Sec. 2.3.2) and text-to-text [HNB$^+$14] as well as text-to-model (model-to-text) translations in particular. In the context of graph grammars, models are represented by graphs whereas the translation of models is specified by a graph grammar, i.e., a set of graph transformation rules together with a start graph. This allows the verification of the completeness of translations. A translation from language $\mathscr{L}_1$ to $\mathscr{L}_2$ is syntactically complete, if all elements of $\mathscr{L}_1$ can be completely translated. The completeness problem can be reformulated as a language inclusion problem $\mathscr{L}_1 \subseteq \mathscr{L}_3$ where $\mathscr{L}_3$ is the language induced by the given graph grammar which specifies the translation (cf. Sec. 5.1.5 and Thm. 5.1). In general, the language inclusion problem is undecidable for context-free grammars $\mathscr{L}_3$, most of their sub-classes and all classes above [AN00] (cf. Sec. 3.1). Therefore, we developed sufficient conditions (called domain completeness [NHBE14]) to approximately solve the inclusion problem for languages $\mathscr{L}_1$ that are given by a type graph [GLEO12] together with a set of graph constraints [HP05, EEGH15] and languages $\mathscr{L}_3$ that are given by a graph grammar (cf. Sects. 3.2 and 4.1). We concentrate on instantiating the general concept of domain completeness of model transformations from Sec. 4.1

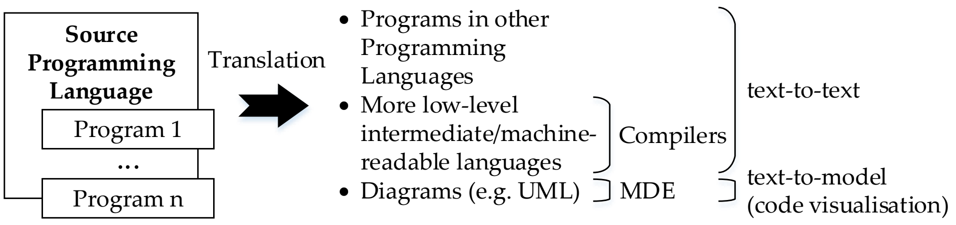

Source Programming Language
Program 1
...
Program n
Translation
Programs in other Programming Languages
More low-level intermediate/machine-readable languages
Diagrams (e.g. UML)
Compilers
MDE
text-to-text
text-to-model (code visualisation)

Figure 5.1: Types of Software Translations

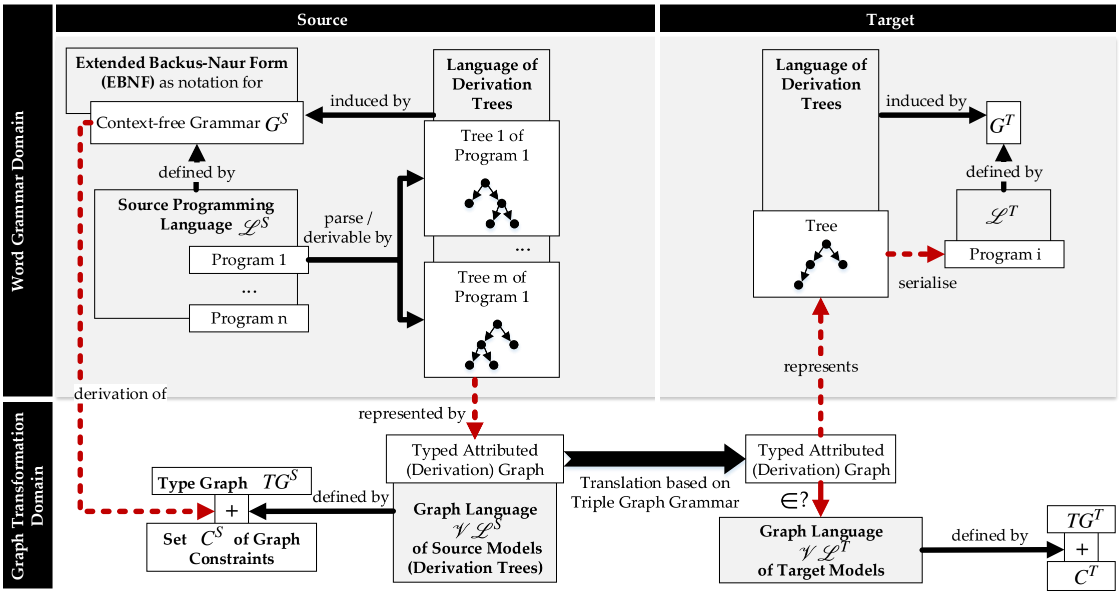

Source
Target
Word Grammar Domain
Graph Transformation Domain
Extended Backus-Naur Form (EBNF) as notation for
Context-free Grammar G^S
induced by
Language of Derivation Trees
Tree 1 of Program 1
...
Tree m of Program 1
defined by
Source Programming Language $\mathscr{L}^S$
Program 1
...
Program n
parse / derivable by
derivation of
represented by
Language of Derivation Trees
induced by
G^T
defined by
$\mathscr{L}^T$
Program i
Tree
serialise
represents
Typed Attributed (Derivation) Graph
Translation based on Triple Graph Grammar
Typed Attributed (Derivation) Graph
∈?
Type Graph TG^S
+
Set C^S of Graph Constraints
defined by
Graph Language $\mathscr{V}\mathscr{L}^S$ of Source Models (Derivation Trees)
Graph Language $\mathscr{V}\mathscr{L}^T$ of Target Models
defined by
TG^T
+
C^T

Figure 5.2: Overview of Software Translations based on Triple Graph Grammars

to the completeness problem of software translations in Sec. 5.1.3 by closing the gap between the definition of programming languages and the definition of graph languages in Sec. 5.1.4. Derivation trees of context-free grammars are represented by typed attributed graphs [GLEO12] with a tree-structure. Therefore, we present an encoding of context-free word grammars to attributed type graphs together with graph constraints such that the language of derivation trees that is induced by the word grammar is isomorphic to the graph language that is induced by the type graph together with the graph constraints (cf. Sec. 5.1.4 and claim 5.1). This allows the application of the domain completeness verification from Sec. 4.1 in the graph transformation world in order to check if all abstract syntax trees that can be formed over a programming language can actually be translated. For unambiguous programming languages L [HMU03] that have a unique derivation tree for each program, this gives a good hint towards the syntactical completeness of the translation of programs written in L.

5.1.2 Software Transformations based on Triple Graph Grammars

Fig. 5.2 illustrates an overview of software translations based on triple graph grammars. In software translations, programs written in a source programming language $\mathscr{L}^S$ are translated into programs (i.e. words) of a target (programming) language $\mathscr{L}^T$ (text-to-text translation) or into models of a target visual modelling language $\mathscr{V}\mathscr{L}^T$ (text-to-model translation) (cf. Fig. 5.1). We

```
 1 PROGRAM:  first=CLASS
   ----------------------------------------------------
 3
   CLASS   :  'class' name1=STRING (':' e=
        ↪     STR_LST)? '{' (attr1=ATTR)? '}'
        ↪     ('\n' (n1=CLASS | n2=ST))?
 5 STR_LST:  name2=STRING (',' n3=STR_LST)?
   ATTR    :  type=STRING name3=STRING ';' (n4
        ↪     =ATTR)?
 7 ----------------------------------------------------

 9 NEW     :  'new' cl2=STRING
   ACCESS  :  obj=VAR '.' attr2=STRING
11 ST      :  (a=ASG | p=PRINT | r=READ | i=IF
        ↪      | g=GOTO) ';' ('\n' n5=ST)?
   ASG     :  (a1=VAR | a2=ACCESS) '=' (a3=VAR
        ↪      | a4=ACCESS | a5=NEW)
13 PRINT   :  'print' out=ACCESS
   READ    :  'read' (in1=VAR | in2=ACCESS)
15 IF      :  'if' c=COND 'then' body=ST 'end'
   COND    :  l=VAR ('='|'!=') (r1=STRING | r2
        ↪     =ACCESS | r3=NULL)
17 GOTO    :  'goto' line=INT
   ----------------------------------------------------
19
   terminal NULL    :  'null'
21 terminal STRING  :  '"'.*'"'
   terminal VAR     :  ('a'..'z'|'A'..'Z')+
23 terminal INT     :  (0..9)+
```

```
 1 class "Person" { "String" "name"; "Person
        ↪   " "next"; }
   class "Employee" : "Person" { "String" "
        ↪   salary"; }
 3 ----------------------------------------------------
   fst = new "Employee";
 5 lst = fst;
   read proceed;
 7 ----------------------------------------------------
   if proceed = "in" then
 9   read lst."name";
     read lst."salary";
11   p = new "Employee";
     lst."next" = p;
13   lst = p;
     goto 6;
15 end;
   ----------------------------------------------------
17 if proceed = "out" then
     read name;
19   current = fst;
     if name = current."name" then
21     print current."salary";
     end;
23   current = current."next";
     if current != null then
25     goto 20;
     end;
27   goto 6;
   end;
```

Figure 5.3: Xtext EBNF Grammar of language *Conditional-IN-OUT* (left) & Program of language *Conditional-IN-OUT* (right)

assume that the source (target) word languages $\mathscr{L}^S$ ($\mathscr{L}^T$) are defined by context-free grammars G^S (G^T) where the extended backus-naur form (EBNF) is used as a short-hand notation for context-free grammars in order to define programming languages. Moreover, we assume that the target visual modelling language $\mathscr{V}\mathscr{L}^T$ is a graph language of typed attributed graphs as models and is defined by an attributed type graph TG^T together with a set of graph constraints C^T. Each context-free grammar induces a language of derivation trees that can be formed over the grammar. Each derivation tree defines how a specific program (word) can be derived from the grammar. Therefore, each derivation tree represents a specific program (word) of the language which is defined by the grammar. Thus, the translation of programs is performed by translating their derivation trees. In the following we give examples for an EBNF grammar, a derivation tree and the corresponding program which serve as running examples for the subsequent sections.

Example 5.1 (Xtext EBNF Grammar) *The EBNF grammar in Fig. 5.3 (left) is presented in Xtext notation and defines the "toy" programming language Conditional-IN-OUT. Xtext [xte16] uses a special EBNF syntax for specifying context-free grammars of textual domain specific languages. Xtext is widely used for the definition of programming languages and generates language parsers and editors automatically.*

Each line represents a grammar rule. In language Conditional-IN-OUT, programs may read input from the user (line 14), assign it to a (member) variable (ACCESS (line 9)) VAR (line 22) and use the variable in comparisons with STRINGs (line 21) and variables as conditions COND (line 16) in if-statements (line 15) in order to print some conditional output to the user (line 13). Additionally to local variables VAR, data may also be stored in a more structured object-oriented *way in member variables of classes. Each PROGRAM first starts with a class definition (line 1). Each CLASS (line 4) optionally refers (n1) to the next class or statement ST (n2). Furthermore, each CLASS has a name (name1) of type STRING and optionally may inherit (reference e) from a set of classes that are given by a comma-separated list STR_LST of STRINGs (line 5) that are given after a colon : in the class definition and represent the class names. Moreover, each CLASS may refer (attr1) to a set of attributes ATTR as member variables. Each attribute (line 6) is given by a type and a name (name3), both defined by a STRING, and optionally may refer (n4) to the next attribute separated by a semicolon. Object creation (line 9) is given by the keyword new followed by the class name (cl2) of the class from which the object is created. ACCESS (line 10) to a member variable of an object is given by the standard dot-notation with a VARiable that refers to the object (obj) on the left and the name of the member (attr2) on the right. Statements ST (line 11) of a PROGRAM are assignments (ASG), PRINT, READ, IF-statements or GOTOs. Furthermore, each statement ST optionally may refer (n5) to the next statement separated by a semicolon and newline. Additionally to the statements that we already discussed above, assignments (line 12) are given by the symbol = with a (member) variable (a2) a1 on the left and a (member) variable (a4) a3 or object creation (a5) on the right. Furthermore, GOTO statements (line 17) are given by the keyword goto together with the number INT of the line in which the execution of the program should proceed.*

In Xtext notation, terminal symbols may be grouped in sorts and defined by so called terminal rules with regular expressions for each sort (lines 20-23). Sort STRING (line 21) is defined by the regular expression '"'.'"' allowing sequences of any character that are enclosed by quotation marks. In contrast to STRINGs, variables VARs (line 22) are given by sequences of alphabetic characters that are not enclosed by quotation marks and INTs (line 23) are defined by sequences of numbers. Moreover, the sort NULL (line 20) is given by the keyword null which refers to the null object.* △

Example 5.2 (Program in language *Conditional-IN-OUT*) *In Fig. 5.3 (right), we present a program of language Conditional-IN-OUT from Ex. 5.1. Two classes Person and Employee are defined. Persons have a name and a next Person (line 1). Employees are a special type of Persons by inheritance and additionally have a salary (line 2). Lines 4-5 initialise an empty list of Employees with a new Employee object as the first (fst) and last (lst) list element. Line 6 reads the user input into variable proceed. Depending on the user input, either the program asks the user to add a new Employee to the list (lines 8-15) or the program returns the salary of an existing Employee in the list (lines 17-27) where the Employee is identified by his name. Finally, the program proceeds with line 6 or terminates if the user input differs from "in" or "out".* △

Example 5.3 (Derivation Tree) *The derivation tree of the program in Ex. 5.2 up to line 3 as induced by the grammar of Ex. 5.1 is presented in graphical notation in Fig. 5.4. Non-terminals become nodes in the tree, terminals of sorts STRING,VAR,INT and NULL become node attributes and references between grammar rules become edges. Explicit keywords of the language (e.g. class) are not included in the tree. By following the grammar, in derivation trees, the list of attributes of a class in a program are represented by edge :attr1 pointing to the first attribute and edges :n4 pointing to the next attribute. Analogously, the lists of class definitions and the list of inherited classes for each class in a program are represented by edges :first, :n1 and :e, :n3. The complete tree that covers the whole program is given analogously.* △

Given a context-free grammar G^S for source word language $\mathscr{L}^S$, then the translation of the induced derivation trees is specified by a triple graph grammar. By Fig. 5.2, each derivation

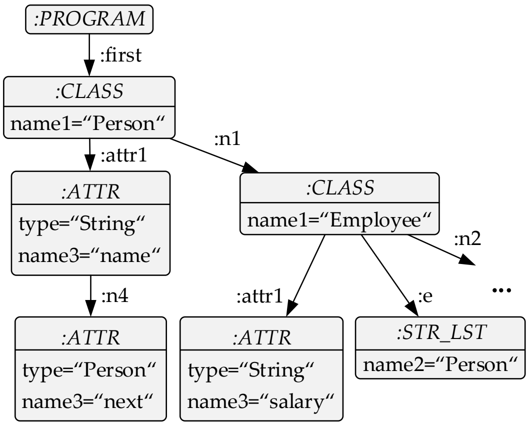


Figure 5.4: Derivation Tree

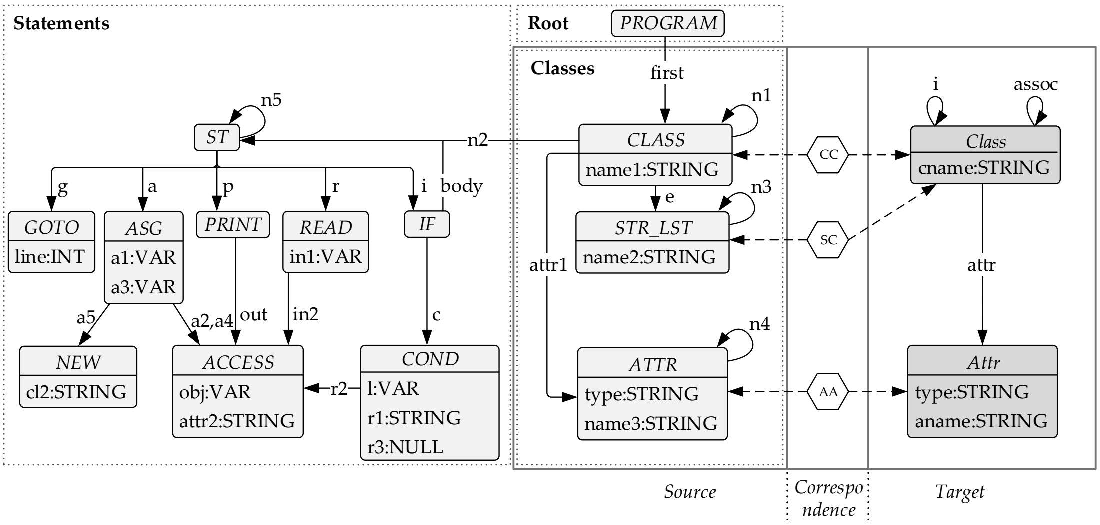


Figure 5.5: EBNF Type Graph of Grammar *Conditional-IN-OUT* (*Root*+*Statements*+*Classes*) & Triple Type Graph (*Source* ← *Correspondence* → *Target*)

tree is represented by a typed attributed graph in the source domain. Then, the translation is performed by executing a model transformation on the graph based on the given triple graph grammar. The result of the translation is a typed attributed graph in the target domain. For text-to-model translations, the resulting graph is intended to directly serve as the resulting visual model in the target visual modelling language $\mathscr{V}\mathscr{L}^T$ of the translation. For text-to-text translations, the resulting graph is intended to be a representation of a derivation tree which is induced by the target context-free grammar G^T of target word language $\mathscr{L}^T$. The overall result of the text-to-text translation is obtained by flattening (serialising) the tree to a word (program) of the target language $\mathscr{L}^T$. We demonstrate the graph-based approach for software translations by a translation of programs of language *Conditional-IN-OUT* from Ex. 5.1 into UML class diagrams in Ex. 5.5. At first, we present the attributed type graphs of the source (*Conditional-IN-OUT*) and target (class diagrams) domains of the translation in Ex. 5.4.

Example 5.4 (Attributed Type Graphs) *Fig. 5.5 illustrates both the attributed type graph of source language Conditional-IN-OUT and the type graph of the target visual modelling language of UML class diagrams. The type graph of Conditional-IN-OUT is given by boxes*

Root,Statements and Classes. The graph represents the structure of the Conditional-IN-OUT grammar in Fig. 5.3 (left) where each grammar rule becomes a node in the graph and references between rules become edges or node attributes. The type graph of class diagrams is given by box Target. Each diagram may have several Class *nodes with a name (node attribute* cname*) of type* STRING *for each class. Furthermore, each class may have several attributes (*Attr *nodes) each assigned by an* attr *edge. Each attribute has a* type *and a name (node attribute* aname*) both of type* STRING*. Moreover, there may be explicit inheritance relationships (*i *edges) and associations (*assoc *edges) between classes.* △

Example 5.5 (Translation of *Conditional-IN-OUT*) *Each program of language Conditional-IN-OUT from Ex. 5.1 is translated into an UML class diagram by transforming each class definition in the program into a* Class *node with optional attributes, association and inheritance relationships in the class diagram. The translation of programs is performed by translating their derivation trees and specified by the triple graph grammar $TGG = (\varnothing, P)$ with empty start graph $\varnothing$ and triple graph productions P as given in Fig. 5.6. The triple graph production rules are typed over the triple type graph (Source $\leftarrow$ Correspondence $\rightarrow$ Target) in Fig. 5.5. The triple type graph is given by boxes Classes* for *Source* and *Target* as described in Ex. 5.4 together with a *Correspondence* part which maps *1.* CLASS*es in programs to* Class *nodes in class diagrams (via correspondence nodes* CC*), 2. the names of inherited classes (*STR_LST *nodes) to* Class *nodes (via* SC *correspondence nodes), and 3. class attributes (*ATTR *nodes) to class attributes (*Attr *nodes) (via* AA *correspondence nodes). The production rules are presented in short-hand notation. The elements of each triple rule are partitioned into the following three parts: 1. source- (Conditional-IN-OUT), 2. target-domain (UML class diagram) of the translation, and 3. correspondence part which maps elements from source to target and vice versa. For each rule $LHS \rightarrow RHS$ with left-hand side LHS and right-hand side RHS, the LHS consists of all elements that are not marked with ++ whereas the RHS additionally consists of all elements that are marked with ++ and therefore, that are created when applying the rule to a LHS context. Moreover, each rule is equipped with a negative application condition (NAC) which restricts the rule application by describing a graph structure that is forbidden to exist for applying the rule. For example, rule 1:C2C in Fig. 5.6 simultaneously creates a CLASS and a Class node with name n in the source and target domain with correspondence node :CC between both but only if the class diagram not already contains a Class node of the same name. While the triple rules specify the translation, the translation itself is performed by applying operational forward translation rules that are derived from the triple rules. Thus, from each rule $p \in P$, an operational forward translation rule p_{FT} is automatically derived. For technical details we refer to Def. 7.29 in [EEGH15] while the idea is rather straightforward. Given a triple rule p, then the corresponding forward translation rule p_{FT} of p translates those elements E that are created by p in the source domain into those elements E' that are created by p in the target domain together with the created correspondence part by adding elements E' to elements E. Moreover, the translation rule marks the translated elements as being translated in order to prevent a second translation of those elements. In the following we give short descriptions for the forward translations rules of the triple rules in Fig. 5.6. The major challenges of the translation include: 1. the mapping of the list of class attributes for each class in a program to a star of attributes around the corresponding class node in the class diagram (rules 2 & 3), 2. the mapping of the list of names of inherited classes for each class in a program to explicit inheritance relationships between actual class nodes in the class diagram (rules 4 & 5), and 3. the mapping of implicit associations between classes in a program as given by the types of class attributes to explicit association relationships between actual class nodes in the class diagram (rules 6 to 10).*

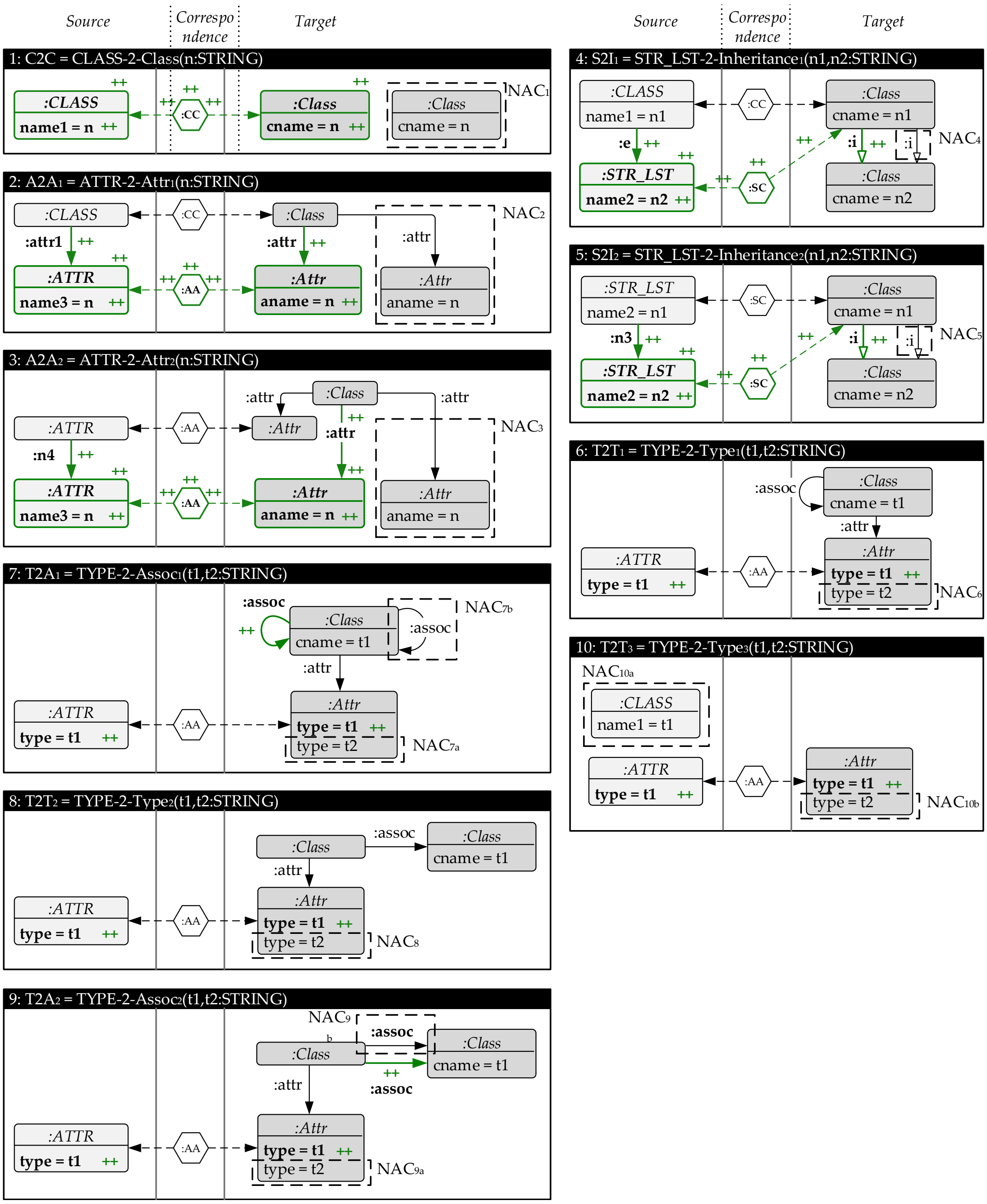

Source
Correspondence
Target
1: C2C = CLASS-2-Class(n:STRING)
:CLASS
name1 = n
:CC
:Class
cname = n
NAC1
2: A2A1 = ATTR-2-Attr1(n:STRING)
:attr1
:ATTR
name3 = n
:AA
:attr
:Attr
aname = n
NAC2
3: A2A2 = ATTR-2-Attr2(n:STRING)
:n4
NAC3
4: S2I1 = STR_LST-2-Inheritance1(n1,n2:STRING)
name1 = n1
:e
:STR_LST
name2 = n2
:SC
cname = n1
:i
cname = n2
NAC4
5: S2I2 = STR_LST-2-Inheritance2(n1,n2:STRING)
name2 = n1
:n3
NAC5
6: T2T1 = TYPE-2-Type1(t1,t2:STRING)
:assoc
cname = t1
type = t1
type = t2
NAC6
7: T2A1 = TYPE-2-Assoc1(t1,t2:STRING)
NAC7b
NAC7a
8: T2T2 = TYPE-2-Type2(t1,t2:STRING)
NAC8
9: T2A2 = TYPE-2-Assoc2(t1,t2:STRING)
NAC9b
NAC9a
10: T2T3 = TYPE-2-Type3(t1,t2:STRING)
NAC10a
name1 = t1
NAC10b
++

Figure 5.6: Triple Graph Rules for Translation of Derivation Trees of Grammar *Conditional-IN-OUT*

1. *Rule 1:C2C$_{FT}$ translates a CLASS of name n in a program into a corresponding Class node with name n in the class diagram but only if there does not already exist a Class node with the same name (cf. NAC$_1$).*

2. *Rule 2:A2A$_{1,FT}$ assumes that a CLASS in a program is already translated into a corresponding Class node in the class diagram. The rule translates the first class attribute (:ATTR) with name n of :CLASS into a corresponding attribute (:Attr) with name n of node :Class in the class diagram but only if node :Class does not already contain an attribute with the same*

name (cf. NAC_2).

3. *Rule $3{:}A2A_{2,FT}$ translates the next attribute of the class into a corresponding attribute in the class diagram but analogously to rule $2{:}A2A_{1,FT}$, only if the correponding Class node in the diagram does not already contain an attribute with the same name (cf. NAC_3).*

4. *Rule $4{:}S2I_{1,FT}$ translates the name n2 of the first inherited class for class n1 in a program into an inheritance edge between the corresponding Class nodes in the class diagram but only if there does not already exist an inheritance edge between both nodes (cf. NAC_4).*

5. *Analogously to rule $3{:}A2A_{2,FT}$ for rule $2{:}A2A_{1,FT}$, rule $5{:}S2I_{2,FT}$ translates the name n2 of the next inherited class for class n1 into an inheritance edge between the corresponding Class nodes in the class diagram but only if there does not already exist an inheritance edge between both nodes (cf. NAC_5).*

6. *Rule $10{:}T2T_{3,FT}$ translates the type t1 of a class attribute in a program into the same type of the corresponding attribute in the class diagram but only if type t1 is not the name of a CLASS in the program and the attribute in the diagram does not already have a type t2 (cf. $NAC_{10a} \wedge NAC_{10b}$).*

7. *In contrast to rule $10{:}T2T_{3,FT}$, if type t1 is the name of a CLASS in the program, then:*

   (a) *If t1 is the name of the class of the class attribute and the corresponding Class node in the class diagram does already have a reflexive association (edge :assoc), then rule 6:T2T$_1$ simply translates t1 into the same type of the corresponding attribute in the class diagram but only if the attribute does not already have a type t2 (cf. NAC_6).*

   (b) *If t1 is the name of the class of the class attribute, then rule 7:T2A$_1$ translates t1 into the same type of the corresponding attribute together with a reflexive association for the corresponding Class node in the class diagram but only if the attribute does not already have a type t2 and the Class node does not already have a reflexive association (cf. $NAC_{7a} \wedge NAC_{7b}$).*

   (c) *If t1 is not the name of the class of the class attribute but the name of another class in the program and there already exists an association between the corresponding Class nodes in the class diagram, then rule 8:T2T$_2$ simply translates t1 into the same type of the corresponding attribute in the class diagram but only if the attribute does not already have a type t2 (cf. NAC_8).*

   (d) *If t1 is not the name of the class of the class attribute but the name of another class in the program, then rule 9:T2A$_2$ translates t1 into the same type of the corresponding attribute together with an association between the corresponding Class nodes in the class diagram but only if the attribute does not already have a type t2 and there does not already exist an association between the two Class nodes (cf. $NAC_{9a} \wedge NAC_{9b}$).*

The result of translating the derivation tree in Ex. 5.3 by the forward translation rules from above is given by the typed attributed graph in Fig. 5.7. The graph serves as a class diagram model in the target domain of the translation. The list of attributes name and next of class Person becomes a star of attributes around the corresponding class node in the class diagram. Furthermore, the implicit inheritance and association relationships between classes in the program become explicit edges in the class diagram. △

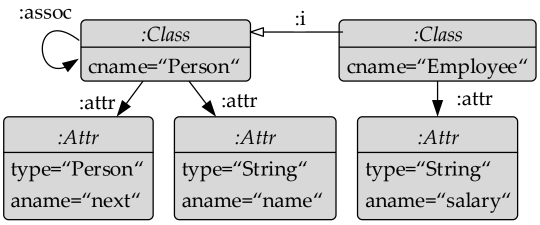


Figure 5.7: Result Graph

Remark 5.1 (Application Conditions in Translation) *The rules in Ex. 5.5 make extensive use of negative application conditions (NACs) in order to control the translation. While the negative application conditions NAC_{7b},NAC_{9b} and NAC_{10a} ensure that at most one rule of rules 6 to 10 as applicable to a given context at the same time, the other application conditions rather define restrictions of the target language of the translation, i.e., restrictions that need to be satisfied by each class diagram which results from a translation of a Conditional-IN-OUT program. 1. NAC_1 ensures that each resulting class diagram does not contain two classes with the same name, 2. NAC_2 and NAC_3 ensure that each resulting class diagram does not contain a class with two or more class attributes having the same name, 3. NAC_4 and NAC_5 ensure that each resulting class diagram does contain at most one inheritance edge between each two classes, 4. NAC_6,NAC_{7a},NAC_8,NAC_{9a} and NAC_{10b} ensure that each resulting class diagram does contain at most one type for each class attribute, and 5. NAC_{7b} and NAC_{9b} ensure that each resulting class diagram does contain at most one association between each two classes.* △

5.1.3 Completeness Problem of Software Transformations

As illustrated in Fig. 5.2, our approach of translating programs of a programming language $\mathscr{L}^S$ is to translate their derivation trees as induced by the word grammar G^S of $\mathscr{L}^S$. Each derivation tree is represented by a typed attributed graph in the source graph language $\mathscr{V}\mathscr{L}^S$ of derivation trees and translated by performing a model transformation based on a given triple graph grammar. The result is a typed attributed graph in the target modelling (graph) language $\mathscr{V}\mathscr{L}^T$. Furthermore, we assume that the source (target) graph language is defined by a type graph TG^S (TG^T) together with a set C^S (C^T) of graph contraints.

Definition 5.1 (Graph Language over Type Graph & Constraints) Given a type graph TG and a set C of graph constraints that are typed over TG. Then, the *graph language over TG and C* is given by $\mathscr{L}(TG,C) = \{G \mid \text{graph } G \text{ is typed over } TG, G \models C\}$ all graphs G that are typed over TG and that satisfy the constraints C. △

The translation is syntactically complete, if all derivation trees of G^S can be completely translated. Intuitively, a derivation tree is completely translated, if all nodes and edges of the tree are translated exactly once. Since, the translation of derivation trees is based on the translation of their graph representations, it is required that the language $Der(G^S)$ of derivation trees of G^S is isomorphic to the graph language $\mathscr{L}(TG^S,C^S)$ of their graph representations (cf. Sec. 5.1.4 and claim 5.1). Thus, each tree in $Der(G^S)$ is represented by exactly one graph in $\mathscr{L}(TG^S,C^S)$ and each graph in $\mathscr{L}(TG^S,C^S)$ represents exactly one derivation tree in $Der(G^S)$.

Definition 5.2 (Isomorphism of Derivation Tree Languages) Given a context-free word grammar G with induced language $Der(G)$ of derivation trees. Let $\mathscr{L}(TG,C)$ be a graph language of derivation trees over type graph TG and graph constraints C. *Languages $Der(G)$ and $\mathscr{L}(TG,C)$ are isomorphic*, if there exists a bijection $m\colon Der(G) \to \mathscr{L}(TG,C)$ between both. △

By Def. 5.3, a software translation is syntactically complete if each source derivation graph $G^S \in \mathscr{L}(TG^S, C^S)$ can be completely translated via a model transformation sequence in the sense that all elements of the graph are translated exactly once by changing their translation attributes from **F** to **T**.

Definition 5.3 (Syntactical Completeness of Software Translations) Given a context-free word grammar G^S for source language $\mathscr{L}^S$ with induced language $Der(G^S)$ of derivation trees. Let $TG = (TG^S \leftarrow TG^C \rightarrow TG^T)$ be a triple type graph with $\mathscr{L}(TG^S, C^S)$ being the graph language of derivation trees isomorphic to $Der(G^S)$. Furthermore, let $TGG = (\varnothing, TR)$ be a triple graph grammar typed over TG with derived forward translation rules TR_{FT} that specifies the model transformation of graphs in $\mathscr{L}(TG^S, C^S)$ into graphs of target graph language $\mathscr{L}(TG^T, C^T)$ based on forward translation rules TR_{FT}. *The translation is syntactically complete*, if for each derivation graph $G^S \in \mathscr{L}(TG^S, C^S)$ there is a model transformation sequence $(G^S, G_0 \xRightarrow{tr^*_{FT}} G_n, G^T)$ based on forward translation rules TR_{FT} with $G_0 = (Att^{\mathbf{F}}(G^S) \leftarrow \varnothing \rightarrow \varnothing)$ and $G_n = (Att^{\mathbf{T}}(G^S) \leftarrow G^C \rightarrow G^T)$. △

From the setting for translations in Fig. 5.2 and from Def. 5.3 and Thm. 5.1, the following problems arise concerning the completeness and correctness of software translations.

Problem 5.1 (Syntactical Completeness & Correctness of Software Translations) *1. Syntactical Completeness & Correctness: Given a context-free word grammar G with induced language $Der(G)$ of derivation trees. How to derive a graph language $\mathscr{L}(TG, C)$ of derivation trees from G that is isomorphic to $Der(G)$?*

2. *Syntactical Completeness: How to verify the language inclusion $\mathscr{L}(TG^S, C^S) \subseteq \mathscr{L}(TGG)^S$ (domain completeness)?*

3. *Syntactical Correctness: How to ensure that the resulting graphs of a translation are actually graphs of the target graph language $\mathscr{L}(TG^T, C^T)$, i.e., $\mathscr{L}(TGG)^T \subseteq \mathscr{L}(TG^T, C^T)$ (domain correctness)?*

4. *Syntactical Completeness & Correctness: Given verification techniques for the language inclusion problems in Items 2 and 3. Are the techniques applicable to the derived type graph TG and the derived set of graph contraints C of Item 1?* △

Beside the completeness problem, Fig. 5.2 also brings forth the correctness problem of software translations in problem 5.1 and Item 3. Therefore, are the resulting graphs 1. models of the target language in text-to-model translations, or 2. actually representations of derivation trees of the target programming language in text-to-text translations such that they can be serialised to target programs? While the correctness problem remains subject of future work under the notion of domain correctness, we furthermore concentrate on the completeness problem based on the notion of domain completeness. While the syntactical completeness of software translations in the graph world itself, i.e., the language inclusion in Thm. 5.1, can be verified by means of the existing verification techniques for domain completeness (cf. problem 5.1 and Item 2), it remains to close the gap between the word grammar and graph world (cf. problem 5.1 and Item 1) such that the verification techniques for domain completeness are applicable (cf. problem 5.1 and Item 4). The gap is closed by presenting a mapping from context-free (EBNF) grammars G to attributed EBNF type graphs TG together with EBNF graph constraints C in Sec. 5.1.4 such that the induced language $Der(G)$ of derivation trees is isomorphic to the graph language $\mathscr{L}(TG, C)$ of derivation trees (cf. Sec. 5.1.4 and claim 5.1). Furthermore, we use several extensions of the verification techniques for domain completeness in view of the derived EBNF type graphs *TG* and EBNF graph constraints C for verifying the completeness of the translation in Sec. 5.1.2 and Ex. 5.5. The extension includes 1. recursive graph constraints from Sec. 3.4, and 2. domain restrictions from Sec. 3.5. While recursive graph constraints are used for the definition of EBNF graph constraints C in Sec. 5.1.4 and Def. 5.9, domain restrictions are necessary for the translation in Sec. 5.1.2 and Ex. 5.5 which only covers those parts of a derivation tree which are related to class definitions in a program. Thus, a restriction of the graph language of derivation trees to a sub-language covering the class definitions only is required for verifying the completeness of the translation, called domain completeness under restrictions (cf. Sec. 3.5 and Def. 3.33).

5.1.4 From EBNF Grammars to Attributed Type Graphs & Graph Constraints

In [HGN$^+$14], Xtext is used for the definition of a source and target satellite procedure language for software translations between both. In the following we take (abstract) syntax trees of programs written in a programming language L and derivation trees of a context-free grammar defining this language L as synonyms. We formalise the notion of Xtext EBNF grammars in Def. 5.4, called EBNF grammars with labels. Labelled elements of the grammar are included in derivation trees whereas unlabelled elements are not. In particular, a separation of the terminals in the set T and a family of sets $(C_s)_{s\in S}$ allows a distinction of which terminals are included in derivation trees and which are not. As software translations aim at the translation of the derivation tree of a program's syntax, this distinction allows to specify which terminals are relevant for the translation and which are not. The terminals T are not included in derivation trees whereas the elements of the family of sets $(C_s)_{s\in S}$ may be included where each sort $s \in S$ represents a terminal rule in the Xtext grammar and the corresponding carrier set C_s is the set of all words that are derivable from the regular expression of the terminal rule.

Definition 5.4 (EBNF Grammar with Labels) An *EBNF grammar* $G = (N, n_S, T, L, S, (C_s)_{s\in S}, P)$ *with labels* is given by

1. A set N of non-terminals,
2. A non-terminal $n_S \in N$ as start symbol,
3. a set T of terminals,
4. a set L of labels,
5. a set S of sorts disjoint from the set N,
6. a family of carrier sets $(C_s)_{s\in S}$ with a carrier set C_s for each sort $s \in S$ (C_s defines the terminals for s),
7. a set P of production rules $p\colon n \in N \to R^*$ with a non-terminal n as the left-hand-side (LHS) of the rule and an (empty) sequence R^* as the right-hand-side (RHS) where $R = (L \times N) \cup T \cup (L \times S)$ such that:
   (a) The labelling is unique. Thus, let $|p|$ be the amount of label occurrences in production p and $L(p)$ be the set of labels that occur in p, then for P it holds that $+_{p\in P}(|p|) = |\cup_{p\in P}(L(p))|$.
   (b) All labels are used in productions, i.e., $|L| = |\cup_{p\in P}(L(p))|$.

The terminals T are called unlabelled whereas $L \times N$ are labelled references to non-terminals N and $L \times S$ are labelled references to sorts S where each sort $s \in S$ represents a set of terminals C_s. For productions with an empty sequence as RHS we write $n \rightarrow \varepsilon$, and

8. the start symbol is not target of a reference. Therefore, for all productions $p \in P$ with $p\colon n \rightarrow R^*$, it holds that $(l, n_s) \in (L \times N)$ does not occur in the RHS R^* of p. △

Def. 5.4, Item 5, and Items 7a and 7b enable the definition of the source and target of each labelled reference in an EBNF grammar based on its label $l \in L$ by well-defined functions $s\colon L \rightarrow N$ for the source and $t\colon L \rightarrow N \cup S$ for the target.

Definition 5.5 (Source & Target of Labels) Let L be the labels and P be the productions of an EBNF grammar with labels. *The source* $s\colon L \rightarrow N$ *of a label* $l \in L$ is given by $s(l) := n$ where $(l, n') \in (L \times N)$ occurs in the RHS R^* of a production $p \in P$ with $p\colon n \rightarrow R^*$. *The target* $t\colon L \rightarrow N \cup S$ *of a label* $l \in L$ is given by

$$t(l) := \begin{cases} n' & \text{if } (l, n') \in (L \times N) \text{ occurs in } R^* \\ & \text{ of a production } p \in P \text{ with } p\colon n \rightarrow R^* \\ s & \text{if } (l, s) \in (L \times S) \text{ occurs in } R^* \\ & \text{ of a production } p \in P \text{ with } p\colon n \rightarrow R^* \end{cases}$$

△

Remark 5.2 (Notational Abbreviations) *In addition to the formal EBNF notation in Def. 5.4, the notation for alternatives in* $N \rightarrow R_1(R_2|R_3)R_4$ *inductively abbreviates the two production rules* $N \rightarrow R_1R_2R_4$ *and* $N \rightarrow R_1R_3R_4$*. Furthermore, the quantification via the optional operator ? in* $N \rightarrow R_1(R_2)?R_3$ *inductively abbreviates the two productions* $N \rightarrow R_1R_3$ *and* $N \rightarrow R_1R_2R_3$. △

Remark 5.3 (Context-Free Grammars) *Note that Def. 5.4 coincides with the classical definition of context-free grammars in two forms.*

1. *Def. 5.4 coincides with the classical notion for* $T = \varnothing$ *and when defining for each terminal* t *a sort* $s_t \in S$ *with carrier set* $C_{s_t} = \{t\}$ *except that an additional unique labelling is required for each occurrence of a non-terminal or sort in the RHS of a production.*

2. *Def. 5.4 coincides with the classical notion for* $S = \varnothing$. △

For each classical context-free grammar that does not satisfy restriction Def. 5.4 and Item 8, the restriction can be bypassed by adding a production to the grammar with a new start symbol as LHS and the old start symbol as RHS without changing the language that is induced by the grammar.

Example 5.6 (EBNF Grammar with Labels) *Each line in lines 1-17 of the EBNF grammar in Sec. 5.1.2 and Ex. 5.1 represents a grammar rule of the form LHS:RHS where the LHS of the rule consists of exactly one non-terminal and the RHS is a sequence of unlabelled terminals and labelled references to non-terminals or sorts. For example, in line 4,* class,':',{ *and* } *are unlabelled terminals whereas* attr1=ATTR *is a reference to non-terminal* ATTR *with label* attr1 *and* name1=STRING *is a reference to sort* STRING *with label* name1*. The sorts are given by Xtext terminal rules in lines 20-23.*

Based on Def. 5.4, the formal definition of the grammar is given by:

1. *The set N of non-terminals that are given by the LHSs of the grammar rules with $N = \{PROGRAM, CL\_LST, CLASS, STR\_LST, ATTR, NEW, ACCESS, ST\_LST, ST, ASG, PRINT, READ, IF, COND, GOTO\}$,*

2. *the first rule $PROGRAM$ as start symbol,*

3. *the set T of terminals that are given by the words enclosed by quotes in the RHSs of the rules with $T = \{\backslash\mathrm{n}, class, :, \{, \}, ,, ;, new, ., =, print, read, if, then, end, != , goto\}$,*

4. *the set L of labels that are given by the names of the references in the RHSs of the rules with $L = \{first, cl1, n1, n2, name1, e, attr1, name2, n3, type, name3, n4, cl2, obj, attr2, st, n5, a, p, r, i, g, a1, a2, a3, a4, a5, out, in1, in2, c, body, l, r1, r2, r3, line\}$,*

5. *the sorts that are given by the terminal rules with $S = \{NULL, STRING, VAR, INT\}$,*

6. *the carrier sets that are given by the sets of all words that are derivable from the regular expressions of the corresponding terminal rules with $C_{STRING} = \{w \mid w$ is derivable from $''''.*''''\}$, $C_{VAR} = \{w \mid w$ is derivable from $('a'..'z'|'A'..'Z')+\}$, $C_{INT} = \{w \mid w$ is derivable from $(0..9)+\}$ and $C_{NULL} = \{null\}$, and*

7. *the following productions by considering the notational abbreviations in Rem. 5.2:*

   a) $PROGRAM \to (first, CL\_LST)$,

   b) $CL\_LST \to (cl1, CLASS)$,

   c) $CL\_LST \to (cl1, CLASS)\ \backslash\mathrm{n}\ (n1, CL\_LST)$,

   d) $CL\_LST \to (cl1, CLASS)\ \backslash\mathrm{n}\ (n2, ST\_LST)$,

   e) $CLASS \to class\ (name1, STRING)\ \{\ \}$,

   f) $CLASS \to class\ (name1, STRING)\ \{\ (attr1, ATTR)\ \}$,

   g) $CLASS \to class\ (name1, STRING)\ :\ (e, STR\_LST)\ \{\ \}$,

   h) $CLASS \to class\ (name1, STRING)\ :\ (e, STR\_LST)\ \{\ (attr1, ATTR)\ \}$,

   i) $STR\_LST \to (name2, STRING)$,

   j) $STR\_LST \to (name2, STRING)\ ,\ (n3, STR\_LST)$,

   k) $ATTR \to (type, STRING)\ (name3, STRING)\ ;$,

   l) $ATTR \to (type, STRING)\ (name3, STRING)\ ;\ (n4, ATTR)$, and

   m) *the productions for the rules in lines 8 to 17 are defined analogously.*

Note that the set S of sorts is disjoint from the set N of non-terminals, the labelling is unique and all labels are used in productions (cf. Def. 5.4 and Items 7a and 7b). This allows the definition of the source and target of label name1 with $s(name1) = CLASS$ and $t(name1) = STRING$ (cf. Def. 5.5). The sources and targets of the other labels are given analogously. Furthermore, the start symbol is not target of a reference (cf. Def. 5.4 and Item 8). △

Software translations are performed by translating derivation trees of programs that are induced by the underlying grammar of the programming language to models of the target language. Def. 5.6 defines the language of derivation trees that are induced by an EBNF grammar. The root node of each derivation tree is the start symbol of the grammar, the internal nodes are non-terminals whereas the leafs are non-terminals or terminals $(C_s)_{s \in S}$ of sorts S. Note that by

Def. 5.6 and Item 2, terminals T of the grammar are not contained in derivation trees making them to a special type of abstract syntax trees of programs where not all details of a grammar are reflected in the trees (cf. concrete syntax trees).

Definition 5.6 (Language of Derivation Trees) Let $G = (N, n_s, T, L, S, (C_s)_{s\in S}, P)$ be an EBNF grammar with labels. Then, *the language $Der(G) := trees(n_S)$ of derivation trees of G* is inductively defined by all trees $trees(n_S)$ with root node n_S and:

1. $trees(\varepsilon) = \{\varepsilon\}$,
2. $trees(t \in T) = \{\varepsilon\}$,
3. $trees((l,s) \in (L \times S)) = \cup_{t\in C_s}\{(l,t)\}$,
4. $trees((l,n) \in (L \times N)) = \cup_{m\in trees(n)}\{(l,m)\}$, and
5. $trees(n \in N) = \cup_{p\in P'} trees(r_n) \circ \ldots \circ trees(r_1) \circ \{n\}$, for $p\colon n \to r_1 \ldots r_n$ and $P \supseteq P' = \{p\colon n \to R^* \mid p \in P \text{ for some arbitrary } R^*\}$. △

Example 5.7 (Derivation Tree) *Below, the derivation tree of the program in Sec. 5.1.2 and Ex. 5.2 up to line 3 is presented in formal notation by following Def. 5.6. The corresponding visual graph-like notation is presented in Sec. 5.1.2 and Ex. 5.3.*

```
PROGRAM (first, CL_LST (cl1, CLASS (name1, "Person") (attr1, ATTR (type,
    ↪   "String") (name3, "name") (n4, ATTR (type, "Person") (name3, "
    ↪   next")))) (n1, CL_LST (cl1, CLASS (name1, "Employee") (e, STR_LST
    ↪    (name2, "Person")) (attr1, ATTR (type, "String) (name3, "salary
    ↪   "))) (n2, ...)))
```

The complete tree that covers the whole program is given analogously. Note that the keywords of the language (terminals T of the grammar) are not contained in the tree. △

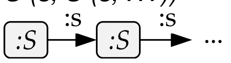
Remark 5.4 (Visual Notation of Graphs) *Note that the graphs in Ex. 5.7 are presented in visual notation. The type of each node and edge is indicated by a preceeding colon. Thus, the notation :n means that the node or edge is of type n. Technically, the typing of a node (edge) :n is given by a mapping from that node (edge) to node (edge) n in the type graph along the given type morphism of the typed graph containing this node (edge) (cf. Def. 5.7). Furthermore, for node attributes we do not use the explicit E-graph notation but visualise them directly with attribute name and value inside the corresponding nodes.* △

Remark 5.5 (Language of Derivation Trees) *In Def. 5.6, operation $\circ$ is the concatenation of words with ε being the empty word, i.e., $w \circ \varepsilon = \varepsilon \circ w = w$ for all words w. The concatenation of sets of words $W' \circ W = \{w' \circ w \mid w \in W, w' \in W'\}$ is given by all possible concatenations of words. Note that Def. 5.6 does not coincide with the classical notion of derivation trees (concrete syntax trees) of context-free grammars, not even if Rem. 5.3 and Item 2 is assumed, since, terminals T are not represented in the trees.* △

Remark 5.6 (Decidability of Derivation Tree Translations) *Note that the context-free language $\{a^n b^n \mid n \in \mathscr{N}\}$ of words with an equal amount of a's and b's that is given by a context-free grammar G with one production p (given below) may induce a regular language $\mathscr{L}(G)$ of derivation trees by following Def. 5.6 (cf. regular structure of derivation trees below). However, as the graph grammar GG for describing the translation of derivation trees may be context-free, the inclusion problem $\mathscr{L}(G) \subseteq \mathscr{L}(GG)$ remains undecidable, in general.*

$p\colon S \to a\ s = S\ b$

Derivation trees: S (s, S (s, . . .))



△

In order to close the gap between word grammars and the theory of graph transformations, we introduce a mapping from derivation trees as defined in Def. 5.6 to typed attributed graphs. The graph language of derivation trees is given by an attributed type graph together with a set of graph constraints. Thus, the mapping is based on the coding of EBNF grammars with labels to attributed type graphs, called EBNF type graphs, together with graph contraints, called EBNF graph constraints, such that the language of derivation trees that is induced by the EBNF grammar is equivalent to the language of graphs that are typed over the type graph and that fulfill the constraints. This enables the application of model transformations based on triple graph grammars to translate the derivation trees. Furthermore, this enables the application of the developed theory of domain completeness in order to verify the completeness of the translation.

Def. 5.7 defines the coding of EBNF grammars from Def. 5.4 to EBNF type graphs (cf. Def. 8.7 in [EEPT06]) and Def. 5.9 to EBNF graph constraints. The coding to EBNF type graphs is performed as follows.

1. Each non-terminal n is represented by a node n.
2. Each reference (l,n') with label l to a non-terminal n' in the RHS of a production $n \to R^*$ is represented by an edge l from node n to n'.
3. Each reference (l,s) with label l to a sort s in the RHS of a production $n \to R^*$ is represented by an attribute l of node n and of type s.

The requirement in Def. 5.4 and Item 5 that the non-terminals N are disjoint from the set of sorts S simplifies the destinction between the labels that become edges and those labels that become node attributes in the type graph in Items 2 and 3 and Def. 5.7 and Items 3c and 3d.

Definition 5.7 (EBNF Type Graph) Given an EBNF grammar with labels $G = (N, n_s, T, L, S, (C_s)_{s\in S}, P)$. *The type graph* $TG_G^{DSIG} = (TG, Z)$ *of* G is given by

1. an algebraic data signature $DSIG = (S, OP)$ with sorts S and an arbitrary set of operations OP,
2. the final $DSIG$-algebra $Z = ((Z_s)_{s\in S}, OP_Z)$ with $Z_s = \{s\}$ for each $s \in S$, and
3. an E-graph $TG = (V_{TG}\ , V_D, E_{TG}, E_{NA}, E_{EA}\ , (source_j, target_j)_{j\in\{TG,NA,EA\}})$ with
   (a) $V_{TG} = N$,
   (b) $V_D = \cup_{s\in S} Z_s$,
   (c) $E_{TG} = \{l \mid l \in L, t(l) \in N\}$,
   (d) $E_{NA} = \{l \mid l \in L, t(l) \in S\}$,
   (e) $E_{EA} = \varnothing$,
   (f) $source_j\colon E_j \to V_{TG}$ with $source_j(e) := s(e)$,
   (g) $target_{TG}\colon E_{TG} \to V_{TG}$ with $target_{TG}(e) := t(e)$, and

(h) $target_{NA} \colon E_{NA} \to V_D$ with $target_{NA}(e) := t(e)$ where $\{t(e)\} = Z_{t(e)}$.

For the definitions of $s(L)$ and $t(L)$ we refer to Def. 5.5. △

Example 5.8 (EBNF Type Graph) *The EBNF type graph of the grammar in Ex. 5.6 is given below. The type graph represents the structure of the grammar where*

1. *all non-terminals N of the grammar become nodes in the graph (e.g., non-terminal PRINT becomes node PRINT),*

2. *all labelled references to non-terminals in the grammar become edges in the graph with corresponding non-terminals nodes as source and target (e.g., labelled reference $(cl1, CLASS)$ in productions $CL\_LST \to \ldots$ becomes edge cl1 with source node CL_LST and target node CLASS), and*

3. *all labelled references to sorts in the grammar become attributes of corresponding non-terminal nodes in the graph. Furthermore, the type of each attribute is given by the corresponding referrenced sort (e.g., labelled reference $(name1, STRING)$ in productions $CLASS \to \ldots$ becomes attribute name1 of node CLASS with type STRING).*

By Def. 5.7, the formal notation of the EBNF type graph from above is given as follows:

1. $S = \{STRING, VAR, INT, NULL\}$,

2. $Z = ((Z_s)_{s \in S}, OP_Z)$ *with* $Z_s = \{s\}$,

3. (a) $V_{TG} = N$,
   (b) $V_D = \{STRING, VAR, INT, NULL\}$,
   (c) $E_{TG} = \{first, n1, n2, n3, n4, n5, cl1, e, attr1, body, st, a, p, r, i, g, a2, a4, a5, out, in2, c, r2\}$,
   (d) $E_{NA} = \{name1, name2, name, type, line, a1, a3, cl2, in1, obj, attr2, l, r1, r3\}$,
   (e) $source_{TG}(n1) = source_{TG}(n2) = source_{TG}(cl1) = CL\_LST$ *for edges* $n1, n2, cl1$ *and* $source_{NA}(name3) = source_{NA}(type) = ATTR$ *for node attributes* $name, type$, *and*
   (f) $target_{TG}(n2) = target_{TG}(n5) = target_{TG}(body) = ST\_LST$ *for edges* $n2, n5, body$ *and* $target_{NA}(name3) = target_{NA}(type) = STRING$ *for node attributes* $name, type$ *(the sources and targets of the other edges and node attributes are given analogously).* △

The visual notation of the EBNF type graph is given by the elements in (Root + Statements + Classes) in Sec. 5.1.2 and Fig. 5.5.

In order that all graphs typed over an EBNF type graph are actually graph representations of an EBNF grammar's derivation tree, the graphs additionally need to be restricted to obtain tree-structures. These restrictions are expressed by EBNF graph constraints which are derived from the EBNF grammar. While the EBNF type graph defines the overall structure of the EBNF grammar and the domains of labelled terminals, the EBNF graph constraints ensure that the syntax graphs are actually trees and represent the grammar structure, i.e.,

1. For the start symbol of the EBNF grammar there exists exactly one root node with no incoming edge,

2. for each grammar production rule there exists exactly one node with at least one outgoing and incoming edge for each edge type, and

3. each syntax graph has no cycles.

The graph constraints are defined based on the notion of graphs over a set of labels. Given a set of labels L such that all labels in L have the same source n, then the graph over L is given by a node :n of type n together with an edge for each label with :n as source node and the target of the label as type of the dedicated target node. Labels with non-terminals as targets become graph edges whereas labels with sorts as targets become node attribute edges in the graph with unique variables as attribute values (cf. Ex. 5.9).

Definition 5.8 (Graph over a set of Labels) Given an EBNF grammar with labels $G = (N, n_s, T, L, S, (C_s)_{s\in S}, P)$ and the corresponding EBNF type graph TG_G^{DSIG}. Let $E \in \mathscr{P}(L)$ be a set of labels with same source n, i.e., for all labels $e_1, e_2 \in E$ it is true that $s(e_1) = s(e_2) = n$ (cf. Def. 5.5). A *graph over a set of labels E* is given by an attributed graph $G_E = (G, T_{DSIG}((X_s)_{s\in S}))$

1. that is typed over TG_G^{DSIG},
2. with $DSIG$-term-algebra $T_{DSIG}((X_s)_{s\in S})$ with an infinite, countable set $X_s = \{x_i \mid i \in \mathbb{N}\}$ of variables for each sort s, and
3. with graph $G = G(E, n, 1)$ which is composed as follows based on graphs R_n, R_e and $R_{e,i}$ in Fig. 5.8 2).

$$G(E,n,i) := \begin{cases} R_n & \text{if } E = \varnothing, \\ R_e +_{R_{s(e)}} G(E \setminus e, s(e), i) & \text{if } t(e) \in N, \\ R_{e,i} +_{R_{s(e)}} G(E \setminus e, s(e), i+1) & \text{if } t(e) \in S \end{cases}$$

△

Remark 5.7 (Uniqueness of Graphs over Labels) *Note that the construction in Def. 5.8 may lead to a set S of different graphs for a given set of labels E, since, the selection of labels during the construction is non-determinstic which may lead to different variables as node attribute values. However, S is an isomorphism class, i.e., all graphs in S are unique up to isomorphisms, and technically Def. 5.8 is defined based on representatives of these classes only.* △

Remark 5.8 (Visual Notation of Graph Morphisms) *By $A +_I B$, we denote the gluing of graphs A and B over a common interface graph I. Technically, the gluing is given by a pushout over morphisms $I \to A$ and $I \to B$. The morphisms for the recursive gluing of graphs in Def. 5.8 are given by the names of the nodes and edges in the graphs. Thus, in addition to Rem. 5.4, nodes and edges may have names. Names are written before the colon. Thus, the notation 1:n means that the node or edge has the name 1. For a morphism $I \to A$ between two arbitrary graphs I and A, the names are used to indicate the mappings along the morphism. For example in Fig. 5.8, node 1:n in graph R_n is explicitly mapped to the source node 1:n of edge :e in graph R_e along morphism $R_n \to R_e$ and is not mapped to node :t(e) which would also be possible for $t(e) = n$ but is not intended, as, this would change the desired semantics of the EBNF graph constraints in Def. 5.9 and Item 2. Furthermore in contrast to Rem. 5.4, for node attributes in graphs of EBNF graph constraints we use the explicit E-graph notation in Def. 5.9, i.e., node attributes are explicit edges from the corresponding node to an explicit data node as attribute value.* △

Example 5.9 (Graph over a set of Labels) *Given the set of labels $L = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ where all labels have the same source ($n = s(a_1) = \ldots = s(a_n) = s(b_1) = \ldots = s(b_m)$), the targets of $a_1, \ldots, a_n$ are non-terminals ($t(a_1), \ldots, t(a_n) \in N$) and the targets of $b_1, \ldots, b_m$ are sorts ($t(b_1), \ldots, t(b_m) \in S$). Then, the graph over L is as follows.*

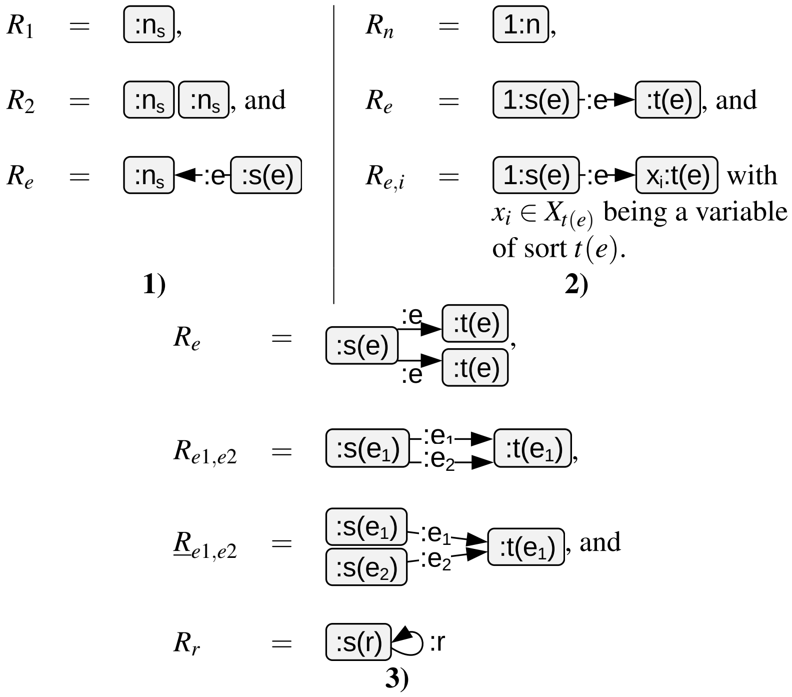
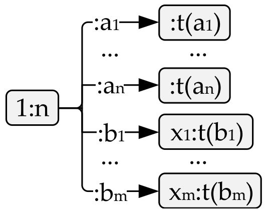

R1 = :ns,
R2 = :ns :ns, and
Re = :ns :e :s(e)
1)
Rn = 1:n,
Re = 1:s(e) :e :t(e), and
Re,i = 1:s(e) :e xi:t(e) with
xi ∈ Xt(e) being a variable
of sort t(e).
2)
Re = :s(e) :e :t(e), :e :t(e)
Re1,e2 = :s(e1) :e1 :e2 :t(e1),
Re1,e2 = :s(e1) :s(e2) :e1 :e2 :t(e1), and
Rr = :s(r) :r
3)

Figure 5.8: Visual notation of graphs for graph constraints in Def. 5.9



Note that the node attribute values $(x_i)_{i\in\{1,\dots,m\}}$ are ensured to be unique by construction Def. 5.8, i.e., there does not exist two different attributes b_j, b_k with $j,k \in \{1,\dots,m\}, j \neq k$ of node 1:n having the same variable x_i as value in the graph. △

Remark 5.9 (Subsets of Labels & Induced Morphisms) *Given a set of labels L and a subset $L' \subseteq L$. Furthermore, let G_L be a graph over L and $G_{L'}$ a graph over L'. If L' is non-empty, then there exists a unique morphism $G_{L'} \to G_L$ that is induced by the labels, since, each label becomes an unique edge with a unique edge type in graphs G_L and $G_{L'}$ by Defs. 5.7 and 5.8, and morphisms are type preserving, i.e., nodes and edges in one graph must be mapped to nodes and edges of the same type in the other graph along morphisms.* △

The set of EBNF graph constraints w.r.t. an EBNF grammar is the union of three sets $C_{Root}, C_{Ref}, C_{Mul}$ of local constraints and a set $\overline{C}_G$ of global constraints. Def. 5.4 and Item 8 simplifies the definition of the set of EBNF graph constraints, as, this rectriction allows the unique denotation of the root node of each tree-like graph.

Definition 5.9 (EBNF Graph Constraints) Given an EBNF grammar with labels $G = (N, n_s, T, L, S, (C_s)_{s\in S}, P)$ and the corresponding EBNF type graph TG_G^{DSIG}. *The set of EBNF graph constraints C_G w.r.t. G* are given by sub-sets $C_G = C_{Root} \cup C_{Ref} \cup C_{Mul} \cup \overline{C}_G$ of graph constraints that are typed over TG_G^{DSIG} and share the same $DSIG$-term-algebra $T_{DSIG}((X_s)_{s\in S})$ with an infinite, countable set $X_s = \{x_i \mid i \in \mathbb{N}\}$ of variables for each sort s. The sub-sets are defined as follows based on the graphs in Fig. 5.8.

1. $C_{Root} = C_1 \cup C_2 \cup C_3$ with
   (a) $C_1 = \{\exists(\varnothing \to R_1, \textbf{true})\}$,
   (b) $C_2 = \{\neg\exists(\varnothing \to R_2, \textbf{true})\}$, and
   (c) $C_3 = \{\neg\exists(\varnothing \to R_e, \textbf{true}) \mid e \in L, t(e) = n_s\}$,

2. $C_{Ref} = C_1 \cup C_2 \cup C_3$ with
   (a) For $n \in N$, with $P_n = \{p \mid p \in P, p \text{ is of form } p\colon n \to R^*\}$ we denote the set of productions with non-terminal n as LHS. Furthermore, for production $p \in P_n$, with $E_p = \{e \mid e \in L, (e, \_) \text{ occurs in RHS } R^* \text{ of } p\}$ we denote the set of labels that occur in p with same source n and therefore must exist for each occurrence of n in a graph.

      $C_1 = \{\forall(\varnothing \to R_n, \vee_{p \in P_n}(\exists(R_n \to G_{E_p}, \textbf{true}))) \mid n \in N\}$ with G_{E_p} being the graph over E_p (cf. Def. 5.8) and $R_n \to G_{E_p}$ being the morphism as induced by the node names of the graphs in Fig. 5.8 (the node of R_n is mapped to the node which is source of the edges in G_{E_p}).

   (b) With $E_n = \cup_{p \in P_n}(E_p)$ we denote the set of labels that occur in productions with $n \in N$ as LHS and with $\mathscr{P}(E_n)$ we denote its power set. With $\mathscr{E}_n = \{E_p \mid p \in P_n\}$ we denote the class of sets of labels that occur in productions with $n \in N$ as LHS. Furthermore, with $\underline{\mathscr{P}}(E_n) = \{E \mid E \in \mathscr{P}(E_n) \setminus \mathscr{E}_n$, there does not exist $\mathscr{E} \in \mathscr{E}_n$ with $E \subseteq \mathscr{E}\}$ we denote the class of sets of labels whose combinations do not occur in productions with $n \in N$ as LHS and therefore, need to be forbidden in graphs.

      $C_2 = \{\forall(\varnothing \to R_n, \neg(\exists(R_n \to G_E, \textbf{true}))) \mid n \in N, E \in \underline{\mathscr{P}}(E_n)\}$ with G_E being the graph over E and $R_n \to G_E$ being the morphism as induced by the node names of the graphs in Fig. 5.8, and

   (c) Similarly to class $\mathscr{E}_n$ in Item 2b, with $\mathscr{E}_{n,E} = \{\mathscr{E} \mid \mathscr{E} \in \mathscr{E}_n, E \subset \mathscr{E}\}$ we denote the class of sets of labels that occur in productions with $n \in N$ as LHS and that enclose E. With $\underline{\underline{\mathscr{P}}}(E_n) = \{E \mid E \in \mathscr{P}(E_n) \setminus \mathscr{E}_n$, there exists $\mathscr{E} \in \mathscr{E}_n$ with $E \subset \mathscr{E}\}$ we denote the class of sets of labels E that only partially occur in productions with $n \in N$ as LHS and therefore, need to be extended to sets $\mathscr{E}_{n,E}$.

      $C_3 = \{\forall(\varnothing \to G_E, \vee_{E' \in \mathscr{E}_{n,E}}(\exists(G_E \to G_{E'}, \textbf{true}))) \mid n \in N, E \in \underline{\underline{\mathscr{P}}}(E_n)\}$ with G_E (or $G_{E'}$) being the graph over E (or E') and $G_E \to G_{E'}$ being the morphism as induced by the node names of the graphs in Fig. 5.8 for $E = \varnothing$. For E being non-empty, the morphism is uniquely induced by the labels E (cf. Rem. 5.9).

3. $C_{Mul} = C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5$ with
   (a) $C_1 = \{\neg\exists(\varnothing \to R_e, \textbf{true}) \mid e \in L\}$,
   (b) $C_2 = \{\neg\exists(\varnothing \to R_{e1,e2}, \textbf{true}) \mid e_1, e_2 \in L, t(e_1) \in N, t(e_1) = t(e_2), s(e_1) = s(e_2)\}$,
   (c) $C_3 = \{\neg\exists(\varnothing \to R_{e1,e2}, \textbf{true}) \mid e_1 \in L, t(e_1) \in S, e_1 = e_2\}$,
   (d) $C_4 = \{\neg\exists(\varnothing \to \underline{R}_{e1,e2}, \textbf{true}) \mid e_1, e_2 \in L, t(e_1) \in N, t(e_1) = t(e_2)\}$, and
   (e) $C_5 = \{\neg\exists(\varnothing \to R_r, \textbf{true}) \mid r \in L, s(r) = t(r)\}$,

4. $\overline{C}_G$: For each production rule $p \in P$ in G construct a recursive graph schema GS_p according to Sec. 3.4 and Def. 3.18 such that the schema reflects all possible paths from the rule to

the start symbol of G along labels in G. Then, constraints $\overline{C}_G$ are given by the set of (weakened or tightened) recursive graph constraints w.r.t. GS_p, for all $p \in P$. Schema $GS_p = ((S',P'),M,s_{GS_p},t_{GS_p})$ for $p\colon n \in N \to R^*$ is constructed as follows:

(a) start graph $S' := \boxed{\boxed{:\mathsf{n}}}$ is given by node $:\mathsf{n}$ with non-terminal n as node type and $out(S') := \boxed{\boxed{:\mathsf{n}}}$,

(b) productions P' and matches M are given by $(P',M) = \mathscr{S}(\{(p,S')\},\varnothing)$ with

$$\mathscr{S}(H,MP') := \begin{cases} \mathscr{S}(\{(p'',\overline{p}'') \mid (p'',\overline{p}'',\_,\_) \in MP''\}, MP' \cup MP'') & \text{,if } MP'' \neq \varnothing \\ (\{\overline{p}'' \mid (\_,\overline{p}'',\_,\_) \in MP'\}, \{\overline{m} \mid (\_,\_,\_,\overline{m}) \in MP'\}) & \text{,otherwise} \end{cases}$$

where $MP'' = \{(p'',\overline{p}'',(l,n),\overline{m}) \mid (p\colon n \in N \to \_,\overline{p}) \in H, p''\colon n' \in N \to R'^* \in P, (l,n) \in R'^*, (p'',\_,(l,n),\_) \notin MP'\} \cup \{(p'',\overline{p}'',(l,n),\overline{m}) \mid (p\colon n \in N \to \_,\overline{p}) \in H, p''\colon n' \in N \to R'^* \in P, (l,n) \in R'^*, (p'',\overline{p}'',(l,n),\_) \in MP', (p'',\overline{p}'',(l,n),\overline{m}) \notin MP'\}$ with productions $\overline{p}'' := \boxed{\boxed{1{:}\mathsf{n}} \xleftarrow{\overset{++}{\mathbf{2{:}l}}} \overset{++}{\boxed{\mathbf{3{:}n'}}}}$ where $out(\overline{p}'')$ is given by node $3:\mathsf{n}'$ and matches $\overline{m} := (1 \mapsto out(\overline{p}))$ with $s_{GS_p}(\overline{m}) := \overline{p}''$ and $t_{GS_p}(\overline{m}) := \overline{p}$

Note that G may contain several production rules with the same non-terminal $n \in N$ as LHS. The schemata for these production rules are combined in the sense that we construct the disjunction over the corresponding recursive graph constraints for $\overline{C}_G$. △

An EBNF type graph together with a set of EBNF graph constraints form the meta-model for the graph language of derivation trees. The elements of this language are called derivation graphs.

Definition 5.10 (Derivation Graph) Let $G = (N,n_S,T,L,S,(C_s)_{s\in S},P)$ be an EBNF grammar with labels, TG_G^{DSIG} be the corresponding EBNF type graph and C_G be the set of EBNF graph constraints w.r.t. G. Then, *a derivation graph* of G is an attributed graph $DG = (\underline{DG},A)$ such that

1. DG is typed over TG_G^{DSIG},
2. DG satisfies the constraints C_G ($DG \models C_G$), and
3. A is a $DSIG$-algebra with carrier sets $(C_s)_{s\in S}$. △

Example 5.10 (Derivation Graph) *Note that the coding of EBNF grammars into EBNF type graphs in Def. 5.7 neglects unlabelled terminals T of the grammar to be part of derivation graphs whereas labelled terminals $(C_s)_{s\in S}$ may be contained in derivation graphs as node attribute values. Sec. 5.1.2 and Fig. 5.4 depicts a derivation graph of the EBNF grammar in Ex. 5.6 with the EBNF type graph in Ex. 5.8 and corresponding EBNF graph constraints.* △

Definition 5.11 (Graph Language of Derivation Trees) Let G be an EBNF grammar with labels. Then, *the graph language $\mathscr{L}(G)$ of derivation trees of* G is given by all derivation graphs of G according to Def. 5.10. △

If all non-terminal in an EBNF grammar G with labels are reachable from the start rule of the grammar, then the language of derivation trees of G is isomorphic to the graph language of derivation trees of G, i.e., according to Def. 5.2 there is a bijective mapping between both languages such that for each derivation tree over G there is a corresponding graph in the graph language. A non-terminal is reachable from the start rule in the grammar, if there is a path via from the start rule to the corresponding non-terminal which induces a derivation tree. Note that for context-free grammars, the detection of unreachable non-terminals is decidable and their elimination terminates for grammars with a finite set of grammar rules, a finite set of non-terminals and where each grammar rule is finite (cf. Sec. 7.1.1. in [HMU03]).

Claim 5.1 (Isomorphism of Derivation Tree Languages) *Let G be an EBNF grammar with labels such that all non-terminals are reachable from the start rule of the grammar. Then, according to Def. 5.2 the language $Der(G)$ of derivation trees of G and the graph language $\mathscr{L}(G)$ of derivation trees of G are isomorphic.* △

5.1.5 Completeness of Software Transformations

Based on the "classical" syntactical correctness and completeness of model transformations by triple graph grammars based on forward translation rules (cf. Cor. 8.5 in [EEGH15] and Sec. 2.4), the completeness of software translations from Def. 5.3 can be redefined as follows. While Def. 5.3 reflects the intuitive meaning behind complete translations, Thm. 5.1 expresses completeness in terms of a language inclusion which can be verified by using the verification techniques for domain completeness in Sec. 4.1 and Thm. 4.1. Therefore, both formulations of completeness in Def. 5.3 and Thm. 5.1 are equivalent.

Theorem 5.1 (Syntactical Completeness of Software Translations) *The translation is syntactically complete according to Def. 5.3 if and only if $\mathscr{L}(TG^S, C^S) \subseteq \mathscr{L}(TGG)^S$.* △

Proof. The proof is analogue to the proof of Sec. 4.1 and Thm. 4.1. □

Example 5.11 (Syntactical Completeness of Software Translations) *Given the translation of Conditional-IN-OUT programs into UML class diagrams from Sec. 5.1.2 and Ex. 5.5 that are performed by model transformation MT based on the forward translation rules of the TGG in Sec. 5.1.2 and Ex. 5.5 that are typed over the EBNF type graph in Sec. 5.1.4 and Ex. 5.8 in the source component, respectively. Furthermore, given corresponding EBNF graph constraints as source domain constraints and the corresponding graph language of derivation trees whose graphs should be translated by model transformation MT. Note that the translation only covers aspects of class diagrams in the programs. Therefore, based on Thm. 5.1, the syntactical completeness of the translation can be successfully verified by using the notion of domain completeness of model transformation MT under restrictions and corresponding verification techniques in Sec. 4.1 and Thm. 4.3 where the restriction only covers the aspects of class diagrams. It turns out that all aspects of class diagrams in Conditional-IN-OUT programs - including* Class *definitions with* Attr*ibutes, inheritance relationships and* associ*ations - can be translated to UML class diagrams.* △

5.2 Completeness of Software Synchronisations

A software synchronisation is syntactically complete if each update on a source derivation graph $G^S \in \mathscr{L}(TG^S, C^S)$ leading to a graph G'^S can be completely propagated to the target domain in the sense that all elements of G'^S are in correspondence with elements in the target domain, i.e., all elements in G'^S are translated to elements in the target domain.

Definition 5.12 (Syntactical Completeness of Software Synchronisations) Given a context-free word grammar G^S for source language $\mathscr{L}^S$ with induced language $Der(G^S)$ of derivation trees.

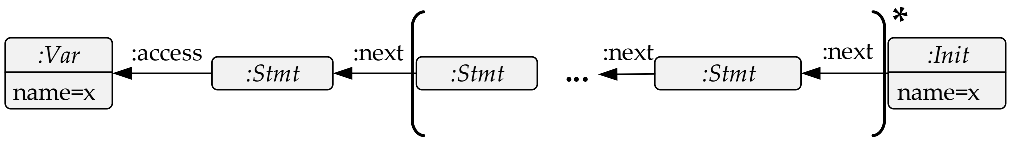


Figure 5.9: Recursive Graph Constraint - "Each variables is initialised before being accessed"

Let $TG = (TG^S \leftarrow TG^C \rightarrow TG^T)$ be a triple type graph with $\mathscr{L}(C^S)$ being the source graph language of derivation trees isomorphic to $Der(G^S)$. Furthermore, let $TGG = (\varnothing, TR)$ be a triple graph grammar typed over TG with derived forward translation rules TR_{FT} that specifies the model transformation of graphs in $\mathscr{L}(C^S)$ into graphs of target graph language $\mathscr{L}(C^T)$ based on forward translation rules TR_{FT}. Let $u\colon M^S \rightarrow M'^S$ with $M^S, M'^S \in \mathscr{L}(C^S)$ be a model update from a model M^S to a model M'^S both in the source graph language of derivation trees $\mathscr{L}(C^S)$. *The synchronisation is syntactically complete*, if for each such update u and triple graph $M = (M^S \leftarrow M^C \rightarrow M^T)$ the forward propagation operation $fPpg(M,u) = (M',u')$ leads to an update $u'\colon M^T \rightarrow M'^T$ in the target domain and integrated model $M' = (M'^S \leftarrow M'^C \rightarrow M'^T)$ such that there is a model transformation sequence $(M'^S, M_0 \xRightarrow{tr^*_{FT}} M_n, M'^T)$ based on the forward translation rules of TGG with $M_0 = (Att^{\mathbf{F}}(M'^S) \leftarrow \varnothing \rightarrow \varnothing)$ and $M_n = (Att^{\mathbf{T}}(M'^S) \leftarrow M'^C \rightarrow M'^T)$. △

Based on the "classical" syntactical completeness and correctness of model transformations and synchronisations by triple graph grammars based on forward translation rules (cf. Cor. 8.5 & Thm 9.25 in [EEGH15] and Sec. 2.4) and the decomposition property of TGGs, the completeness of software synchronisations can be reformulated as follows. While Def. 5.12 reflects the intuitive meaning behind complete synchronisations, Prop. 5.1 expresses completeness in terms of a language inclusion which can be verified by using the verification techniques for domain completeness in Sec. 4.1. Therefore, both formulations of completeness in Def. 5.12 and Prop. 5.1 are equivalent.

Proposition 5.1 (Syntactical Completeness of Software Synchronisations) *Let $Synch(TGG)$ be the derived TGG synchronisation framework with forward propagation operation fPpg such that the sets of operational translation rules derived from TGG are kernel-grounded and deterministic. Then, the synchronisation via fPpg is syntactically complete according to Def. 5.12 if and only if $\mathscr{L}(C^S) \subseteq \mathscr{L}(TGG)^S$.* △

Proof. The proof is analogue to the proof of Sec. 4.2 and Thm. 4.4. □

5.3 Completeness of Static Semantics

In the following, we focus on static semantics of models that can be expressed by (infinite) graph constraints on the structure of graphs by using the concept of recursive graph constraints from Sec. 3.4. For example, for abstract syntax trees of source code in Sec. 5.1, we can define a recursive graph schema which leads to the infinite recursive graph constraint as indicated in Fig. 5.9. The constraint expresses the requirement that each variable Var of name x that is accessed by some statement Stmt in the program has to be Initialised before. The path from the accessed variable to its initialisation is expressed by a regular path with an arbitrary number of statements in between by a recursive graph schema and induced infinite constraint. By the construction in Sec. 3.4, we can construct a finite graph constraint from the infinite one which then can be used within the verification of completeness of software transformations in Sec. 5.1.

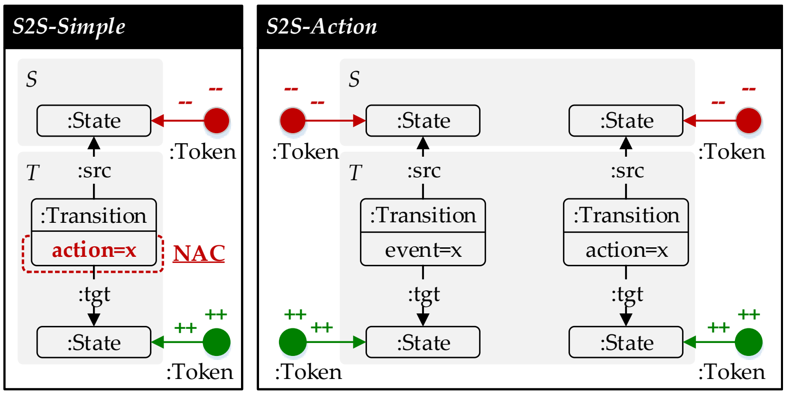


Figure 5.10: Operational Semantics of UML Statecharts

5.4 Completeness of Operational Semantics

In the following, we focus on operational semantics of models that are defined by a set of graph transformation rules. In Fig. 5.10, two rules are presented that define aspects of the operational semantics of UML statecharts from Sec. 1.1. Rule S2S – Simple defines the Transition from on State to another state in the statechart but only if the transition performs no action when being fired as ensured by the negative application condition (NAC). When being applied, the rule moves the Token from the source (src) state of the transition to the target (tgt) state by deleting the active token and creating a new token and assigned it to the target state. Similarly, we define rule S2S – Action that performs a transition from a source to a target state if the transition performs an action when being fired. Simultaneously, the rule fires the transition with that event which is in correspondence with the action.

Therefore, both rules define a token semantics by replacing tokens. The deleted token points to the source part of the graph (marked with S) and the created token points to the target part of the graph (marked with T). We can intepret both rules as non-deleting rules where the S has to exist and the T part is being created by the rule. Therefore, we can apply the verification of domain completeness in order to verify if all elements in UML statecharts are covered by the operational semantics. As the verification is restricted to graph grammars with empty start graphs only, an additional rule has to be defined which creates the initial token to the initial state of the statechart. The verification returns minimal examples of statechart excerpts that cannot be completely covered by the semantics. This can be taken as the basis for the definition of a set of domain constraints that successively restrict the structure of statecharts such that they can be completely simulated. The constraints may serve as guidelines for modellers, e.g., in order to avoid statecharts with deadlocks. Note that in the case of statecharts, the constraints will only allow to represent single processes without choices in the statechart. However, single traces can simply be combined to more complex statecharts with choices.

Chapter 6

Conclusion, Related & Future Work

We have stated the domain completeness problem which focuses on the relationship between nested graph constraints and graph grammars. More precisely, domain completeness claims that a given set of nested graph constraints is more restrictive on the graph structure than (or as restrictive as) a given graph grammar. Therefore, domain completeness enlightens the relationship between the descriptive approach of specifying graph languages via graph constraints and the operational approach via graph grammars.

In Sec. 3.1, we proved the undecidability of the domain completeness problem for plain graphs and derivative categories of graphs. The undecidability of the problem led to the development of an under-approximative verification approach of domain completeness in Sec. 3.2. This approach is applicable in the $\mathcal{M}$-adhesive category $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ of typed attributed graphs with node type inheritance. The termination of the verification is ensured by an upper bound of the graph size. By doing this, also a complete check of all graphs up to the given upper bound can be performed for verifying the language inclusion. However, in most cases, the presented approach is more efficient and terminates without an explicit upper bound. In Sec. 3.3 the limitations of the approach are discussed, i.e., the approach is only applicable to: 1. graph grammars with empty start graph, 2. graph grammars with non-deleting productions, 3. graph grammars with application conditions in $\mathcal{M}$-normal form, 4. domain constraints in $\mathcal{M}$-normal form, and 5. domain constraints that are designated for general satisfaction. However, this setting is suitable for model transformations and synchronisations based on the theory of triple graph grammars (TGGs).

Consequently in Chap. 4, the results of domain completeness from Sec. 3.2 are applied to verify the domain completeness of model transformations and model synchronisations.

In Chap. 5, further applications for the presented verification approach are discussed. In Sec. 5.1, a coding of context-free word grammars in EBNF notation into type graphs and graph constraints is given such that the language of derivation trees over the EBNF grammar is equivalent (isomorphic) to the graph language over the derived type graph and graph constraints. This result can be used for the verification of domain completeness of software translations and synchronisations (cf. also Sec. 5.2). Apart from that, the verification of domain completeness of software translations and synchronisations are direct extensions of the results for verifying domain completeness of model transformations and synchronisations. Due to the generality of the results and the existing formal framework, these immediate steps are possible.

If the EBNF grammar of the programming language of programs that are translated is ambiguous, then there may exist several derivation trees for each program written in the language. Therefore, the approach for verifying the completeness of software translations is particularly suitable for unambiguous EBNF grammars. For the coding in Sec. 5.1, graph constraints are needed that are able to express infinite structures in graphs, i.e., regular paths. Therefore, we developed recursive graph constraints in Sec. 3.4 and showed how they can be used for domain completeness verifications that terminate by presenting a construction for deriving finite constraints from infinite recursive graph constraints. Recursive graph schemata and derived recursive graph constraints can also be used for specifying static semantics of models, e.g., static semantics of programs as discussed in Sec. 5.3. Furthermore, in Sec. 3.5, we discuss how to restrict the source domain of model transformations and synchronisations to relevant domain elements. In particular, this involves the restriction of the domain type graph and type constraints to relevant domain elements only. In Sec. 5.1, we use the results of domain restrictions in order to translate only the class definitions in source code to UML class diagrams while neglecting the other syntactic aspects in the programs. In Sec. 5.4, we address that the presented approach for verifying domain completeness can also be used to verify the completeness of operational (dynamic) token semantics. The operational semantics are given by a set of graph transformation rules where models are simulated by applying the rules. The rules assign a token to some part of the graph (model) and re-assign it to another part of the graph, therefore performing a semantic step. The presented approach is not directly applicable to operational token semantics, since, the rules are deleting. However, based on a given example it is clarified how deleting token rules can be interpreted as non-deleting rules such that the approach is applicable.

Nested graph constraints were introduced in [HP09] in order to ensure that a given graph grammar is more restrictive on the graph structure than (or as restrictive as) a given set of nested graph constraints, i.e., the opposite direction of domain completeness. More precisely, a given set of nested graph constraints is translated into application conditions of rules of a given graph grammar such that it is guaranteed that all transformations via the rules of the grammar lead to graphs that satisfy the graph constraints. A similar approach to recursive graph constraints in Sec. 3.4 was introduced by adaptive star grammars [DHJ$^+$06]. However, to the best of our knowledge there is no construction to obtain finite constraints from infinite ones which is essential for the termination of the verification of domain completeness. The results for domain restrictions in Sec. 3.5 are basically an extension of the results in [SEM$^+$12] from initial to general satisfaction of graph constraints. The results for domain completeness in Chap. 4 are based on the formal frameworks of model transformations and synchronisations in [EEGH15] which are originated from the delta-lens framework [DXC10a, DXC$^+$11, DXC10b, JR15]. In view of Sec. 5.1.4, while context-free (EBNF) word grammars allow a generative approach to obtain the language of all syntax trees that are derivable from the grammar, type graphs together with graph constraints, together forming a language meta-model, allow a declarative approach for defining this language of abstract syntax graphs. For existing work on deriving meta-models from word grammars we refer to [BW13].

In future work the approach should be extended from the $\mathcal{M}$-adhesive category $(\mathbf{AGraphs}_{ATGI}, \mathcal{M})$ of typed attributed graphs with node type inheritance to the general framework of $\mathcal{M}$-adhesive categories which comprises a variety of other (graph-like) structures. Basically, this includes the development of a notion of boolean-valued marking of objects in $\mathcal{M}$-adhesive categories which was started in [HEOG10] and need to be extended to triple rules with application conditions. As the presented verification approach is an approximation only, the approach is open for optimisations with regard to different application scenarios. Possible starting points for optimisations are presented by the limitations of the approach in Sec. 3.3. The extension of the approach to graph grammars with deleting productions seems to be of most importance while an extension to grammars with non-empty start graphs and application conditions not in $\mathcal{M}$-normal form seems to be interesting but of less importance. Finally, the approach for verifying domain completeness of model transformations and synchronisations can be extended to a multi-model environment [TA15] without dedicated source and target models but with an arbitrary number of interlinked models. The sufficient conditions for ensuring the language inclusion of restricted graph constraints and their originals are rather strict and may be refined in future work. Finally, the entire approach for verifying domain completeness should be evaluated based on a case study of relevant size with the help of an implementation in future work. The existing HenshinTGG tool [Hen15, HNB$^+$14] represents a suitable basis for implementing the approach in order to enable automatic tool support for verifying domain completeness in the future. The PIL2SPELL project [HGN$^+$14] seems to be an appropriate real world scenario for evaluating the approach concerning the verification of the completeness of software translations. In PIL2SPELL, satellite control procedures were translated between different programming languages.

Appendix A

Detailed Proofs

A.1 Proof of Sec. 2.2.3 and Prop. 2.1

We prove the result for positive conditions at first and for non-positive conditions afterwards. By induction over the structure of nested conditions:

$$\begin{tikzcd}
P \arrow[rrrr, "a"] \arrow[drr, "o_3"] \arrow[ddr, "e_p"'] \arrow[ddd, "p"'] & & (2) & & C \arrow[r, phantom, "\lhd"] \arrow[ddl, "e'_p"] \arrow[dddllll, bend left, "q"] & ac_C \\
& (3) & O \arrow[urr, "o_1"] \arrow[dl, "o_2"] & (1) & & \\
& P' \arrow[rr, "a'"] \arrow[dl, "m_p"] & & C' \arrow[dlll, "m_q"] & & \\
G & & & & &
\end{tikzcd}$$

"⇒" **Basis.** For $ac_P = \textbf{true}$, let $m_p \circ e_p = p$ be the extremal $\mathscr{E}$-$\mathscr{M}$ factorisation of p with $(e_p, \textbf{true}) \in \text{Inst}(ac_P)$. It holds that $m_p \models \textbf{true}$ and therefore, $p \models \exists(e_p, \textbf{true})$ implying further that $p \models \overline{ac}_P$. For $ac_P = \exists(a\colon P \to C \in \mathscr{M}, \textbf{true})$, $p \models ac_P$ implies that there is $q \in \mathscr{M}$ with $q \circ a = p$. Let $m_p \circ e_p = p$ be the extremal $\mathscr{E}$-$\mathscr{M}$ factorisation of $p \in \mathscr{M}$ with $e_p \in \mathscr{M}$ by $\mathscr{M}$-decomposition. We construction pullback (o_1, o_2) over (m_p, q) with $o_1, o_2 \in \mathscr{M}$, since, $\mathscr{M}$-morphisms $m_p, q \in \mathscr{M}$ are closed under pullbacks. We construct pushout (1) with $a', e'_p \in \mathscr{M}$, since, $\mathscr{M}$-morphisms $o_1, o_2 \in \mathscr{M}$ are closed under pushouts. By effective pushouts, the induced morphism m_q is in $\mathscr{M}$ with $m_q \circ a' = m_p$ and $m_q \models \textbf{true}$ and therefore, $m_p \models \exists(a', \textbf{true})$. It remains to show that $\exists(a', \textbf{true})$ is in $\text{Merge}(e_p, ac_P)$ which would imply that $m_p \models \text{Merge}(e_p, ac_P)$ and furthermore, $p \models \overline{ac}_P$ by Rem. 2.13. By the universal pullback property, there is o_3 with (2) and (3) commute. By $(1), (2)$ and (3) commute, respectively, $(1) + (2) + (3)$ commutes. Furthermore, $a', e'_p \in \mathscr{M}$ and therefore, $e'_p \in \mathscr{O}$ according to Rem. 2.3. Moreover, pushout (1) implies that (a', e'_p) are jointly epimorphic and $\text{Merge}(e'_p, \textbf{true}) = \textbf{true}$ by construction. **Hypothesis.** The result holds for conditions ac_C and $ac_{P,i}, i \in I$. **Step.** For $ac_P = \exists(a\colon P \to C \in \mathscr{M}, ac_C)$, we conclude for morphism a like before with induced morphism $m_q \in \mathscr{M}$ and $m_q \circ e'_p = q$. By Lem. 3.11, $e'_p \in \mathscr{E}$ and therefore, $m_q \circ e'_p$ is an extremal $\mathscr{E}$-$\mathscr{M}$ factorisation of q. Thus, $p \models ac_P$ implies $q \models ac_C$ implying further by induction hypothesis that $q \models \overline{ac}_C$ and therefore by Rem. 2.13, $m_q \models \text{Merge}(e'_p, ac_C)$. Therefore, $m_p \models \text{Merge}(e_p, ac_P)$ and furthermore, $p \models \overline{ac}_P$ by Rem. 2.13. For $ac_P = \vee_{i \in I}(ac_{P,i})$, $p \models ac_P$ implies $p \models ac_{P,i}$ for some $i \in I$. By induction hypothesis, $p \models \overline{ac}_{P,i}$ implying further that $m_p \models \text{Merge}(e_p, ac_{P,i})$ for extremal $\mathscr{E}$-$\mathscr{M}$ factorisation $m_p \circ e_p$ of p by Rem. 2.13. Thus, $m_p \models \vee_{i \in I}(\text{Merge}(e_p, ac_{P,i})) = \text{Merge}(e_p, ac_P)$

and therefore, $p \models \overline{ac_P}$ by Rem. 2.13. Analogously, we prove the fact for $ac_P = \wedge_{i\in I}(ac_{P,i})$.

"$\Leftarrow$" **Basis.** For $ac_P = \mathbf{true}$, $p \models ac_P$. For $ac_P = \exists(a\colon P \to C \in \mathscr{M}, \mathbf{true})$, $p \models \overline{ac_P}$ implies $m_p \models \mathrm{Merge}(e_p, ac_P)$ for extremal $\mathscr{E}$-$\mathscr{M}$ factorisation $m_p \circ e_p = p$ of p by Rem. 2.13. Therefore, $m_p \models \exists(a', \mathbf{true})$ for some commuting diagram $(1)+(2)+(3)$ with $a' \in \mathscr{M}$ and $e'_p \in \mathscr{O}$ by construction Def. 2.16, i.e., there is $m_q \in \mathscr{M}$ with $m_q \circ a' = m_p$. It remains to show that $e'_p \in \mathscr{M}$, since, by $\mathscr{M}$-composition it would follow that $m_q \circ e'_p \in \mathscr{M}$ and furthermore, $m_q \circ e'_p \circ a = m_q \circ a' \circ e_p = m_p \circ e_p = p$, i.e., $p \models ac_P$. By $\mathscr{M}$-decomposition of $p \in \mathscr{M}$, $e_p \in \mathscr{M}$ and therefore by $\mathscr{M}$-composition, $a' \circ e_p = e'_p \circ a \in \mathscr{M}$. By $e'_p \in \mathscr{O}$ and Rem. 2.3, $e'_{p,G}$ is componentwise injective except perhaps for the data part $e'_{p,D}$. By $a, e'_p \circ a \in \mathscr{M}$ and Rem. 2.3 it follows that a_D and $e'_{p,D} \circ a_D$ are isomorphisms with inverse isomorphisms a_D^{-1} and $(e'_{p,D} \circ a_D)^{-1}$. Thus, $a_D^{-1} \circ a_D \circ (e'_{p,D} \circ a_D)^{-1} \overset{a_D \text{ is iso}}{=} (e'_{p,D} \circ a_D)^{-1} \overset{e'_{p,D} \circ a_D \text{ is iso}}{=} (e'_{p,D} \circ a_D)^{-1} \circ e'_{p,D} \circ a_D \circ (e'_{p,D} \circ a_D)^{-1}$. By $a_D, (e'_{p,D} \circ a_D)^{-1}$ are isomorphisms, composition $a_D \circ (e'_{p,D} \circ a_D)^{-1}$ is an iso- and therefore, also epi-morphism, i.e., $a_D^{-1} = (e'_{p,D} \circ a_D)^{-1} \circ e'_{p,D}$. Therefore, $a_D \circ (e'_{p,D} \circ a_D)^{-1} \circ e'_{p,D} = a_D \circ a_D^{-1} \overset{a_D \text{ is iso}}{=} id_{D_C}$ and furthermore, $e'_{p,D} \circ a_D \circ (e'_{p,D} \circ a_D)^{-1} = id_{D_{C'}}$. Thus, $e'_{p,D}$ is an isomorphism. Furthermore, e'_p is type strict for all nodes $a(P)$ in C, since, $e'_p \circ a \in \mathscr{M}$ is type strict, and furthermore e'_p is type strict for all nodes $n \in C \setminus a(P)$ in C that are not in P, since, condition ac_P is type strict and therefore, the type of node n cannot be refined along e'_p. Thus according to Rem. 2.3, e'_p is in $\mathscr{M}$. **Hypothesis.** The result holds for conditions ac_C and $ac_{P,i}, i \in I$. **Step.** For $ac_P = \exists(a\colon P \to C \in \mathscr{M}, ac_C)$, we conclude analogously to the base case from before where $p \models \overline{ac_P}$ implies $m_p \models \exists(a', \mathrm{Merge}(e'_p, ac_C))$ for some commuting diagram $(1)+(2)+(3)$ with (a', e'_p) being jointly epimorphic by construction Def. 2.16, i.e., $m_q \models \mathrm{Merge}(e'_p, ac_C)$. By Lem. 3.11, $e'_p \in \mathscr{E}$ and therefore, $m_q \circ e'_p$ is an extremal $\mathscr{E}$-$\mathscr{M}$ factorisation of itself. By Rem. 2.13, $m_q \circ e'_p \models \overline{ac_C}$ and furthermore by induction hypothesis, $m_q \circ e'_p \models ac_C$, i.e., $p \models ac_P$. For $ac_P = \vee_{i\in I}(ac_{P,i})$, $p \models \overline{ac_P}$ implies that $m_p \models \mathrm{Merge}(e_p, ac_P) = \vee_{i\in I}(\mathrm{Merge}(e_p, ac_{P,i}))$ by Def. 2.16 and Rem. 2.13 with extremal $\mathscr{E}$-$\mathscr{M}$ factorisation $m_p \circ e_p = p$. Thus, $m_p \models \mathrm{Merge}(e_p, ac_{P,i})$ for some $i \in I$ implying further that $p \models \overline{ac_{P,i}}$ by Rem. 2.13. By induction hypothesis, $p \models ac_{P,i}$ and therefore, $p \models ac_P$. For $ac_P = \wedge_{i\in I}(ac_{P,i})$, we conclude analogously.

For non-positive conditions $ac_P = \neg ac'_P$ and extremal $\mathscr{E}$-$\mathscr{M}$ factorisation $m_P \circ e_p = p$ we conclude as follows: $p \models ac_P \overset{Def.\ 2.12}{\Leftrightarrow} \neg(p \models ac'_P) \overset{Hypothesis}{\Leftrightarrow} \neg(p \models \overline{ac'_P}) \overset{Rem.\ 2.13}{\Leftrightarrow} \neg(m_p \models \mathrm{Merge}(e_p, ac'_P)) \overset{Def.\ 2.12}{\Leftrightarrow} m_p \models \neg\mathrm{Merge}(e_p, ac'_P) \overset{Def.\ 2.16}{=} \mathrm{Merge}(e_p, ac_P) \overset{Rem.\ 2.13}{\Leftrightarrow} p \models \overline{ac_P}$. □

A.2 Proof of Sec. 2.2.3 and Prop. 2.2

$$\begin{tikzcd}
P \arrow[rrrr, "a"] \arrow[rr, "x"] \arrow[dr, "e_p"] \arrow[dddd, "p"'] & & & & C \lhd ac_C \arrow[dd, "e_q"'] \arrow[ddddllll, bend left, "q"] \\
& (1) & O \arrow[dl, "o_1"'] \arrow[ddrr, "o_2"] & (2) & \\
& P' \arrow[dr, "o'_1"] \arrow[ddl, "m_p"] & (3) & & \\
& (4) & O' \arrow[ddll, "y"] & (5) & C' \arrow[ll, "o'_2"'] \arrow[dllll, "m_q"] \\
G & & & &
\end{tikzcd}$$

We prove the result for positive conditions at first and for non-positive conditions afterwards. By induction over the structure of nested conditions: Let $m_p \circ e_p = p$ be an extremal $\mathcal{E}$-$\mathcal{M}$ factorisation of $p \in \mathcal{O}$.

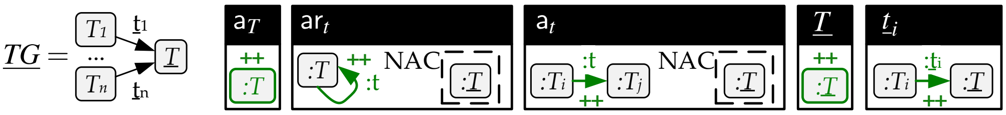
"$\Rightarrow$" **Basis.** For $ac_P = \textbf{true}$, $p \models \exists(e_p \in \mathcal{E}_P, \textbf{true})$. By construction Def. 2.18, $\overline{ac}_P = \vee_{e \in \mathcal{E}_P}(\exists(e, \textbf{true}))$ and therefore, $p \models \overline{ac}_P$. For $ac_P = \exists(a\colon P \to C, \textbf{true})$, $p \models_{\mathcal{O}} ac_P$ implies that there is $q \in \mathcal{O}$ with $q \circ a = p$. Let $m_q \circ e_q = q$ be an extremal $\mathcal{E}$-$\mathcal{M}$ factorisation of q. We construct pullback (o_1, o_2) over (m_p, m_q) with induced morphism x such that (1) and (2) commute. We construction pushout (3) with $o_1, o_2, o_1', o_2' \in \mathcal{M}$, since, $\mathcal{M}$-morphisms $m_p, m_q \in \mathcal{M}$ are closed under pullbacks and pushouts, and induced morphism $y \in \mathcal{M}$ such that (4) and (5) commute by effective pushouts. Thus, $m_p \models \exists(o_1', \textbf{true})$. It remains to show that $\exists(o_1', \textbf{true})$ is in $\text{Merge}(e_p, ac_P)$ which would imply that $m_p \models \text{Merge}(e_p, ac_P)$ implying further that $p \models \overline{ac}_P$ by Rem. 2.13. As already shown, $o_1' \in \mathcal{M}$. Furthermore, $(1)+(2)+(3)$ commutes, since $(1),(2)$ and (3) commute, respectively. By $q \in \mathcal{O}$ and Rem. 2.3, q_G is injective except perhaps for data part q_D, i.e., $m_q \circ e_q = q$ implies that $e_{q,G}$ is injective except perhaps for data part $e_{q,D}$. By $o_2' \in \mathcal{M}$, o_2' is injective and therefore, $o_{2,G}' \circ e_{q,G}$ is injective except perhaps for data part $o_{2,D}' \circ e_{q,D}$, i.e., $o_2' \circ e_q \in \mathcal{O}$. By Lem. 3.9 and Item 3 with $o_1' \in \mathcal{M}, e_q \in \mathcal{E}$ and (o_1', o_2') being jointly epimorphic by pushout (3), $(o_1', o_2' \circ e_q)$ is jointly epimorphic. Therefore, $\exists(o_1', \textbf{true})$ is in $\text{Merge}(e_p, ac_P)$. **Hypothesis.** The result holds for conditions ac_C and $ac_{P,i}, i \in I$. **Step.** For $ac_P = \exists(a\colon P \to C, ac_C)$, we conclude analogously to the base case from before. It remains to show that $y \models \text{Merge}(o_2' \circ e_q, ac_C)$ which would imply that $m_p \models \text{Merge}(e_p, ac_P)$ implying further that $p \models \overline{ac}_P$. By Lem. 3.11, $(1)+(2)+(3)$ being commuting, $e_p \in \mathcal{E}$ and $(o_1', o_2' \circ e_q)$ being jointly epimorphic, it follows that $o_2' \circ e_q \in \mathcal{E}$. Furthermore, by (5) commutes and the uniqueness of extremal $\mathcal{E}$-$\mathcal{M}$ factorisations with $y \in \mathcal{M}$, $y \circ (o_2' \circ e_q) = q$ is the extremal $\mathcal{E}$-$\mathcal{M}$ factorisation of q. By $p \models_{\mathcal{O}} ac_P$ it follows that $q \models_{\mathcal{O}} ac_C$ implying further by induction hypothesis that $q \models \overline{ac}_C$, i.e., $y \models \text{Merge}(o_2' \circ e_q, ac_C)$ by Rem. 2.13. For $ac_P = \vee_{i \in I}(ac_{P,i})$, $p \models_{\mathcal{O}} ac_P$ implies $p \models_{\mathcal{O}} ac_{P,i}$ for some $i \in I$. By induction hypothesis, $p \models \overline{ac}_{P,i} \overset{Rem.\ 2.13}{\Rightarrow} m_p \models \text{Merge}(e_p, ac_{P,i}) \Rightarrow m_p \models \vee_{i \in I}(\text{Merge}(e_p, ac_{P,i})) \overset{Def.\ 2.16}{=} \text{Merge}(e_p, ac_P) \overset{Rem.\ 2.13}{\Rightarrow} p \models \overline{ac}_P$. For $ac_P = \wedge_{i \in I}(ac_{P,i})$, we conclude analogously.

"$\Leftarrow$" **Basis.** For $ac_P = \textbf{true}$, $p \models_{\mathcal{O}} ac_P$. For $ac_P = \exists(a\colon P \to C, \textbf{true})$, $p \models \overline{ac}_P$ implies $m_p \models \text{Merge}(e_p, ac_P)$ by Rem. 2.13. Thus, by construction Def. 2.16, there is some diagram $(1)+(2)+(3)$ which commutes and with $o_2' \circ e_q \in \mathcal{O}$ and $y \in \mathcal{M}$ such that (4) commutes. Therefore, by Rem. 2.3, $y \circ o_2' \circ e_q \in \mathcal{O}$ and furthermore, $(1)+(2)+(3)+(4)$ commutes, i.e., $p \models_{\mathcal{O}} ac_P$. **Hypothesis.** The result holds for conditions ac_C and $ac_{P,i}, i \in I$. **Step.** For $ac_P = \exists(a\colon P \to C, ac_C)$, $p \models \overline{ac}_P$ additionally implies that $y \models \text{Merge}(o_2' \circ e_q, ac_C)$ and $(o_1', o_2' \circ e_q)$ are jointly epimorphic. By Lem. 3.11, $o_2' \circ e_q \in \mathcal{E}$ and furthermore by the uniqueness of extremal $\mathcal{E}$-$\mathcal{M}$ factorisations, $y \circ (o_2' \circ e_q)$ is the extremal $\mathcal{E}$-$\mathcal{M}$ factorisation of itself. Thus by Rem. 2.13, $y \circ o_2' \circ e_q \models \overline{ac}_C$ implying further by induction hypothesis that $y \circ o_2' \circ e_q \models_{\mathcal{O}} ac_C$, i.e., $p \models_{\mathcal{O}} ac_P$. For $ac_P = \vee_{i \in I}(ac_{P,i})$, $p \models \overline{ac}_P$ implies that $m_p \models \text{Merge}(e_p, ac_P) = \vee_{i \in I}(\text{Merge}(e_p, ac_{P,i}))$ by Def. 2.16 and Rem. 2.13. Thus, $m_p \models \text{Merge}(e_p, ac_{P,i})$ for some $i \in I$ implying further that $p \models \overline{ac}_{P,i}$ by Rem. 2.13. By induction hypothesis, $p \models_{\mathcal{O}} ac_{P,i}$ and therefore, $p \models_{\mathcal{O}} ac_P$. For $ac_P = \wedge_{i \in I}(ac_{P,i})$, we conclude analogously.

For non-positive conditions $ac_P = \neg ac_P'$ we conclude as follows: $p \models_{\mathcal{O}} ac_P \overset{Def.\ 2.12}{\Leftrightarrow} \neg(p \models_{\mathcal{O}} ac_P') \overset{Hypothesis}{\Leftrightarrow} \neg(p \models \overline{ac}_P') \overset{Rem.\ 2.13}{\Leftrightarrow} \neg(m_p \models \text{Merge}(e_p, ac_P')) \overset{Def.\ 2.12}{\Leftrightarrow} m_p \models \neg\text{Merge}(e_p, ac_P') \overset{Def.\ 2.16}{=} \text{Merge}(e_p, ac_P) \overset{Rem.\ 2.13}{\Leftrightarrow} p \models \overline{ac}_P$. □

A.3 Proof of Sec. 3.1 and Thm. 3.1

Let $\mathbf{C} = (\mathbf{Graphs}_{TG,fin}, \mathcal{M}_{fin})$ be the finitary $\mathcal{M}$-adhesive category of finite graphs typed over finite type graph TG together with the class $\mathcal{M}_{fin}$ of injective morphisms. Note that $\mathbf{C}$ has $\mathcal{E}$-$\mathcal{M}$-factorisation and $\mathcal{M}$-initial object $\varnothing$ such that Lem. 3.1 can be used. The same is true for the category $\mathbf{C}' = (\mathbf{Graphs}_{TG',fin}, \mathcal{M}_{fin})$ with $TG' \supseteq TG$. The undecidability is shown by a reduction from the undecidable tautology problem of (finite) nested graph constraints (cf. Cor. 9 in [HP09]), i.e., for a given (finite) set of (finite) constraints C over $\mathcal{M}$-initial object $\varnothing$ in $\mathbf{C}$ it is undecidable, whether C is satisfied by every graph in $\mathbf{C}$ $^{(*^1)}$. By Fact 6 in [HP09], for each constraint over the $\mathcal{M}$-initial object $\varnothing$, there is an equivalent constraint over $\varnothing$ in $\mathcal{M}$-normal form. Thus, $(*^1)$ does also hold for constraints C in $\mathcal{M}$-normal form $^{(*^2)}$. The results from [HP09] can be directly applied to category $\mathbf{C}$ by expressing labels by types and the finite label alphabet by a finite type graph TG. Furthermore, the results from [HP09] can be directly transferred from weak adhesive HLR categories with $\mathcal{E}$-$\mathcal{M}$-factorisation, $\mathcal{M}$-initial object and strict $\mathcal{M}$-decomposition to $\mathcal{M}$-adhesive categories having these properties (like $\mathbf{C}$), since, only basic HLR properties and general categorical properties are additionally used in the proofs. The reduction is given by a computable mapping from $C = (c_j)_{j\in J}$ and TG to TG' and the set of constraints $C' = \{\mathbf{true}\}$ together with a grammar $GG = (S,P)$, both typed over TG', with empty start graph $S = \varnothing$ and productions P that are defined as follows.



Type graph $TG' = TG +_{V_{TG}} \underline{TG}, V_{TG} = \{T_1, \ldots, T_n\}$ is TG extended by node $\underline{T}$ and edges $\underline{t}_i$ to $\underline{T}$ from each node $T_i \in TG$ with inclusion $i_t\colon TG \to TG'$. We define the obvious functor $F\colon \mathbf{C} \to \mathbf{C}'$ with $F((G,type_G)) := (G, i_t \circ type_G)$ and $F(m) := m$, for all $(G,type_G) \in Ob_{\mathbf{C}}$ and $m \in Mor_{\mathbf{C}}$ which is naturally extended to the mapping $\overline{F}(C)$ of sets C of constraints inductively defined by:

$$\begin{gathered}\overline{F}(\mathbf{true}) := \mathbf{true},\\ \overline{F}(\exists(a\colon P \to C, c)) := \exists(F(a)\colon F(P) \to F(C), \overline{F}(c)),\\ \overline{F}(\neg c) := \neg\overline{F}(c),\\ \overline{F}(\wedge_{j\in J}(c_j)) := \wedge_{j\in J}(\overline{F}(c_j)), \text{ and}\\ \overline{F}(\vee_{j\in J}(c_j)) := \vee_{j\in J}(\overline{F}(c_j))\end{gathered}$$

The mapping $\overline{F}(C)$ is given by $\overline{F}(C) := \cup_{c\in C}(\{\overline{F}(c)\})$. Obviously, for all $G \in \mathbf{C}$, all sets of constraints C and all $G' \in \mathbf{C}'$ with inclusions $i_G\colon F(G) \to G'$ and $type_{G'}(e) \in TG' \setminus i_t(TG), \forall e \in G' \setminus i_G(F(G))$, it holds that $G \models C \Leftrightarrow G' \models \overline{F}(C)$ $^{(*^3)}$. For each node $T \in TG$ there is a plain rule $\mathsf{a}_T \in P$ defined that creates a single node of type T. For each edge $t \in TG$ with source T_i and target T_j there is a rule $(\mathsf{a}_t, NAC) \in P$ defined that creates an edge of type t between existing source and target nodes $:\mathsf{T_i}$ and $:\mathsf{T_j}$ but only if there does not already exist a node of type $\underline{T}$. For each reflexive edge $t \in TG$ on node T there is a rule $(\mathsf{ar}_t, NAC) \in P$ defined that creates a reflexive edge of type t on node $:\mathsf{T}$ but only if there does not already exist a node of type $\underline{T}$. Additionally, there is rule $(\underline{T}, LA(\underline{T}, \overline{F}(C))) \in P$ defined that creates a single node of type $\underline{T}$ and for each edge $\underline{t}_i \in TG'$ from node T_i to $\underline{T}$ there is a rule $(\underline{t}_i, LA(\underline{t}_i, \overline{F}(C))) \in P$ defined that creates an edge of type $\underline{t}_i$ between nodes $:\mathsf{T_i}$ and $:\underline{\mathsf{T}}$. Note that the last rules are only applicable if the application conditions $LA(\underline{T}, \overline{F}(C))$ and $LA(\underline{t}_i, \overline{F}(C))$ are satisfied, respectively, which are obtained from conditions C as given in Lem. 3.1. We assume $\mathcal{M}$-matching for rule applications. If neglecting the application conditions for rules $\underline{T}$ and $\underline{t}_i$, then $\mathcal{L}(GG)$ contains all graphs in $\mathbf{C}'$ $^{(*^4)}$. It remains to show that C is satisfied by all graphs in $\mathbf{C}$ if and only if $\mathcal{L}(C') \subseteq \mathcal{L}(GG)$

holds in $\mathbf{C}'$. Then, assuming that the language inclusion problem is decidable would imply that the tautology problem is decidable leading to a contradiction by $(*^2)$.

"$\Rightarrow$" We need to show that the tautology of C in $\mathbf{C}$ implies that $\mathscr{L}(GG)$ contains all graphs in $\mathbf{C}'$, since, $\mathscr{L}(C')$ contains all graphs in $\mathbf{C}'$. By contradiction, assume that there is some $G \in \mathbf{C}'$ with $G \notin \mathscr{L}(GG)$. Thus by $(*^4)$, there is some $G' \in \mathbf{C}'$, production $(p\colon L \to R, LA(p, \overline{F}(C))) \in P$ with $p = \underline{T}$ or $p = \underline{t}_i$ and match $m\colon L \to G'$ such that $m \not\models LA(p, \overline{F}(C))$. By Lem. 3.1, it follows that $G' \not\models \overline{F}(C)$. By definition of $\mathbf{C}, \mathbf{C}'$ and F, there exists $G'' \in \mathbf{C}$ with inclusion $i_{G''}\colon F(G'') \to G'$ and $type_{G'}(e) \in TG' \setminus i_t(TG), \forall e \in G' \setminus i_{G''}(F(G''))$. By $(*^3)$, $G'' \not\models C$.

"$\Leftarrow$" We need to show that if $\mathscr{L}(GG)$ contains all graphs in $\mathbf{C}'$, then C is satisfied by every graph in $\mathbf{C}$. By contradiction, assume that there is some $G \in \mathbf{C}$ with $G \not\models C$.

Case $G = \varnothing$ We consider a graph $G' \in \mathbf{C}'$ which consists of a single $:\underline{\mathsf{T}}$ node only. By $(*^3)$, $F(G) \not\models \overline{F}(C)$. By Lem. 3.1, for all matches $m\colon L \to F(G)$ from production $\underline{T}\colon L \to R$ it is true that $m \not\models LA(\underline{T}, \overline{F}(C))$. Thus, $G' \notin \mathscr{L}(GG)$, since, $(\underline{T}, LA(\underline{T}, \overline{F}(C)))$ is the only rule in P that creates $:\underline{\mathsf{T}}$ nodes.

Case $G \neq \varnothing$ We consider a graph $G' \in \mathbf{C}'$ which is G on the $i_t(TG)$ part and where all nodes $:\mathsf{T}_\mathsf{i}$ typed over $i_t(TG)$ are connected to a single $:\underline{\mathsf{T}}$ node via $:\underline{\mathsf{t}}_\mathsf{i}$ edges. By the construction of P with NACs for rules a_t and ar_t it is guaranteed that rule $(\underline{t}_i, LA(\underline{t}_i, \overline{F}(C)))$ is the last rule applied via a match $m\colon L \to G''$ to obtain G' with inclusion $i_G\colon F(G) \to G''$ and $type_{G''}(e) \in TG' \setminus i_t(TG), \forall e \in G'' \setminus i_G(F(G))$. By $(*^3)$, $G'' \not\models \overline{F}(C)$ and furthermore by Lem. 3.1, $m \not\models LA(\underline{t}_i, \overline{F}(C))$. Thus, $G' \notin \mathscr{L}(GG)$.

Finally, GG is non-deleting by construction, P is finite, since, TG is finite and furthermore, all application conditions in P are finite and in $\mathscr{M}$-normal form by construction, since, $\overline{F}(C)$ preserves the finiteness of set C and the finiteness and $\mathscr{M}$-normal form of each $c_j \in C$. Furthermore, transformation LA leads to finite conditions by the finiteness of set $\overline{F}(C)$ for conjunction $\wedge_{j \in J}$ and the fact that transformation A preserves the finiteness of each $c_j \in \overline{F}(C)$ by the finiteness of the graphs in $\mathbf{C}'$ (cf. [HP09]). As we assume $\mathscr{M}$-matches for rule applications only, all application conditions over L that are not in $\mathscr{M}$-normal form can be transformed into equivalent application conditions over L in $\mathscr{M}$-normal form. The construction and proof is identical to Def. 5 and the proof of Fact 6 in [HP09]. C' is trivially finite and in $\mathscr{M}$-normal form.

□

A.4 Proof of Sec. 3.2 and Thm. 3.4

The termination of the verification of both conditions is considered separately implying further the termination of the verification of domain completeness up to the given upper bound.

1. For C-extension completeness we conclude as follows. The set of atoms that are typed over the type graph is finite up to isomorphism, since, the type graph is finite. Due to the fact that upper bound G_u is finite, the set of graphs $Graphs_{G_u} = \{G \mid G \in \mathscr{L}(TG), \exists G \to G_u \in \mathscr{M}\}$ is finite up to isomorphism as well as its computation terminates and therefore, also the set of graphs $\mathscr{L}(C)_{G_u} \subseteq Graphs_{G_u}$ is finite up to isomorphism. The check whether $G \in Graphs_{G_u}$ is also in $\mathscr{L}(C)_{G_u}$, i.e., $G \in \mathscr{L}(C)_{G_u}$, terminates in each case, since, G is finite by finite upper bound G_u, C is finite and moreover, each constraint $c \in$ C

is finite with a finite number of nestings. Therefore, the set of effective atoms $EAtoms(C)$ is finite up to isomorphism and furthermore, the identification of their effectiveness w.r.t. $\mathscr{L}(C)_{G_u}$ terminates, since, for each atom a we can iterate over finite $\mathscr{L}(C)_{G_u}$ in order to find $G \in \mathscr{L}(C)_{G_u}$ with inclusion $a \to G \in \mathscr{M}$. The inclusion checking terminates in each case, since, G is finite by finite upper bound G_u. Note that if a graph G is significant w.r.t. $\mathscr{L}(C)_{G_u}$ ($\exists G \to H \in \mathscr{M}, H \in \mathscr{L}(C)_{G_u}$), then G is finite, since, all graphs in $\mathscr{L}(C)_{G_u}$ are finite by finite upper bound G_u and so their $\mathscr{M}$-subobjects G. Concerning the extensions $Extensions(a,C)$ of each effective atom a, all graphs in $Extensions(a,C)$ are significant w.r.t. $\mathscr{L}(C)_{G_u}$ by construction Def. 3.5 and therefore also finite. Note that if a graph G is significant w.r.t. $\mathscr{L}(C)_{G_u}$ ($\exists i_1\colon G \to H \in \mathscr{M}, H \in \mathscr{L}(C)_{G_u}$ with $i_2\colon H \to G_u \in \mathscr{M}$ by definition of $\mathscr{L}(C)_{G_u}$), then $G \in Graphs_{G_u}$, since, $\exists i_2 \circ i_1\colon G \to G_u \in \mathscr{M}$ by $\mathscr{M}$-composition. Thus, $Graphs_{G_u}$ contains all graphs that are significant w.r.t. $\mathscr{L}(C)_{G_u}$. Therefore, the fact that $Extensions(a,C)$ only contains graphs that are significant w.r.t. $\mathscr{L}(C)_{G_u}$ and that set $Graphs_{G_u}$ is finite implies that each extension $E \in Extensions(a,C)$ is a finite set of graphs up to isomorphism. Note that the power set $\mathscr{P}(Graphs_{G_u})$ of the set $Graphs_{G_u}$ of significant graphs w.r.t. $\mathscr{L}(C)_{G_u}$ is finite by the finiteness of $Graphs_{G_u}$. Thus, as $Extensions(a,C)$ only contains significant graphs w.r.t. $\mathscr{L}(C)_{G_u}$, $\mathscr{P}(Graphs_{G_u})$ represents all possible extensions and therefore, $Extensions(a,C)$ is a finite set of extensions up to isomorphism. Finally, we can iterate over all effective atoms $a \in EAtoms(C)$, iterate over all extensions $E \in Extensions(a,C)$ and check for each graph $G \in E$ whether G can be created via grammar GG from the empty start graph. The check terminates for each G, since, the set of rules of GG is finite and each rule is non-deleting and non-trivial, i.e., the number of branches of direct transformation steps are finite at each step and the graph is extended by at least one element (node, edge or attribute) after each step and therefore, for each branch, we stop if the size of the graph is bigger than finite G. The verification of the satisfaction of application conditions of rules in each step terminates, since, the conditions are finite with a finite number of nestings. Therefore, the verification of C-extension completeness terminates.

Note that construction Def. 3.5 for constructing the $Extensions(a,C)$ of each effective atom a terminates as shown below and therefore, the construction can be directly used as algorithm to compute the extensions. At each extension via $extend(G_E, ac_P, m)$, the number of overlappings of G_E and C via P' in $extend(G_E, f, m)$ is finite up to isomorphism in each case, since, all conclusions C of all constraints are finite graphs. Analogously to the identification of the effectiveness of atoms w.r.t. $\mathscr{L}(C)_{G_u}$, the identification of the significance of graphs w.r.t. $\mathscr{L}(C)_{G_u}$ terminates. Furthermore, the constraints ac_P are finite with a finite number of nestings leading to a finite number of iterations via $extend(G_E, ac_P, m)$, i.e., each extension via $extend(G_E, ac_P, m)$ terminates. Moreover, the set of constraints C is finite, the premise of each constraint is a finite graph and therefore, the set of instances $\mathrm{Inst}(C)$ of C is finite up to isomorphism. Therefore, there are finitely many extensions via $extend(G_E, ac_P, m)$ for each graph of an extension, which is again a finite set of graphs as shown above. Therefore, we can step-wise compute extensions and stop a branch if it yields an extension that was already computed. The procedure terminates, since, there are only finitely many $Extensions(a,C)$ as shown above.

2. For C-conflict-freeness of marking rules we conclude as follows. Note that $Graphs_{G_u}$ is finite up to isomorphism and the set of all graphs that are significant w.r.t. $\mathscr{L}(C)_{G_u}$ as shown above. Therefore, we can iterate over all graphs $O \in Graphs_{G_u}$ in order to collect all critical pairs with conflict graph O that are significant w.r.t. $\mathscr{L}(C)_{G_u}$. The procedure

terminates, since, the set of rules of GG and therefore the set of all their combinations is finite, all graphs are finite for matching and all application conditions are finite with a finite number of nestings such that verifying their satisfaction terminates. Finally, we can iterate over the finite set of critical pairs that are significant w.r.t. $\mathscr{L}(C)_{G_u}$ and check if each pair is strictly confluent. The confluence analysis terminates in each case, since, we have finitely many non-trivial marking rules where each direct transformation step via marking rules updates at least one marking attribute from **F** to **T** until there is no applicable marking rule anymore or all marking attributes in finite O are set to **T**. □

A.5 Proof of Sec. 3.2 and Lem. 3.2

The main part of this result has been shown already in [HEO$^+$11] for the more complex setting of triple graphs based on the fully formalised concepts for translation attributes [HEGO10]. Hence, we reuse that equivalence result for triple sequences for the present case. First, note that each graph H can be extended to a triple graph $\mathscr{I}(H) = (H \xleftarrow{\varnothing} \varnothing \xrightarrow{\varnothing} \varnothing)$ with empty correspondence and target component via inclusion functor $\mathscr{I}$. Moreover, each rule (morphism) $(r\colon L \to R) \in P$ can be extended to a triple rule (morphism) $\mathscr{I}(r) = (r, \varnothing, \varnothing)\colon \mathscr{I}(L) \to \mathscr{I}(R)$ with empty correspondence and target component via inclusion functor $\mathscr{I}$. Given a set of rules P, then $\mathscr{I}(P) = \{\mathscr{I}(p) \mid p \in P\}$. Analogously, each match morphism can be extended and each marking rule $m(p) = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ can be extended to $\mathscr{I}(m(p)) = (\mathscr{I}(L) \xhookleftarrow{\mathscr{I}(l)} \mathscr{I}(K) \xhookrightarrow{\mathscr{I}(r)} \mathscr{I}(R))$. We derive the equivalence as follows:

$\exists$ transformation sequence $(1) = (G \oplus Att^{\mathbf{F}}_{G} \overset{*}{\Rightarrow} G \oplus Att^{\mathbf{T}}_{G_k} \oplus Att^{\mathbf{F}}_{G\setminus G_k})$ via marking rules $m(GG)$ with intermediate steps $G'_i \xRightarrow{(m'_i, m(p_i))} G'_{i+1}$

$\Leftrightarrow$ $\exists$ transformation sequence $\mathscr{I}(G) \oplus Att^{\mathbf{F}}_{\mathscr{I}(G)} \overset{*}{\Rightarrow} \mathscr{I}(G) \oplus Att^{\mathbf{T}}_{\mathscr{I}(G_k)} \oplus Att^{\mathbf{F}}_{\mathscr{I}(G)\setminus\mathscr{I}(G_k)}$ via marking rules $\mathscr{I}(m(GG))$

$\Leftrightarrow$ $\exists$ a transformation sequence $\mathscr{I}(\varnothing) \overset{*}{\Rightarrow} \mathscr{I}(G_k)$ via $\mathscr{I}(P)$ with injective embedding $\mathscr{I}(f)\colon \mathscr{I}(G_k) \to \mathscr{I}(G)$ by Sec. 2.3.3 and Fact 2.1

$\Leftrightarrow$ $\exists$ a transformation sequence $(2) = \varnothing \overset{*}{\Rightarrow} G_k$ via P with injective embedding $f\colon G_k \to G$ and with intermediate steps $G_i \xRightarrow{(m_i, p_i)} G_{i+1}$ (by restriction to the source component).

The correspondence of rules between the intermediate steps $G'_i \xRightarrow{(m'_i, m(p_i))} G'_{i+1}$ in sequence (1) and $G_i \xRightarrow{(m_i, p_i)} G_{i+1}$ in sequence (2) follows from Lemma 1 in [HEO$^+$11] and for the equivalence of triple steps and corresponding marking (consistency creating) steps and its application in the proof for Sec. 2.3.3 and Fact 2.1. □

A.6 Proof of Sec. 3.2 and Lem. 3.3

Let $GG = (S, P)$ be a graph grammar with a set P of non-deleting rules and $m(GG)$ be the set of derived marking rules of GG. Let G be a graph and let GG' be a graph grammar given by $GG' = (\varnothing, P \cup \{p_S = (\varnothing \to S)\})$. Then, there is the transformation step $\varnothing \xRightarrow{p_S} S$ via GG' and there is the transformation step $G \oplus Att^{\mathbf{F}}_{G} = G \oplus Att^{\mathbf{F}}_{S} \oplus Att^{\mathbf{F}}_{G\setminus S} \xRightarrow{m(p_S)} G \oplus Att^{\mathbf{T}}_{S} \oplus Att^{\mathbf{F}}_{G\setminus S}$ via $m(GG')$. Using Lem. 3.2, this implies that the following are equivalent.

- $\exists$ a transformation sequence $G \oplus Att^{\mathbf{F}}_{G} = G \oplus Att^{\mathbf{F}}_{S} \oplus Att^{\mathbf{F}}_{G\setminus S} \xRightarrow{m(p_S)} G \oplus Att^{\mathbf{T}}_{S} \oplus Att^{\mathbf{F}}_{G\setminus S} \overset{*}{\Rightarrow} G \oplus Att^{\mathbf{T}}_{G}$ via marking rules $m(GG)$.

- $\exists$ a transformation sequence $\varnothing \xRightarrow{p_S} S \overset{*}{\Rightarrow} G$ via P.

Since the first step in any of the two sequences is always possible, we derive the result of the lemma. □

A.7 Proof of Sec. 3.2 and Lem. 3.6

Let $G \in \mathscr{L}(C), a \in A \in Atoms(G)$ with embedding $e\colon a \to G \in \mathscr{M}$ and $E_n \in Extensions(a,C)$. We assume that E_n is derived by extension sequence $(E_j \xRightarrow{extend(a_E^j,c_j,m_j)} E_{j+1})_{j\in\{0,\dots,n-1\}}$ with $E_0 = \{a\}, \forall j \in \{0,\dots,n-1\}.a_E^j \in E_j, (f_j \in \mathscr{E},\mathscr{O},c_j) \in \mathrm{Inst}(C), c_j \equiv \vee_{i\in I}\exists(ac_i\colon P'_j \to C'_i, ac_{C'_i})$ and $m_j\colon P'_j \to a_E^j \in \mathscr{M}$ according to Def. 3.5. In the following we show that there is $a_E^n \in E_n$ with morphisms $e_1\colon a \to a_E^n \in \mathscr{M}$ and $e_2\colon a_E^n \to G \in \mathscr{M}$ with $e = e_2 \circ e_1$ by induction over the extension sequence. For each step j of the sequence we conclude as follows. For graph a_E^j with embedding $e_j\colon a_E^j \to G \in \mathscr{M}$, we derive extended graph $a_E^{j+1} \in E_{j+1}$ with embeddings $c_j\colon a_E^j \to a_E^{j+1} \in \mathscr{M}$ and $e'_j\colon a_E^{j+1} \to G \in \mathscr{M}$ and with $e_j = e'_j \circ c_j$ as follows.

$$\begin{tikzcd}
P \arrow[dr, "f_j\in\mathscr{E},\mathscr{O}"] \arrow[ddrr, bend right=40, "p_j\in\mathscr{O}"'] & & & & & \\
 & P'_j \arrow[r, dotted, "a_i"] \arrow[rrr, bend left=30, "ac_i\in\mathscr{M}"] \arrow[rrr, phantom, bend left=15, "(3)"] \arrow[dr, "m_j\in\mathscr{M}"] \arrow[ddr, "p'_j\in\mathscr{M}"'] \arrow[dr, phantom, bend left=20, "(2)"] & P''_j \arrow[rr, dotted, "ac'_i"] \arrow[d, dotted, "m'_j"] \arrow[drr, phantom, "(1)"] & & C'_i \arrow[dl, dotted, "b_j"] \arrow[ddll, bend left=40, "q'_j\in\mathscr{M}"] \arrow[dl, phantom, bend left=30, "(5)"] & \arrow[l, phantom, "\triangleleft"] ac_{C'_i} \\
 & & a_E^j \arrow[r, dotted, "c_j"] \arrow[d, hook, "e_j\in\mathscr{M}"'] \arrow[dr, phantom, "(4)"] & a_E^{j+1} \arrow[dl, dotted, "e'_j"] & & \\
 & & G & & &
\end{tikzcd}$$

Let match $p_j\colon P \to G = p'_j \circ f_j$ with $p'_j = e_j \circ m_j$. By Sec. 2.2.2 and Rem. 2.3 with $f_j \in \mathscr{O}$ and $m_j, e_j \in \mathscr{M}$, $p_j \in \mathscr{O}$ and $p'_j \in \mathscr{M}$ by $\mathscr{M}$-composition. $G \in \mathscr{L}(C)$ and Sec. 2.2.3 and Def. 2.19 imply that $p'_j \models c_j$. Therefore, $p'_j \models \exists(ac_i, ac_{C'_i})$ for some $i \in I$, i.e., there exists morphism $q'_j\colon C'_i \to G \in \mathscr{M}$ with $q'_j \circ ac_i = p'_j$. We construct pullback (ac'_i, m'_j) over morphisms (e_j, q'_j) with $e_j \circ m'_j = q'_j \circ ac'_i$ $^{(*^1)}$. By $e_j, q'_j \in \mathscr{M}$ and $\mathscr{M}$-morphisms are closed under pullbacks, it follows that $ac'_i, m'_j \in \mathscr{M}$. By the universal pullback property with $e_j \circ m_j = p' = q'_j \circ ac_i$ we obtain unique morphism $a_i\colon P'_j \to P''_j$ with commuting (2) and (3). Furthermore, $ac_i, ac'_i \in \mathscr{M}$ and $\mathscr{M}$-decomposition imply that $a_i \in \mathscr{M}$. We construct pushout (1) over morphisms (m'_j, ac'_i). Since $ac'_i, m'_j \in M$ and $\mathscr{M}$-morphisms are closed under pushouts, it follows that $b_j, c_j \in \mathscr{M}$. By $(*^1)$ and the universal pushout property we obtain unique morphism $e'_j\colon a_E^{j+1} \to G$ with commuting (4) and (5). By effective pushouts it follows that $e'_j \in \mathscr{M}$. Thus, there is an embedding e'_j from extended graph a_E^{j+1} to G, i.e., by assumption $G \in \mathscr{L}(C)$ it follows that a_E^{j+1} is significant w.r.t. $\mathscr{L}(C)$ implying further that a_E^{j+1} is also not C-inconsistent by Rem. 3.2. We can conclude recursively for nestings $ac_{C'_i}$ of condition c_j. By construction Def. 3.5 it follows that $(b_j, a_E^{j+1}) \in extend(a_E^j, c_j, m_j)$. Therefore, for an extension step $E_j \xRightarrow{extend(a_E^j,c_j,m_j)} E_{j+1}$ with graph $a_E^j \in E_j$ and embedding $e_j\colon a_E^j \to G \in \mathscr{M}$, it holds that there exists a graph $a_E^{j+1} \in E_{j+1}$ with embeddings $c_j\colon a_E^j \to a_E^{j+1} \in \mathscr{M}$ and $e'_j\colon a_E^{j+1} \to G \in \mathscr{M}$ and with $e_j = e'_j \circ c_j$. By induction over the steps s_j of extension sequence $(s_j\colon E_j \xRightarrow{extend(a_E^j,c_j,\_)} E_{j+1})_{j\in\{0,\dots,n-1\}}$, we obtain the desired result, that extension E_n contains a graph a_E^n with morphisms $e_1\colon a \to a_E^n \in \mathscr{M}$ and $e_2\colon a_E^n \to G \in \mathscr{M}$ and with $e = e_2 \circ e_1$. **Basis.** For $n = 1$, we have extension step $s_0\colon E_0 = \{a\} \xRightarrow{extend(a,c_0,\_)} E_1$ with embedding $e\colon a \to G \in \mathscr{M}$ for $a \in A \in Atoms(G)$. By using the result from above, there exists graph $a_E^1 \in E_1$ with embeddings $e_1\colon a \to a_E^1 \in \mathscr{M}$ and $e_2\colon a_E^1 \to G \in \mathscr{M}$ and with $e = e_2 \circ e_1$.

Hypothesis. There is an $n \in \mathbb{N}$ and extension sequence $(s_j\colon E_j \xRightarrow{extend(a_E^j,c_j,\_)} E_{j+1})_{j\in\{0,\dots,n-1\}}$ with $E_0 = \{a\}, e\colon a \to G \in \mathscr{M}, \forall j \in \{0,\dots,n-1\}.a_E^j \in E_j, (\_,c_j) \in \mathrm{Inst}(C)$ such that there exists a graph $a_E^n \in E_n$ with embeddings $e_1\colon a \to a_E^n \in \mathscr{M}$ and $e_2\colon a_E^n \to G \in \mathscr{M}$ and with $e = e_2 \circ e_1$.

Step. For $n+1$, we focus on the last step $s_n\colon E_n \xRightarrow{extend(a_E^n,c_n,\_)} E_{n+1}$ of the extension sequence.

Case (a) Let $a_E^n \in E_n$ be a graph with embeddings $e'_1\colon a \to a_E^n \in \mathscr{M}$ and $e'_2\colon a_E^n \to G \in \mathscr{M}$ and with $e = e'_2 \circ e'_1$, then we can apply the result from above for step s_n and obtain graph $a_E^{n+1} \in E_{n+1}$ with embeddings $a \xrightarrow{e'_1} a_E^n \xrightarrow{e_1} a_E^{n+1} \in \mathscr{M}$ (by $\mathscr{M}$-composition) and $e_2\colon a_E^{n+1} \to G \in \mathscr{M}$ and with $e'_2 = e_2 \circ e_1$, i.e., $e'_2 \circ e'_1 = e_2 \circ e_1 \circ e'_1 \Rightarrow e = e_2 \circ e_1 \circ e'_1$.

Case (b) Let $a_E^n \in E_n$ be a graph without the embeddings from Case (a), then there exists another graph $a'^n_E \in E_n$ with embeddings $e_1\colon a \to a'^n_E \in \mathscr{M}$ and $e_2\colon a'^n_E \to G \in \mathscr{M}$ and with $e = e_2 \circ e_1$ by induction hypothesis. By Def. 3.5, it holds that $a'^n_E \in E_{n+1}$, since, graph a'^n_E is not extended by step s_n. □

A.8 Proof of Sec. 3.2 and Lem. 3.7

Let $A = (a_i)_{i\in\{1,\dots,n\}} \in Atoms(G)$ be the atoms of graph $G \in \mathscr{L}(C)$ which generally satisfies the set of constraints C with induced morphisms $(e_i\colon a_i \to G \in \mathscr{M})_{i\in\{1,\dots,n\}}$ by Lem. 3.5. Let $f_E \in SELECT_E(A,C)$ be a function that selects a C-extension $E_i \in Extensions(a_i,C)$ for each atom $a_i \in A$. By Lem. 3.6, it follows that each C-extension E_i contains a graph $a_E^i \in E_i$ with morphisms $e_1^i\colon a_i \to a_E^i \in \mathscr{M}$ and $e_2^i\colon a_E^i \to G \in \mathscr{M}$ and with $e_i = e_2^i \circ e_1^i$. Therefore, we can define a function $f_{a_E} \in SELECT_{a_E}(A,C,f_E)$ such that $\forall a_i \in A.f_{a_E}(a_i) = a_E^i$ with morphisms $e_1^i\colon a_i \to f_{a_E}(a_i) \in \mathscr{M}$ and $e_2^i\colon f_{a_E}(a_i) \to G \in \mathscr{M}$ and with $e_i = e_2^i \circ e_1^i$. It remains to show that there exists the corresponding sequence of pushouts $(PO_k^E +_{G_k^E} f_{a_E}(a_{k+1}) = PO_{k+1}^E)_{k\in\{1,\dots,n-1\}}$ with pushout objects PO_{k+1}^E, $PO_1^E = f_{a_E}(a_1)$ and $PO_n^E = G$. By induction over the list of atoms $(a_i)_{i\in\{1,\dots,n\}}$ we conclude as follows. For $n = 1$ the assumption holds trivially. **Basis.** For $n = 2$ we focus on the diagram below (right). By Lem. 3.5, there exists pushout $(f'_{n-1} \in \mathscr{M}, e_n \in \mathscr{M})$ over $(g_{n-1} \in \mathscr{M}, f_{n-1} \in \mathscr{M})$ with $PO_{n-1} = a_{n-1}, PO_n = G$ and $f'_{n-1} = e_{n-1}$. Furthermore, let $PO_{n-1}^E = f_{a_E}(a_{n-1}), \overline{e}_1^{n-1} = e_1^{n-1} \in \mathscr{M}$ and $\overline{e}_2^{n-1} = e_2^{n-1} \in \mathscr{M}$ such that (3) commutes. Analogously, commuting (4) is given with $e_1^n, e_2^n \in \mathscr{M}$. We can construct pullback (1) by $\overline{e}_2^{n-1} \in \mathscr{M}$ with $G_{n-1}^E \to f_{a_E}(a_n), G_{n-1}^E \to PO_{n-1}^E \in \mathscr{M}$ by $\mathscr{M}$-morphisms are closed under pullbacks. Analogously, we can construct pullback (2) with $G'_{n-1} \to a_n \in \mathscr{M}$. By pullback composition, also $(1)+(2)$ is a pullback with induced unique morphism $G_{n-1} \to G'_{n-1}$ and commuting (5) and (6) by universal pullback property. By $\mathscr{M}$-pushout-pullback decomposition with the outer pushout diagram, commuting $(3), (4)$ and (6), pullback $(1)+(2)$, and $\overline{e}_2^{n-1}, g_{n-1} \in \mathscr{M}$, it follows that $(1)+(2)$ is a pushout. Again by $\mathscr{M}$-pushout-pullback decomposition with pushout $(1)+(2)$, pullback (1), and $G'_{n-1} \to a_n, e_2^n \in \mathscr{M}$, it follows that (1) is the requested pushout with $PO_n^E = PO_n = G$.

For $n = 3$ we focus on the diagram below (left) with $k = 1$. By Lem. 3.5, there exists pushout (7) with $PO_1 = a_1$ and induced morphisms $e_1\colon a_1 \xrightarrow{f'_1} PO_2 \xrightarrow{f'_2 \in \mathscr{M}} G \in \mathscr{M}, e_2\colon a_2 \xrightarrow{g'_1} PO_2 \xrightarrow{f'_2 \in \mathscr{M}} G \in \mathscr{M}$. Furthermore, let $PO_1^E = f_{a_E}(a_1)$ with $\overline{e}_1^1 = e_1^1 \in \mathscr{M}$ and $\overline{e}_2^1 = e_2^1 \in \mathscr{M}$ implying further that $(7a)$ and $(7b)$ commute. We construct effective pushout (8) over $(\overline{e}_2^1, e_2^2)$ with all morphisms in $\mathscr{M}$ by $\mathscr{M}$ is closed under pushouts and pullbacks, pullback $(8)+(8a)+(8b)$ and induced morphism $\overline{e}_2^2 \in \mathscr{M}$ such that $(8a)$ and $(8b)$ commute. By the universal pushout property of (7) with $f'_1 \circ g_1 \overset{(7)}{=} g'_1 \circ f_1 \Rightarrow f'_2 \circ f'_1 \circ g_1 = f'_2 \circ g'_1 \circ f_1$ there is a unique morphism $PO_2 \to G = f'_2 \in \mathscr{M}$ such that $(7a)+(8a)+(8c)$ and $(7b)+(8b)+(8d)$ commute. Again by the universal pushout property of (7) with $f'_2 \circ f'_1 \circ g_1 = f'_2 \circ g'_1 \circ f_1 \Leftrightarrow e_1 \circ g_1 = e_2 \circ f_1 \overset{(7a),(7b)}{\Rightarrow} \overline{e}^1_2 \circ \overline{e}^1_1 \circ g_1 = e^2_2 \circ e^2_1 \circ f_1 \overset{(8a),(8b)}{\Rightarrow} \overline{e}^2_2 \circ a_1 \circ \overline{e}^1_1 \circ g_1 = \overline{e}^2_2 \circ b_1 \circ \overline{e}^2_1 \circ f_1 \overset{\overline{e}^2_2 \text{ is mono}}{\Rightarrow} a_1 \circ \overline{e}^1_1 \circ g_1 = b_1 \circ \overline{e}^2_1 \circ f_1$, there is morphism $\overline{e}^2_1$ such that $(8c)$ and $(8d)$ commute. Thus, for $\overline{e}^2_2 \circ \overline{e}^2_1$, $(7a)+(8a)+(8c)$ and $(7b)+(8b)+(8d)$ commute, i.e., the uniqueness of morphism $f'_2 \in \mathscr{M}$ implies that $f'_2 = \overline{e}^2_2 \circ \overline{e}^2_1$. By $\mathscr{M}$-decomposition with $\overline{e}^2_2 \in \mathscr{M}$ it follows that $\overline{e}^2_1 \in \mathscr{M}$. By Lem. 3.5, there exists pushout $(f'_2 \in \mathscr{M}, e_3 \in \mathscr{M}) = (1) + \ldots + (6)$ over $(g_{n-1} \in \mathscr{M}, f_{n-1} \in \mathscr{M})$ in the diagram below (right) with $PO_3 = G$, $\overline{e}^2_1, \overline{e}^2_2, f'_2 \in \mathscr{M}$ from above such that (3) commutes. Analogously to base case with $n = 2$, we obtain puhsout (1) leading to the requested sequence of pushouts (8) and (1) with pushout object $PO^E_3 = G$.

$$\begin{tikzcd}
 & & G & & \\
 & (8a) & PO^E_{k+1} \arrow[u, "\overline{e}^{k+1}_2"] & (8b) & \\
PO^E_k \arrow[uurr, bend left, "\overline{e}^k_2"] \arrow[urr, dotted] & & (8) & & f_{a_E}(a_{k+1}) \arrow[uull, bend right, "e^{k+1}_2"'] \arrow[ull, dotted] \\
(7a) & & G^E_k \arrow[ull, dotted] \arrow[urr, dotted] & & (7b) \\
 & (8c) & PO_{k+1} \arrow[uuu, dotted, "\overline{e}^{k+1}_1"'] & (8d) & \\
PO_k \arrow[uuu, "\overline{e}^k_1"] \arrow[urr, "f'_k"] \arrow[uuuuurr, bend left, "f'_{n-1}\circ\ldots\circ f'_k"] & & (7) & & a_{k+1} \arrow[uuu, "e^{k+1}_1"'] \arrow[ull, "g'_k"'] \arrow[uuuuull, bend right, "f'_{n-1}\circ\ldots\circ f'_{k+1}\circ g'_k"'] \\
 & & G_k \arrow[ull, "g_k"] \arrow[urr, "f_k"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & (3) & PO^E_n = PO_n = G & (4) & \\
 & PO^E_{n-1} \arrow[ur, "\overline{e}^{n-1}_2"] & (1) & f_{a_E}(a_n) \arrow[ul, "e^n_2"'] & \\
PO_{n-1} \arrow[ur, "\overline{e}^{n-1}_1"] \arrow[uurr, bend left, "f'_{n-1}"] & & G^E_{n-1} \arrow[ul, dotted] \arrow[ur, dotted] & (2) & a_n \arrow[ul, "e^n_1"'] \arrow[uull, bend right, "e_n"'] \\
 & (5) & & G'_{n-1} \arrow[ul, dotted] \arrow[ur, dotted] & \\
 & & G_{n-1} \arrow[ur, dotted] \arrow[uull, bend left, "g_{n-1}"] \arrow[uurr, bend right, "f_{n-1}"'] & (6) &
\end{tikzcd}$$

Hypothesis. There is $n \in \mathbb{N}$ such that for given pushout (7) with $k = 1$, for given induced morphisms $PO_1 \xrightarrow{f'_1} PO_2 \xrightarrow{f'_{n-1}\circ\ldots\circ f'_2 \in \mathscr{M}} G \in \mathscr{M}$ and $e_2\colon a_2 \xrightarrow{g'_1} PO_2 \xrightarrow{f'_{n-1}\circ\ldots\circ f'_2 \in \mathscr{M}} G \in \mathscr{M}$ according to Lem. 3.5, and for given PO^E_1 with morphisms $\overline{e}^1_1, \overline{e}^1_2 \in \mathscr{M}$ such that $(7a)$ commutes, the following holds: There exists a sequence of pushouts $(PO^E_k +_{G^E_k} f_{a_E}(a_{k+1}) = PO^E_{k+1})_{k \in \{1,\ldots,n-1\}}$ with pushout objects PO^E_{k+1}, all morphisms being in $\mathscr{M}$ and $PO^E_n = G$. **Step.** For $n+1$ and the first pushout in the sequence $(k = 1)$, we can conclude analogously to the base case for $n = 3$ with induced morphisms $e_1\colon a_1 \xrightarrow{f'_1} PO_2 \xrightarrow{f'_n\circ\ldots\circ f'_2 \in \mathscr{M}} G \in \mathscr{M}$ and $e_2\colon a_2 \xrightarrow{g'_1} PO_2 \xrightarrow{f'_n\circ\ldots\circ f'_2 \in \mathscr{M}} G \in \mathscr{M}$ and obtain pushout (8) and morphisms $\overline{e}^2_1\colon PO_2 \to PO^E_2, \overline{e}^2_2\colon PO^E_2 \to G \in \mathscr{M}$ such that $f'_n \circ \ldots \circ f'_2 = \overline{e}^2_2 \circ \overline{e}^2_1$. By Lem. 3.5, there exists pushout (7) with $k = 2$ and induced morphisms $PO_2 \xrightarrow{f'_2} PO_3 \xrightarrow{f'_n\circ\ldots\circ f'_3 \in \mathscr{M}} G \in \mathscr{M}$ and $e_3\colon a_3 \xrightarrow{g'_2} PO_3 \xrightarrow{f'_n\circ\ldots\circ f'_3 \in \mathscr{M}} G \in \mathscr{M}$. Thus, by induction hypothesis there exists the remaining sequence of pushouts $(PO^E_k +_{G^E_k} f_{a_E}(a_{k+1}) = PO^E_{k+1})_{k \in \{2,\ldots,n\}}$ with pushouts objects PO^E_{k+1}, all morphisms being in $\mathscr{M}$ and $PO^E_{n+1} = G$. □

A.9 Proof of Sec. 3.2 and Lem. 3.12

The proof is based on the structure and satisfaction of conditions (cf. Def. 2.12) as well as the construction of merge (cf. Def. 2.16). By the satisfaction of conditions, the equivalence $\mathrm{Merge}(b_2, \mathrm{Merge}(b_1, ac_P)) \equiv \mathrm{Merge}(b_2 \circ b_1, ac_P)$ means that for all morphisms $p\colon P'' \to G \in \mathscr{O}$ to some G it is true that $p \models \mathrm{Merge}(b_2, Merge(b_1, ac_P))$ if and only if $p \models \mathrm{Merge}(b_2 \circ b_1, ac_P)$.

We prove this equivalence by induction over the number i of nestings of ac_P where the number of nestings is given by the number of successive morphisms in ac_P. For $i = 0$ (induction base), the equivalence is shown by induction over the structure of condition ac_P.

(base case $ac_P = \textbf{true}$**)** By the merge construction, it follows that $ac_P = \textbf{true} = Merge(b_1, ac_P)$ $= \text{Merge}(b_2, Merge(b_1, ac_P)) = \text{Merge}(b_2 \circ b_1, ac_P)$. Therefore, the equivalence holds.

(base case $ac_P = \exists(a\colon P \hookrightarrow C, \textbf{true})$**)**

"$\Rightarrow$" By the satisfaction of conditions and the merge construction, $p \models \text{Merge}(b_2, Merge(b_1, ac_P))$ implies that there exists diagrams (1) and (2) such that (1) commutes $^{(*^1)}$ and (2) commutes $^{(*^2)}$ with $b'_1, b'_2 \in \mathscr{O}, a', a'' \in \mathscr{M}$ and (b'_1, a') as well as (b'_2, a'') being jointly epimorphic.

$$\begin{tikzcd}
 & ac_C \arrow[d, phantom, "\bigtriangledown"] & ac_P \arrow[d, phantom, "\bigtriangledown"] & \\
 & C \arrow[d, "b'_1\in\mathscr{O}"'] \arrow[dr, phantom, "(1)"] & P \arrow[l, hook', "a\in\mathscr{M}"'] \arrow[d, "b_1"] & \\
ac_{C'} \arrow[r, phantom, "\rhd"] & C' \arrow[d, "b'_2\in\mathscr{O}"'] \arrow[dr, phantom, "(2)"] & P' \arrow[l, hook', "a'\in\mathscr{M}"'] \arrow[d, "b_2"] \arrow[r, phantom, "\lhd"] & ac_{P'} \\
ac_{C''} \arrow[r, phantom, "\rhd"] & C'' \arrow[rr, hook, bend right=40, "q\in\mathscr{M}"'] \arrow[r, phantom, bend right, "(=)"'] & P'' \arrow[l, hook', "a''\in\mathscr{M}"'] \arrow[r, "p"] & G \\
 & & ac_{P''} \arrow[u, phantom, "\bigtriangleup"] &
\end{tikzcd}$$

$$ac_{C'} = \text{Merge}(b'_1, ac_C), ac_{C''} = \text{Merge}(b'_2, ac_{C'})$$
$$ac_{P'} = \text{Merge}(b_1, ac_P), ac_{P''} = \text{Merge}(b_2, ac_{P'})$$

Furthermore, $p \models \exists(a'', ac_{C''})$ with $ac_{C''} = \text{Merge}(b'_2, ac_{C'})$. This implies that there exists a morphism $q\colon C'' \to G \in \mathscr{M}$ with $q \circ a'' = p$ and $q \models ac_{C''}$ $^{(*^3)}$. By the composition of $\mathscr{O}$-morphisms (cf. Lem. 3.8, item 1), it follows that $b'_2 \circ b'_1 \in \mathscr{O}$. By the composition of jointly epimorphic pairs of morphisms (cf. Lem. 3.9, item 1) and $(*^2)$, it follows that $(b'_2 \circ b'_1, a'')$ is a pair of jointly epimorphic morphisms. Diagram $(1)+(2)$ commutes, since, $(*^1)$ implies $b'_1 \circ a = a' \circ b_1 \implies b'_2 \circ b'_1 \circ a = b'_2 \circ a' \circ b_1 \overset{(*^2)}{\implies} b'_2 \circ b'_1 \circ a = a'' \circ b_2 \circ b_1$. Therefore, $ac_i = \exists(a'', \text{Merge}(b'_2 \circ b'_1, ac_C))$ is a condition of $\text{Merge}(b_2 \circ b_1, ac_P) = \vee_i ac_i$. Furthermore, by the merge construction, $ac_C = \textbf{true}$ implies $\text{Merge}(b'_2, ac_{C'}) = \text{Merge}(b'_2 \circ b'_1, ac_C) = \textbf{true}$. Therefore, by $(*^3)$, $q \models \text{Merge}(b'_2 \circ b'_1, ac_C)$. Thus, $p \models ac_i$ and furthermore, $p \models \text{Merge}(b_2 \circ b_1, ac_P)$.

"$\Leftarrow$" By the satisfaction of conditions and the merge construction, $p \models \text{Merge}(b_2 \circ b_1, ac_P)$ implies that there exists an outer diagram with morphisms $(a, b', b_2 \circ b_1$ and $a')$ such that the diagram commutes $^{(*^1_a)}$ with $b' \in \mathscr{O}, a' \in \mathscr{M}, a \in \mathscr{M}$ (we assume conditions in $\mathscr{M}$-normal form) and (a', b') being jointly epimorphic $^{(*^1_b)}$. Furthermore, there exists a morphism $q\colon C'' \to G \in \mathscr{M}$ with $q \circ a' = p$ and $q \models ac_{C''}$ $^{(*^1_c)}$. For the $\mathscr{M}$-adhesive category $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ (cf. [GLEO12] Thm. 7), we can use the properties from Def. 4.9 and Fact 2.20 in [EEPT06] ($\mathscr{M}$-morphisms are closed under composition, decomposition, pushouts and pullbacks; pushouts and pullbacks exist along $\mathscr{M}$-morphisms; pushouts are closed under composition).

Since $a' \in \mathscr{M}$ and pullbacks exist along $\mathscr{M}$-morphisms in $\mathbf{AGraphs}_{ATGI}$, we can construct the pullback $(\underline{C}, \underline{c}, b'')$ over morphisms (a', b'). By $\mathscr{M}$ is closed under pullbacks, from $a' \in \mathscr{M}$ it follows that $\underline{c} \in \mathscr{M}$. By $(*^1_a)$ and the universal pullback property, there exists a morphism $\underline{a}\colon P \to \underline{C}$ with $\underline{c} \circ \underline{a} = a$ $^{(*^2)}$. By $\mathscr{M}$-decomposition an $(*^2)$, it follows that $\underline{a} \in \mathscr{M}$. Since $\underline{a} \in \mathscr{M}$ and pushouts exist along $\mathscr{M}$-morphisms in $\mathbf{AGraphs}_{ATGI}$, we can construct pushout (1) over morphisms $(\underline{a}, b_1)$. Analogously, since $\underline{c} \in \mathscr{M}$, we can construct pushout (2) over morphisms $(\underline{c}, b_1')$. By pushout composition, it follows that (1)+(2) is a pushout with $\underline{c} \circ \underline{a} \in \mathscr{M}$, since, $\mathscr{M}$ is closed under composition. Since $\mathscr{M}$ is closed under pushouts, it follows that $\underline{c'} \circ \underline{a'} \in \mathscr{M}$ $^{(*^3)}$. By $(*^1_a)$,$(*^2)$ and the universal pushout property, it follows that there exists a morphism $c' \colon C' \to C''$ with $c' \circ b_1'' = b'$ $^{(*^4)}$ and $c' \circ \underline{c'} \circ \underline{a'} = a' \circ b_2$ $^{(*^5)}$. By $b' \in \mathscr{O}$, $(*^4)$ and Lem. 3.8, item 2(a), it follows that $b_1'' \in \mathscr{O}$ $^{(*^6)}$. By (1)+(2) being a pushout, it follows that (1)+(2) commutes and $(b_1'', \underline{c'} \circ \underline{a'})$ are jointly epimorphic $^{(*^7)}$ (cf. Fact 2.17, item 2 in [EEPT06]). By $(*^2)$, $(*^3)$, $(*^6)$, $(*^7)$ and the merge construction, it follows that (1)+(2)+(4) is a diagram of Merge(b_1, ac_P).

$$\begin{tikzcd}
ac_C \arrow[d, phantom, "\bigtriangledown"] & & (4) & ac_P \arrow[d, phantom, "\bigtriangledown"] & \\
C \arrow[dr, "b_1''"] \arrow[dddd, "b'\in\mathscr{O}"'] \arrow[drr, phantom, "(2)"] & & \underline{C} \arrow[ll, "\underline{c}"'] \arrow[d, "b_1'"] \arrow[dddr, "b''"] \arrow[dr, phantom, "(1)"] & P \arrow[l, "\underline{a}"'] \arrow[lll, hook', bend right, "a\in\mathscr{M}"'] \arrow[d, "b_1\in\mathscr{E}"] & \\
ac_{C'} \arrow[r, phantom, "\triangleright"] & C' \arrow[d, "b_2'"] \arrow[ddl, "c'"'] & \underline{C'} \arrow[l, "\underline{c'}"'] \arrow[ddr, phantom, "(3)"] & P' \arrow[l, "\underline{a'}"'] \arrow[dd, "b_2"] \arrow[r, phantom, "\triangleleft"] & ac_{P'} \\
& \underline{C''} \arrow[dl, "\underline{c''}"'] \arrow[r, phantom, "\triangleleft"] \arrow[dr, phantom, "(5)"] & ac_{\underline{C''}} & & \\
C'' \arrow[rrrr, hook, bend right, "q\in\mathscr{M}"'] \arrow[d, phantom, "\bigtriangleup"] & & & P'' \arrow[ull, "a''"'] \arrow[lll, hook', "a'\in\mathscr{M}"'] \arrow[r, "p"] \arrow[d, phantom, "\bigtriangleup"] & G \\
ac_{C''} & & (=) & ac_{P''} &
\end{tikzcd}$$

$$ac_{C'} = \text{Merge}(b_1'', ac_C), ac_{C''} = \text{Merge}(b', ac_C), ac_{\underline{C''}} = \text{Merge}(b_2', ac_{C'})$$
$$ac_{P'} = \text{Merge}(b_1, ac_P), ac_{P''} = \text{Merge}(b_2 \circ b_1, ac_P)$$

It remains to show that diagram (3)+(5) is a diagram of Merge$(b_2, \text{Merge}(b_1, ac_P))$ and that $p \models \text{Merge}(b_2, \text{Merge}(b_1, ac_P))$.

Since $\underline{c'} \circ \underline{a'} \in \mathscr{M}$ (cf. $(*^3)$) and pushouts exist along $\mathscr{M}$-morphisms in $\mathbf{AGraphs}_{ATGI}$, we can construct pushout (3) over morphisms $(\underline{c'} \circ \underline{a'}, b_2)$. By $(*^5)$ and the universal pushout property, there exists a morphism $\underline{c''} \colon \underline{C''} \to C''$ with $\underline{c''} \circ b_2' = c'$ $^{(*^8_a)}$ and $\underline{c''} \circ a'' = a'$ $^{(*^8_b)}$. Furthermore, (3) being a pushout implies that (3) commutes and it follows that $b_2' \circ \underline{c'} \circ \underline{a'} = a'' \circ b_2 \implies \underline{c''} \circ b_2' \circ \underline{c'} \circ \underline{a'} = \underline{c''} \circ a'' \circ b_2 \overset{(*^8_b)}{=} a' \circ b_2$, i.e., diagram (3)+(5) commutes $^{(*^9)}$. Since, morphisms (a', b') are jointly epimorphic (cf. $(*^1_b)$) and $(a', b') \overset{(*^4)}{=} (a', c' \circ b_1'') \overset{(*^8_a)}{=} (a', \underline{c''} \circ b_2' \circ b_1'')$, it follows that $(a', \underline{c''} \circ b_2')$ are jointly epimorphic morphisms $^{(*^{10})}$ (cf. Lem. 3.9, item 2). It remains to show that $\underline{c''} \circ b_2' \in \mathscr{O}$ for showing that diagram (3)+(5) is a diagram of Merge$(b_2, \text{Merge}(b_1, ac_P))$. Since, $a \in \mathscr{M}$ and $b' \in \mathscr{O}$ (cf. $(*^1_b)$) it follows that $b' \circ a \in \mathscr{O}$. Furthermore, $b' \circ a = a' \circ b_2 \circ b_1$ (cf. $(*^1_a)$) implies that $b_1 \in \mathscr{O}$ by Lem. 3.8, item 2(a). Since, $b_1 \in \mathscr{O}$ and $b_1 \in \mathscr{E}$ by assumption it follows that $b_{1,S}$ is an isomorphism (cf. Lem. 3.10). Thus, $b''_{1,S}$ is an isomorphism, since, pushouts preserve isomorphisms and so do pushouts (1) and (2). By Lem. 3.8, item 2(b), $b''_{1,S}$ being an isomorphism, $b' \in \mathscr{O}$ (cf. $(*^1_b)$) and $b' \overset{(*^4)}{=} c' \circ b''_1 \overset{(*^8_a)}{=} \underline{c''} \circ b'_2 \circ b''_1$ implies that $\underline{c''} \circ b'_2 \in \mathscr{O}$ $(*^{11})$. Therefore, by $a' \in \mathscr{M}$ (cf. $(*^1_b)$), $(*^9)$, $(*^{10})$, $(*^{11})$ and the merge construction it follows that diagram (3)+(5) is a diagram of $\text{Merge}(b_2, \text{Merge}(b_1, ac_P))$ with $ac_{C''} = \text{Merge}(\underline{c''} \circ b'_2, \text{Merge}(b''_1, ac_C)) = \textbf{true}$ by assumption $ac_C = \textbf{true}$ and the merge construction.

Therefore, $ac_i = \exists(a', ac_{C''}) = \exists(a', \textbf{true})$ is a nested-condition of $\text{Merge}(b_2, \text{Merge}(b_1, ac_P)) = \vee_i ac_i$. Thus, $q \models ac_{C''}$ and furthermore by $(*^1_c)$ it follows that $p \models ac_i$ and thus, $p \models \text{Merge}(b_2, \text{Merge}(b_1, ac_P))$.

(induction step $ac_P = \vee_{i \in I} ac_{P,i}$**)** By showing the equivalence for the base cases, it follows that $\text{Merge}(b_2, \text{Merge}(b_1, ac)) \equiv \text{Merge}(b_2 \circ b_1, ac)$ for some ac (hypothesis of the structural induction). We assume the hypothesis for all $ac_{P,i}$. By the merge construction, it follows that $\text{Merge}(b_2, \text{Merge}(b_1, ac_P)) = \text{Merge}(b_2, \vee_{i \in I} \text{Merge}(b_1, ac_{P,i})) = \vee_{i \in I} \text{Merge}(b_2, \text{Merge}(b_1, ac_{P,i}))$ and $\text{Merge}(b_2 \circ b_1, ac_P) = \vee_{i \in I} \text{Merge}(b_2 \circ b_1, ac_{P,i})$.

"⇒" $p \models \text{Merge}(b_2, \text{Merge}(b_1, ac_P))$ implies that there is an $i \in I$ with $p \models \text{Merge}(b_2, \text{Merge}(b_1, ac_{P,i}))$ by the satisfaction of conditions. By induction hypothesis, it follows that $p \models \text{Merge}(b_2 \circ b_1, ac_{P,i})$ and therefore, $p \models \vee_{i \in I} \text{Merge}(b_2 \circ b_1, ac_{P,i}) = \text{Merge}(b_2 \circ b_1, ac_P)$.

"⇐" $p \models \text{Merge}(b_2 \circ b_1, ac_P)$ implies that there is an $i \in I$ with $p \models \text{Merge}(b_2 \circ b_1, ac_{P,i})$ by the satisfaction of conditions. By induction hypothesis, it follows that $p \models \text{Merge}(b_2, \text{Merge}(b_1, ac_{P,i}))$ and therefore, $p \models \vee_{i \in I} \text{Merge}(b_2, \text{Merge}(b_1, ac_{P,i})) = \text{Merge}(b_2, \text{Merge}(b_1, ac_P))$.

(induction step $ac_P = \wedge_{i \in I} ac_{P,i}$**)** Analogously to the previous step, by the merge construction and induction hypothesis it follows that $\text{Merge}(b_2, \text{Merge}(b_1, ac_P)) = \wedge_{i \in I} \text{Merge}(b_2, \text{Merge}(b_1, ac_{P,i})) \equiv \wedge_{i \in I} \text{Merge}(b_2 \circ b_1, ac_{P,i}) = \text{Merge}(b_2 \circ b_1, ac_P)$.

(induction step $ac_P = \neg ac'_P$**)** By induction hypothesis, we assume that $\text{Merge}(b_2, \text{Merge}(b_1, ac'_P)) \equiv \text{Merge}(b_2 \circ b_1, ac'_P)$. By merge construction, $\text{Merge}(b_2, \text{Merge}(b_1, ac_P)) = \text{Merge}(b_2, \neg\text{Merge}(b_1, ac'_P)) = \neg\text{Merge}(b_2, \text{Merge}(b_1, ac'_P))$ and $\text{Merge}(b_2 \circ b_1, ac_P) = \neg\text{Merge}(b_2 \circ b_1, ac'_P)$.

"⇒" $p \models \neg\text{Merge}(b_2, \text{Merge}(b_1, ac'_P)) = \text{Merge}(b_2, \text{Merge}(b_1, ac_P))$ implies $\neg(p \models \text{Merge}(b_2, \text{Merge}(b_1, ac'_P)))$ by the satisfaction of conditions. By induction hypothesis, it follows that $\neg(p \models \text{Merge}(b_2 \circ b_1, ac'_P))$, i.e., $p \models \neg\text{Merge}(b_2 \circ b_1, ac'_P) = \text{Merge}(b_2 \circ b_1, ac_P)$.

"⇐" $p \models \neg\text{Merge}(b_2 \circ b_1, ac'_P) = \text{Merge}(b_2 \circ b_1, ac_P)$ implies $\neg(p \models \text{Merge}(b_2 \circ b_1, ac'_P))$ by the satisfaction of conditions. By induction hypothesis, it follows that $\neg(p \models \text{Merge}(b_2, \text{Merge}(b_1, ac'_P)))$, i.e., $p \models \neg\text{Merge}(b_2, \text{Merge}(b_1, ac'_P)) = \text{Merge}(b_2, \text{Merge}(b_1, ac_P))$.

Therefore, for some i and conditions ac_P in $\mathscr{M}$-normal form with i nestings, the equivalence $\text{Merge}(b_2, \text{Merge}(b_1, ac_P)) \equiv \text{Merge}(b_2 \circ b_1, ac_P)$ holds if $b_1 \in \mathscr{E}$ (induction hypothesis). For conditions in $\mathscr{M}$-normal form with $i+1$ nestings (induction step), the equivalence is shown by induction over the structure of condition ac_P again.

base case ($ac_P = \exists(a\colon P \hookrightarrow C, ac_C)$**)** The equivalence is shown based on the base case with $ac_P = \exists(a\colon P \hookrightarrow C, \textbf{true})$.

"⇒" By Lem. 3.11, assumption $b_1 \in \mathcal{E}$ implies $b'_1 \in \mathcal{E}$, since, (1) commutes and (a', b'_1) are jointly epimorphic by the merge construction. Therefore, by induction hypothesis, we can assume that $\text{Merge}(b'_2, \text{Merge}(b'_1, ac_C)) \equiv \text{Merge}(b'_2 \circ b'_1, ac_C)$ for condition ac_C in $\mathcal{M}$-normal form. Thus, $q \models \text{Merge}(b'_2, \text{Merge}(b'_1, ac_C))$ implies that $q \models \text{Merge}(b'_2 \circ b'_1, ac_C)$ and therefore, $p \models \text{Merge}(b_2 \circ b_1, ac_P)$ which is shown analogously to the base case.

"⇐" By Lem. 3.11, assumption $b_1 \in \mathcal{E}$ implies $b''_1 \in \mathcal{E}$, since, (1)+(2) commutes and $(b''_1, \underline{c'} \circ \underline{a'})$ are jointly epimorphic by the definition of pushout (1)+(2) (cf. Fact 2.17, item 2 in [EEPT06]). Therefore, by induction hypothesis, we can assume that $\text{Merge}(\underline{c''} \circ b'_2, \text{Merge}(b''_1, ac_C)) \equiv \text{Merge}(\underline{c''} \circ b'_2 \circ b''_1, ac_C)$ for condition ac_C in $\mathcal{M}$-normal form. Thus, $q \models \text{Merge}(b', ac_C) = \text{Merge}(\underline{c''} \circ b'_2 \circ b''_1, ac_C)$ implies that $q \models \text{Merge}(\underline{c''} \circ b'_2, \text{Merge}(b''_1, ac_C))$ and therefore, $p \models \text{Merge}(b_2, \text{Merge}(b_1, ac_P))$ which is shown analogously to the base case.

induction step (Disjunction, Conjunction and Negation) The equivalence is shown analogously to the disjunction, conjunction and negation in the induction base. □

A.10 Proof of Sec. 3.2 and Lem. 3.14

Given condition ac in $\mathcal{M}$-normal form and its AC-schema $\overline{ac}$. Furthermore, given match $m \in \mathcal{O}$ and instance morphism $i \in \mathcal{E}, \mathcal{O}$ by Def. 3.17. Note that the satisfaction of conditions by morphisms in Sec. 2.2.3 and Def. 2.12 is defined based on $\mathcal{O}$-morphisms. However, $i \circ m \in \mathcal{O}$ by composition Lem. 3.8 and Item 1 of $m, i \in \mathcal{O}$. The equivalence is shown by induction over the number i of nestings and the structure of ac (cf. Def. 2.12). **Basis.** For non-nested conditions ($i = 0$) we proceed as follows.

(base case $ac = \textbf{true}$) For $ac = \textbf{true}$, $m \models \overline{ac} = \textbf{true} \Leftrightarrow i \circ m \models \overline{ac} = \textbf{true}$.

(base case $ac = \exists(a\colon P \hookrightarrow C, \textbf{true})$)

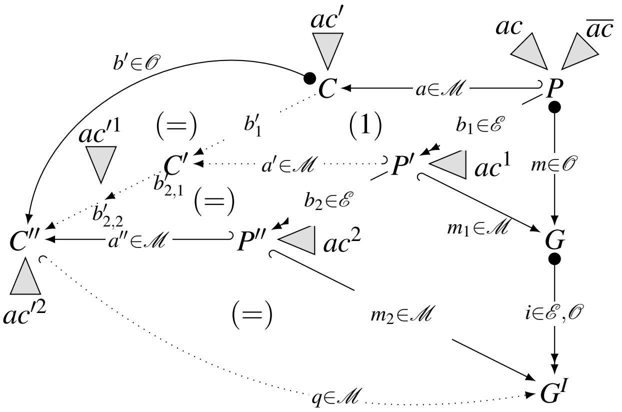
"⇒" Match m is $\mathcal{E}$-$\mathcal{M}$ factorised into $b_1 \in \mathcal{E}$ and $m_1 \in \mathcal{M}$ with $m_1 \circ b_1 = m$ $^{(*^1)}$. By Sec. 2.2.3 and Rem. 2.13, $m \models \overline{ac}$ implies $m_1 \models ac^1 = \text{Merge}(b_1, ac)$.

$$\begin{tikzcd}
 & & & & & & ac \ \triangle\ \triangle\ \overline{ac} \\
 & & & ac' \rhd C \arrow[dl, "b'_1\in\mathcal{O}"'] \arrow[drr, phantom, "(1)"] & & & P \arrow[lll, hook', "a\in\mathcal{M}"'] \arrow[dl, "b_1\in\mathcal{E}"] \arrow[ddd, "m\in\mathcal{O}"] \\
 & & ac'^1 \rhd C' \arrow[dl, dotted, "b'_2"'] \arrow[drr, phantom, "(2)"] & & & P' \lhd ac^1 \arrow[lll, hook', "a'\in\mathcal{M}"'] \arrow[dl, "b_2\in\mathcal{E}"] \arrow[ddr, "m_1\in\mathcal{M}"] & \\
 & ac'^2 \rhd C'' \arrow[ddrrrrr, dotted, "q'\in\mathcal{M}"'] & & & P'' \lhd ac^2 \arrow[lll, dotted, hook', "a''\in\mathcal{M}"'] \arrow[drr, dotted, "q\in\mathcal{M}"] \arrow[ddrr, "m_2\in\mathcal{M}"] & & \\
 & & & & & & G \arrow[d, "i\in\mathcal{E},\mathcal{O}"] \\
 & & & & & & G^I
\end{tikzcd}$$

$ac^1 = \text{Merge}(b_1, ac), ac^2 = \text{Merge}(b_2, \text{Merge}(b_1, ac))$
$ac'^1 = \text{Merge}(b'_1, ac'), ac'^2 = \text{Merge}(b'_2, \text{Merge}(b'_1, ac'))$

This implies that there exists a commuting diagram (1) by the merge construction (cf. Def. 2.16) with $a' \in \mathcal{M}, b'_1 \in \mathcal{O}$ and (a', b'_1) are jointly epimorphic $^{(*^2)}$. Furthermore, by the satisfaction of conditions (cf. Def. 2.12) there exists morphism $q\colon C' \to G \in \mathcal{M}$ with $q \circ a' = m_1$ and $q \models ac'^1 = \text{Merge}(b'_1, ac')$ $^{(*^3)}$. Morphism $i \circ m_1$ is $\mathcal{E}$-$\mathcal{M}$ factorised into $b_2 \in \mathcal{E}$ and $m_2 \in \mathcal{M}$ with $m_2 \circ b_2 = i \circ m_1$ $^{(*^4)}$. By $b_1, b_2 \in \mathcal{E}$ and Lem. 3.13 it follows that $b_2 \circ b_1 \in \mathcal{E}$. By the uniqueness of extremal $\mathcal{E}$-$\mathcal{M}$ factorisations and $i \circ m \overset{(*^1)}{=} i \circ m_1 \circ b_1 \overset{(*^4)}{=} m_2 \circ b_2 \circ b_1$, it follows that $m_2 \circ b_2 \circ b_1$ is the extremal $\mathcal{E}$-$\mathcal{M}$ factorisation of $i \circ m$ with $b_2 \circ b_1 \in \mathcal{E}$ and $m_2 \in \mathcal{M}$. Thus, by Sec. 2.2.3 and Rem. 2.13 it remains to show that $m_2 \models \text{Merge}(b_2 \circ b_1, ac)$ in order to show that $i \circ m \models \overline{ac}$. We construct pushout (2) along $a' \in \mathcal{M}$ with $a'' \in \mathcal{M}$ $^{(*^5)}$ by $\mathcal{M}$-morphisms are closed under pushouts. Diagram (2) is a merge diagram of $\text{Merge}(b_2, \text{Merge}(b_1, ac))$ as argued in the following. Diagram (2) commutes and (a'', b'_2) are jointly epimorphic by (2) being a pushout. Furthermore, $b'_2 \in \mathcal{O}$ as argued in the following. By the definition of instance morphism i (cf. Def. 3.17), it follows that $i \in \mathcal{O}$. Therefore, by $m_1 \in \mathcal{M} \subseteq \mathcal{O}$ from Lem. 3.8, item 1, it follows that $i \circ m_1 \in \mathcal{O}$. By $(*^4)$ and Lem. 3.8, item 2(a), it follows that $b_2 \in \mathcal{O}$. By Lem. 3.8, item 3, and (2) is a pushout, $b_2 \in \mathcal{O}$ implies that $b'_2 \in \mathcal{O}$. Since (2) is a diagram of $\text{Merge}(b_2, \text{Merge}(b_1, ac)) = \vee_i ac_i$, it remains to show that $m_2 \models ac_i = \exists(a'', ac'^2)$ with $ac'^2 = \text{Merge}(b'_2, \text{Merge}(b'_1, ac')) = \textbf{true}$ by assumption $ac' = \textbf{true}$ (cf. Def. 2.16) implying that $m_2 \models \text{Merge}(b_2, \text{Merge}(b_1, ac))$ implying further that $m_2 \models \text{Merge}(b_2 \circ b_1, ac)$ by Lem. 3.12 with ac being in $\mathcal{M}$-normal form and $b_1 \in \mathcal{E}$. By the universal pushout property of (2), there exists morphism $q'\colon C'' \to G^I$ with $q' \circ a'' = m_2$ and $q' \circ b'_2 = i \circ q$ $^{(*^6)}$, since, $m_2 \circ b_2 \overset{(*^4)}{=} i \circ m_1 \overset{(*^3)}{=} i \circ q \circ a'$. Since, $q' \models \textbf{true} = ac'^2$, it remains to show that $q' \in \mathcal{M}$. By Def. 2.12 it would follow that $m_2 \models \exists(a'', ac'^2)$. Morphisms $a'', m_2 \in \mathcal{M}$ imply that $a''_D, m_{2,D}$ are isomorphisms. Thus, by $(*^6)$, $q'_D \circ a''_D = m_{2,D} \Rightarrow q'_D \circ a''_D \circ a''^{-1}_D = m_{2,D} \circ a''^{-1}_D \Leftrightarrow q'_D = m_{2,D} \circ a''^{-1}_D$. By the composition of isomorphisms with inverse isomorphism a''^{-1}_D, it follows that q'_D is an isomorphism. By the definition of instance morphism i (cf. Def. 3.17), i_S is an isomorphism, i.e., $i_S \circ m_{1,S} \in \mathcal{M}$ by $\mathcal{M}$-composition and $b_{2,S} \in \mathcal{M}$ by $(*^4)$, $m_{2,S} \in \mathcal{M}$ and $\mathcal{M}$-decomposition. Morphism $b_2 \in \mathcal{E}$ implies that $b_{2,S}$ is an epimorphism. Thus, by Lem. 3.10, it follows that $b_{2,S}$ is an isomorphism and therefore, $b'_{2,S}$ is an isomorphism, since, isomorphisms are preserved by pushouts. Consequently, by $(*^6)$ $q'_S \circ b'_{2,S} = i_S \circ q_S \Rightarrow q'_S \circ b'_{2,S} \circ b'^{-1}_{2,S} = i_S \circ q_S \circ b'^{-1}_{2,S} \Leftrightarrow q'_S = i_S \circ q_S \circ b'^{-1}_{2,S}$. Therefore, $q'_S \in \mathcal{M}$ by $\mathcal{M}$-composition with $i_S, q_S, b'^{-1}_{2,S} \in \mathcal{M}$ and furthermore, q' is type strict by $(*^6)$ and $m_2 \in \mathcal{M}$ is type strict, i.e., $q' \in \mathcal{M}$.

"$\Leftarrow$" Morphism m is $\mathcal{E}$-$\mathcal{M}$ factorised into $b_1 \in \mathcal{E}$ and $m_1 \in \mathcal{M}$ with $m_1 \circ b_1 = m$ $^{(*^1)}$. Morphism $i \circ m_1$ is $\mathcal{E}$-$\mathcal{M}$ factorised into $b_2 \in \mathcal{E}$ and $m_2 \in \mathcal{M}$ with $i \circ m_1 = m_2 \circ b_2$ $^{(*^2)}$. By Lem. 3.13, $b_2 \circ b_1 \in \mathcal{E}$ with $(*^1)$ $m_1 \circ b_1 = m \Rightarrow i \circ m_1 \circ b_1 = i \circ m \overset{(*^2)}{\Leftrightarrow} m_2 \circ b_2 \circ b_1 = i \circ m$ $^{(*^3)}$. By the uniqueness of $\mathcal{E} - \mathcal{M}$ factorisations, $(b_2 \circ b_1 \in \mathcal{E}, m_2 \in \mathcal{M})$ is a factorisation of $i \circ m$. Assumption $i \circ m \models \overline{ac}$ with $i \circ m \in \mathcal{O}$ by Lem. 3.8 and Item 1 implies $m_2 \models \text{Merge}(b_2 \circ b_1, ac)$ $^{(*^4)}$ by Sec. 2.2.3 and Rem. 2.13. Therefore by Def. 2.16, there exists commuting diagram $b' \circ a = a'' \circ b_2 \circ b_1$ with $a'' \in \mathcal{M}$, $b' \in \mathcal{O}$ and (a'', b') are jointly epimorphic. Furthermore, there exists morphism $q \in \mathcal{M}$ with $q \circ a'' = m_2$ and $q \models \text{Merge}(b', ac')$ $^{(*^5)}$. We construct pushout (1) with $a' \in \mathcal{M}$ by $\mathcal{M}$-morphisms are closed under pushouts. By universal pushout property there is $b'_{2,2} \circ b'_{2,1}$ with $b'_{2,2} \circ b'_{2,1} \circ b'_1 = b'$ and $b'_{2,2} \circ b'_{2,1} \circ a' = a'' \circ b_2$ $^{(*^4)}$. We construct epimorphism $b'_{2,1}\colon C' \to \overline{C}$ with data part $b'_{2,1,D}$ being an isomorphism such that $b'_{2,2}$ is type strict. By Lem. 3.11 with $b_1 \in \mathscr{E}$ and pushout (1), i.e., (1) commutes and (a',b'_1) are jointly epimorphic, it follows that $b'_1 \in \mathscr{E}$. Therefore, $(*^4)$ together with $b' \in \mathscr{O}$ (b'_S is injective) imply that $b'_{1,S}$ is injective ($b'_1 \in \mathscr{O}$) by injective morphisms are closed under decomposition and furthermore, $b'_1 \in \mathscr{E}$ ($b'_{1,S}$ is epimorphism, i.e., surjective) imply that $b'_{2,2,S} \circ b'_{2,1,S}$ is injective and therefore, also $b'_{2,1,S}$ is injective ($b'_{2,1} \in \mathscr{O}$). Analogously, $b'_{2,1,S}$ is epimorphism, i.e., surjective, with $b'_{2,2,S} \circ b'_{2,1,S}$ is injective imply that $b'_{2,2,S}$ is injective.



$$ac^1 = \text{Merge}(b_1,ac), ac^2 = \text{Merge}(b_2 \circ b_1, ac)$$
$$ac'^1 = \text{Merge}(b'_{2,1} \circ b'_1, ac'), ac'^2 = \text{Merge}(b', ac')$$

Furthermore, $b'_{2,1} \circ a'$ is type strict, since, instance morphism i is type strict by Def. 3.17 and Item 3 and $m_1 \in \mathscr{M}$ is type strict $\Rightarrow$ $i \circ m_1$ is type strict $\overset{(*^2)}{\Rightarrow}$ $m_2 \circ b_2$ is type strict $\Rightarrow$ b_2 is type strict $\overset{a''\in\mathscr{M}}{\Rightarrow}$ $a'' \circ b_2$ is type strict $\overset{(*^4)}{\Rightarrow}$ $b'_{2,2} \circ b'_{2,1} \circ a'$ is type strict $\Rightarrow$ $b'_{2,1} \circ a'$ is type strict. Thus, $b'_{2,1,S} \circ a'_S$ is injective by composition of injective morphisms, $b'_{2,1,D} \circ a'_D$ is an isomorphism by composition of isomorphisms and $b'_{2,1} \circ a'$ is type strict implying further that $b'_{2,1} \circ a' \in \mathscr{M}$. Moreover by Lem. 3.8 and Item 1, $b'_{2,1} \circ b'_1 \in \mathscr{O}$, $b'_{2,1} \circ b'_1 \circ a \overset{(1)}{=} b'_{2,1} \circ a' \circ b_1$ and $(b'_{2,1} \circ b'_1, b'_{2,1} \circ a')$ are jointly epimorphic by Lem. 3.9 and Item 4 with $b'_{2,1}$ being an epimorphism and (b'_1, a') being jointly epimorphic by pushout (1) and therefore, (1) with subsequent morphism $b'_{2,1}$ is a diagram of $\text{Merge}(b_1,ac) = \vee_i ac_i$ with $ac_i = \exists(b'_{2,1} \circ a', ac'^1)$ and $ac'^1 = \text{Merge}(b'_{2,1} \circ b'_1, ac') = \textbf{true}$ for assumption $ac' = \textbf{true}$ (cf. Def. 2.16). It remains to show that there is a morphism $q'\colon \overline{C} \to G \in \mathscr{M}$ such that $q' \circ b'_{2,1} \circ a' = m_1$. As implicitly $q' \models ac'^1 = \textbf{true}$ this would imply that $m_1 \models \text{Merge}(b_1,ac)$ implying further that $m \models \overline{ac}$ by $(*^1)$ and Sec. 2.2.3 and Rem. 2.13 with $m \in \mathscr{O}$. We construct q' as follows: By Def. 3.17, instance morphism i is in $\mathscr{E}$ and $\mathscr{O}$, i.e., i_S is an isomorphism. Therefore, $q'_S = i_S^{-1} \circ q_S \circ b'_{2,2,S}$ with inverse isomorphism i_S^{-1}. By Sec. 2.2.2 and Rem. 2.3, $a', m_1 \in \mathscr{M}$ imply that $a'_D, m_{1,D}$ are isomorphisms. Thus, $q'_D = m_{1,D} \circ a'^{-1}_D \circ b'^{-1}_{2,1,D}$ with inverse isomorphisms $b'^{-1}_{2,1,D}$ and a'^{-1}_D. It holds that $q' \circ b'_{2,1} \circ a' = m_1$ as shown in the following: $q'_D \circ b'_{2,1,D} \circ a'_D = m_{1,D} \circ a'^{-1}_D \circ b'^{-1}_{2,1,D} \circ b'_{2,1,D} \circ a'_D \overset{b'_{2,1,D},a'_D \text{ are isos}}{=} m_{1,D}$ and $q'_S \circ b'_{2,1,S} \circ a'_S = i_S^{-1} \circ q_S \circ b'_{2,2,S} \circ b'_{2,1,S} \circ a'_S \overset{(*^4)}{=} i_S^{-1} \circ q_S \circ a''_S \circ b_{2,S} \overset{(*^5)}{=} i_S^{-1} \circ m_{2,S} \circ b_{2,S} \overset{(*^2)}{=} i_S^{-1} \circ i_S \circ m_{1,S} \overset{i_S \text{ is iso}}{=} m_{1,S}$. It remains to show that:

1. q' is a typed attributed graph morphism concerning separately defined q'_S and

q'_D, i.e., $t^G_{EA} \circ q'_{G,E_{EA}} = q'_D \circ t^{\overline{C}}_{EA}$ and $t^G_{NA} \circ q'_{G,E_{NA}} = q'_D \circ t^{\overline{C}}_{NA}$, and that q' is in $\mathscr{M}$, i.e.,:

2. q' is type strict,
3. q'_S is injective, and
4. q'_D is an isomorphism.

1. $\forall e \in E^{\overline{C}}_j, j \in \{EA, NA\}$ we have $i_D(q'_D(t^{\overline{C}}_j(e))) \overset{q'_D}{=} i_D(m_{1,D}(a'^{-1}_D(b'^{-1}_{2,1,D}(t^{\overline{C}}_j(e)))))$ $\overset{(*^2),(*^4),(*^5)}{=}$ $q_D(b'_{2,2,D}(b'_{2,1,D}(a'_D(a'^{-1}_D(b'^{-1}_{2,1,D}(t^{\overline{C}}_j(e)))))))$ $\overset{b'_{2,1,D} \text{ and } a'_D \text{ are isos}}{=}$ $q_D(b'_{2,2,D}(t^{\overline{C}}_j(e)))$ $\overset{q \circ b'_{2,2} \in Mor}{=}$ $t^{G^I}_j(q_{G,E_j}(b'_{2,2,G,E_j}(e)))$. Furthermore, we have $i_D(t^G_j(q'_{G,E_j}(e))) \overset{q'_S}{=} i_D(t^G_j(i^{-1}_{G,E_j}(q_{G,E_j}(b'_{2,2,G,E_j}(e)))))$ $\overset{i \in Mor}{=}$ $t^{G^I}_j(i_{G,E_j}(i^{-1}_{G,E_j}(q_{G,E_j}(b'_{2,2,G,E_j}(e)))))$ $\overset{i_S \text{ is iso}}{=}$ $t^{G^I}_j(q_{G,E_j}(b'_{2,2,G,E_j}(e)))$. Thus, $i_D(t^G_j(q'_{G,E_j}(e))) = i_D(q'_D(t^{\overline{C}}_j(e)))$ and furthermore, $i_D(t^G_j(q'_{G,E_j}(e))) \in t^{G^I}_j(E^{G^I}_j)$. Assumption $t^G_j(q'_{G,E_j}(e)) \neq q'_D(t^{\overline{C}}_j(e))$ contradicts Def. 3.17 and Item 5. Therefore, $t^G_j(q'_{G,E_j}(e)) = q'_D(t^{\overline{C}}_j(e))$, i.e., $t^G_j \circ q'_{G,E_j} = q'_D \circ t^{\overline{C}}_j$.
2. Morphism $q'_S = i^{-1}_S \circ q_S \circ b'_{2,2,S}$ and therefore q' is type strict, since, $b'_{2,2,S}$, q_S and i^{-1}_S are type strict by the construction of $b'_{2,1}$, $q \in \mathscr{M}$ (cf. Sec. 2.2.2 and Rem. 2.3) and by Def. 3.17 for instance morphism i together with inverse isomorphism i^{-1}_S.
3. Morphism $q'_S = i^{-1}_S \circ q_S \circ b'_{2,2,S}$ is injective by composition of injective morphisms where i^{-1}_S is injective by being an isomorphism (cf. Sec. 2.2.2 and Rem. 2.2), q_S is injective by $q_S \in \mathscr{M}$ (cf. Sec. 2.2.2 and Rem. 2.3) and $b'_{2,2,S}$ is injective as already shown before.
4. Morphism $q'_D = m_{1,D} \circ a'^{-1}_D \circ b'^{-1}_{2,1,D}$ is an isomorphism by composition of isomorphisms where $b'^{-1}_{2,1,D}$ and a'^{-1}_D are inverse isomorphisms as already shown before and $m_{1,D}$ is an isomorphism by $m_1 \in \mathscr{M}$ (cf. Sec. 2.2.2 and Rem. 2.3).

(hypothesis) There are conditions ac' and $(ac_i)_{i \in I}$ in $\mathscr{M}$-normal form such that $m \models \overline{ac}'$ if and only if $i \circ m \models \overline{ac}'$ and $m \models \overline{ac}_i$ if and only if $i \circ m \models \overline{ac}_i$ for all $i \in I$.

(step $ac = \wedge_{i \in I}(ac_i)$, $ac = \vee_{i \in I}(ac_i)$ and $ac = \neg ac'$) Note that ac_i, ac' are in $\mathscr{M}$-normal form by ac is in $\mathscr{M}$-normal form. For $ac = \wedge_{i \in I}(ac_i)$, $m \models \overline{ac} \overset{Sec.\ 2.2.3 and\ Rem.\ 2.13}{\Leftrightarrow} m_1 \models \text{Merge}(e_1, ac)$ for extremal $\mathscr{E}$-$\mathscr{M}$ factorisation $m_1 \circ e_1 = m \overset{Sec.\ 2.2.3 and\ Def.\ 2.16}{\Leftrightarrow} m_1 \models \wedge_{i \in I}(\text{Merge}(e_1, ac_i)) \overset{Sec.\ 2.2.3 and\ Def.\ 2.12}{\Leftrightarrow} \forall i \in I . m_1 \models \text{Merge}(e_1, ac_i) \overset{Sec.\ 2.2.3 and\ Rem.\ 2.13}{\Leftrightarrow} \forall i \in I . m \models \overline{ac}_i \overset{hypothesis}{\Leftrightarrow} \forall i \in I . i \circ m \models \overline{ac}_i \overset{Sec.\ 2.2.3 and\ Rem.\ 2.13}{\Leftrightarrow} \forall i \in I . m_2 \models \text{Merge}(e_2, ac_i)$ for extremal $\mathscr{E}$-$\mathscr{M}$ factorisation $m_2 \circ e_2 = i \circ m \overset{Sec.\ 2.2.3 and\ Def.\ 2.12}{\Leftrightarrow} m_2 \models \wedge_{i \in I}(\text{Merge}(e_2, ac_i)) \overset{Sec.\ 2.2.3 and\ Def.\ 2.16}{\Leftrightarrow} m_2 \models \text{Merge}(e_2, ac) \overset{Sec.\ 2.2.3 and\ Rem.\ 2.13}{\Leftrightarrow} i \circ m \models \overline{ac}$. The equivalence for $ac = \vee_{i \in I}(ac_i)$ is shown analogously. For $ac = \neg ac'$, $m \models \overline{ac} \overset{Sec.\ 2.2.3 and\ Rem.\ 2.13}{\Leftrightarrow} m_1 \models \text{Merge}(e_1, ac)$ for extremal $\mathscr{E}$-$\mathscr{M}$ factorisation $m_1 \circ e_1 = m \overset{Sec.\ 2.2.3 and\ Def.\ 2.16}{\Leftrightarrow} m_1 \models \neg(\text{Merge}(e_1, ac')) \overset{Sec.\ 2.2.3 and\ Def.\ 2.12}{\Leftrightarrow} \neg(m_1 \models \text{Merge}(e_1, ac')) \overset{Sec.\ 2.2.3 and\ Rem.\ 2.13}{\Leftrightarrow} \neg(m \models \overline{ac}') \overset{hypothesis}{\Leftrightarrow} \neg(i \circ m \models \overline{ac}') \overset{Sec.\ 2.2.3 and\ Rem.\ 2.13}{\Leftrightarrow} \neg(m_2 \models \text{Merge}(e_2, ac'))$ for extremal $\mathscr{E}$-$\mathscr{M}$ factorisation $m_2 \circ e_2 = i \circ m \overset{Sec.\ 2.2.3 and\ Def.\ 2.12}{\Leftrightarrow} m_2 \models \neg(\text{Merge}(e_2, ac')) \overset{Sec.\ 2.2.3 and\ Def.\ 2.16}{\Leftrightarrow} m_2 \models \text{Merge}(e_2, ac) \overset{Sec.\ 2.2.3 and\ Rem.\ 2.13}{\Leftrightarrow} i \circ m \models \overline{ac}$.

Hypothesis. There is number i of nestings for ac' in $\mathscr{M}$-normal form such that $m \models \overline{ac}'$ if and only if $i \circ m \models \overline{ac}'$. **Step.** For $i+1$ nestings, we conclude as follows.

(base case $ac = \exists(a\colon P \hookrightarrow C, ac')$**)**

"$\Rightarrow$" We close analogously to the base case $ac = \exists(a\colon P \to C, \mathbf{true})$ for direction "$\Rightarrow$" in the basis for non-nested conditions. Note that 1. $q \circ b'_1 \in \mathscr{O}$ $^{(*^B)}$ by $q \in \mathscr{M}$ ($\Rightarrow q \in \mathscr{O}$ – cf. Sec. 2.2.2 and Rem. 2.3) and Lem. 3.8 and Item 1, 2. $q' \circ b'_2 \circ b'_1 \in \mathscr{O}$ $^{(*^C)}$ by $q' \circ b'_2 \circ b'_1 \overset{(*^6)}{=} i \circ q \circ b'_1$, $q \in \mathscr{M}$ ($\Rightarrow q \in \mathscr{O}$ – cf. Sec. 2.2.2 and Rem. 2.3) and Lem. 3.8 and Item 1, and 3. ac' is in $\mathscr{M}$-normal form $^{(*^D)}$ by assumption ac is in $\mathscr{M}$-normal form. Furthermore, by Lem. 3.11 with $(*^1),(*^2)$ it follows that $b'_1 \in \mathscr{E}$ and by Lem. 3.11 with $(*^4)$, diagram (2) commutes, (a'', b'_2) are jointly epimorphic it follows that $b'_2 \in \mathscr{E}$, i.e., $b'_2 \circ b'_1 \in \mathscr{E}$ by Lem. 3.13 $\Rightarrow q \circ b'_1$ and $q' \circ b'_2 \circ b'_1$ are extremal $\mathscr{E}$-$\mathscr{M}$ factorisations $^{(*^A)}$, implying further that $q \models \mathrm{Merge}(b'_1, ac')$ $\overset{Sec.\ 2.2.3 and\ Rem.\ 2.13, (*^A),(*^B)}{\Rightarrow}$ $q \circ b'_1 \models \overline{ac}'$ $\overset{Hypothesis, (*^B),(*^D)}{\Rightarrow}$ $i \circ q \circ b'_1 \models \overline{ac}' \overset{(*^6)}{=} q' \circ b'_2 \circ b'_1 \models \overline{ac}'$ $\overset{Sec.\ 2.2.3 and\ Rem.\ 2.13, (*^A),(*^C)}{\Rightarrow}$ $q' \models \mathrm{Merge}(b'_2 \circ b'_1, ac')$ $\overset{Lem.\ 3.12, b'_1 \in \mathscr{E}, (*^D)}{\Rightarrow}$ $q' \models \mathrm{Merge}(b'_2, \mathrm{Merge}(b'_1, ac')) = ac'^2$. Thus, $i \circ m \models \overline{ac}$.

"$\Leftarrow$" We close analogously to the base case $ac = \exists(a\colon P \to C, \mathbf{true})$ for direction "$\Leftarrow$" in the basis for non-nested conditions. Note that 1. $q' \circ b'_{2,1} \circ b'_1 \in \mathscr{O}$ $^{(*^B)}$ by $q' \in \mathscr{M}$ ($\Rightarrow q' \in \mathscr{O}$ – cf. Sec. 2.2.2 and Rem. 2.3) and Lem. 3.8 and Item 1, 2. $q \circ b' \in \mathscr{O}$ $^{(*^C)}$ by $q \in \mathscr{M}$ ($\Rightarrow q \in \mathscr{O}$ – cf. Sec. 2.2.2 and Rem. 2.3) and Lem. 3.8 and Item 1, 3. ac' is in $\mathscr{M}$-normal form $^{(*^D)}$ by assumption ac is in $\mathscr{M}$-normal form, and 4. $i \circ q' = q \circ b'_{2,2}$ $^{(*^E)}$, since, $i_S \circ q'_S \overset{\text{Def. } q'_S}{=} i_S \circ i_S^{-1} \circ q_S \circ b'_{2,2,S} \overset{i_S \text{ is iso}}{=} q_S \circ b'_{2,2,S}$ and $i_D \circ q'_D \overset{\text{Def. } q'_D}{=} i_D \circ m_{1,D} \circ a'^{-1}_D \circ b'^{-1}_{2,1,D} \overset{(*^2)}{=} m_{2,D} \circ b_{2,D} \circ a'^{-1}_D \circ b'^{-1}_{2,1,D} \overset{(*^5)}{=} q_D \circ a''_D \circ b_{2,D} \circ a'^{-1}_D \circ b'^{-1}_{2,1,D} \overset{(*^4)}{=} q_D \circ b'_{2,2,D} \circ b'_{2,1,D} \circ a'_D \circ a'^{-1}_D \circ b'^{-1}_{2,1,D} \overset{a'^{-1}_D, b'^{-1}_{2,1,D} \text{ are isos}}{=} q_D \circ b'_{2,2,D}$. Furthermore, by Lem. 3.11 with $(*^1)$, (1) commutes and (a', b'_1) are jointly epimorphic, it follows that $b'_1 \in \mathscr{E}$ and furthermore, epimorphism $b'_{2,1} \in \mathscr{E}$ in $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$, i.e., $b'_{2,1} \circ b'_1 \in \mathscr{E}$ by Lem. 3.13, and by Lem. 3.13 with $b_1, b_2 \in \mathscr{E}$, outer diagram commutes, (a'', b') are jointly epimorphic it follows that $b' \in \mathscr{E}$, i.e., $q' \circ b'_{2,1} \circ b'_1$ and $q \circ b'$ are extremal $\mathscr{E}$-$\mathscr{M}$ factorisations $^{(*^A)}$, implying further that $q \models \mathrm{Merge}(b', ac')$ $\overset{Sec.\ 2.2.3 and\ Rem.\ 2.13, (*^A),(*^C)}{\Rightarrow}$ $q \circ b' \models \overline{ac}'$ $\overset{(*^4),(*^E)}{\Rightarrow}$ $i \circ q' \circ b'_{2,1} \circ b'_1 \models \overline{ac}'$ $\overset{Hypothesis, (*^B),(*^D)}{\Rightarrow}$ $q' \circ b'_{2,1} \circ b'_1 \models \overline{ac}'$ $\overset{Sec.\ 2.2.3 and\ Rem.\ 2.13, (*^A),(*^B)}{\Rightarrow}$ $q' \models \mathrm{Merge}(b'_{2,1} \circ b'_1, ac') = ac'^1$. Thus, $m \models \overline{ac}$.

(hypothesis) There are conditions ac' and $(ac_i)_{i \in I}$ in $\mathscr{M}$-normal form such that $m \models \overline{ac}'$ if and only if $i \circ m \models \overline{ac}'$ and $m \models \overline{ac}_i$ if and only if $i \circ m \models \overline{ac}_i$ for all $i \in I$.

(step $ac = \wedge_{i \in I}(ac_i)$**,** $ac = \vee_{i \in I}(ac_i)$ **and** $ac = \neg ac'$**)** We close analogously to the step for conjunctions, disjunctions and negations in the basis for non-nested conditions. □

A.11 Proof of Sec. 3.2 and Thm. 3.3

Let $G \underset{I}{=} ((G^E, D_G), type_G)$ be a graph in $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ with $G \in \mathscr{L}(C) = \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$, i.e., $G \models C_I$ and $G \models C_G$ with $C = C_I \cup C_G$ and C_I being the contained constraints that are designated for initial satisfaction and C_G being the contained constraints that are designated for general satisfaction (cf. Sec. 3.1 and Def. 3.1). Let $i\colon G_A \to G$ be an instance morphism, i.e., graph G_A is the abstraction of G in the sense that G_A shares the *DSIG*-term algebra $T_{DSIG}(X)$ where all attribute values in G are substituted by variables $x \in X$ in G_A. By Lem. 3.5, there are atoms $A = (a_i)_{i \in \{1,\dots,n\}} \in Atoms(G_A)$ of G_A with induced morphisms $i_a\colon a \to G_A \in \mathscr{M}, \forall a \in A$. Note that $\forall a \in A.a \in Atoms(ATG)$ (cf. Def. 3.6), since, a shares algebra $T_{DSIG}(X)$ up to isomorphism by $i_a \in \mathscr{M}$, i.e., $i_{a,D}$ being an isomorphism, and furthermore, all attribute values in a are variables $x \in X$ by $i_{a,D}$ being the unique homomorphism for a given variable assignment that explicitly maps terms $t \notin X$ in a to terms $t' \notin X$ in G_A by Fact B.16, Item 1, and Def. B.14 in [EEPT06]. Furthermore, $\forall a \in A.a \in EAtoms(C)$ by induced morphism $i_a\colon a \to G_A \in \mathscr{M}$ and $G_A \in \mathscr{L}(C)$ (cf. Def. 3.7). We show that $G_A \in \mathscr{L}(C) = \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$, i.e., $G_A \overset{I}{\models} C_I$ and $G_A \models C_G$, as follows. By general assumption, constraints are interpreted via their AC-schemata, i.e., $G \in \mathscr{L}(C)$ implies that

1. G initially satisfies AC-schema $\overline{ac_I}$ of constraint ac_I for all $ac_I \in C_I$, and

2. G generally satisfies AC-schema $\overline{ac_G}$ of constraint ac_G for all $ac_G \in C_G$.

We have to show that

1. $\Rightarrow G_A$ initially satisfies AC-schema $\overline{ac_I}$ of constraint ac_I for all $ac_I \in C_I$, and

2. $\Rightarrow G_A$ generally satisfies AC-schema $\overline{ac_G}$ of constraint ac_G for all $ac_G \in C_G$.

1. For all constraints $ac_I \in C_I$ over P: G initially satisfies $\overline{ac_I}$ implies that $\exists p\colon P \to G \in \mathscr{O}$ such that $p \models \overline{ac_I}$ by Sec. 2.2.3 and Def. 2.19. We have to show that $\exists p'\colon P \to G_A \in \mathscr{O}$ such that $p' \models \overline{ac_I}$ implying further that G_A initially satisfies $\overline{ac_I}$. Note that instance morphism $i\colon G_A \to G$ is in $\mathscr{O}$ and in $\mathscr{E}$ by Def. 3.17, i.e., graph part i_S is an isomorphism with inverse isomorphism i_S^{-1}. Thus, we can construct morphism $p'\colon P \to G_A$ such that $i \circ p' = p$ as follows: Graph part $p'_S = i_S^{-1} \circ p_S$. Note that analogously to G_A also P shares the *DSIG*-term algebra $T_{DSIG}(X)$ with all attribute values in P being variables $x \in X$ by the general assumption. Therefore, we can define the variable assignment $asg\colon X \to D_{G_A}$ from P to G_A such that $\forall e \in E_j^P.asg(t_j^P(e)) = t_j^{G_A}(p'_S(e)), j \in \{NA, EA\}$ $^{(*^1)}$. W.l.o.g. we assume that instance morphism $i\colon G_A \to G$ is given such that $i_D(asg(x)) = p_D(x), \forall x \in X$ $^{(*^2)}$. Based on variable assignment asg, we construct data part $p'_D = \overline{asg}\colon T_{DSIG}(X) \to T_{DSIG}(X)$ as defined in Def. B.14 in [EEPT06]. By the construction of $\overline{asg}$ it follows that $\forall e \in E_j^P.p'_D(t_j^P(e)) = t_j^{G_A}(p'_S(e)), j \in \{NA, EA\}$ by $(*^1)$, i.e., p' is actually a typed attributed graph morphism, and $i_D(p'_D(x)) = p_D(x), \forall x \in X$ by $(*^2)$ $^{(*^3)}$. By Fact B.16, Item 1, in [EEPT06] the homomorphism $T_{DSIG}(X) \to D_G$ w.r.t. a given variable assignment $asg\colon X \to D_G$ is unique and therefore, $i_D \circ p'_D = p_D$ by $(*^3)$. Furthermore, $p'_S = i_S^{-1} \circ p_S$ $\Rightarrow i_S \circ p'_S = i_S \circ i_S^{-1} \circ p_S \overset{i_S \text{ is iso}}{\Rightarrow} i_S \circ p'_S = p_S$. Thus, $i \circ p' = p$ implying further that $p' \in \mathscr{O}$ by $p \in \mathscr{O}$ and Lem. 3.8 and Item 2a. Therefore, by Lem. 3.14 with ac_I being in $\mathscr{M}$-normal form by assumption, $p = i \circ p' \models \overline{ac_I}$ implies $p' \models \overline{ac_I}$. Thus, G_A initially satisfies $\overline{ac_I}$.

2. For all constraints $ac_G \in C_G$ over P: G_A generally satisfies AC-schema $\overline{ac_G}$ means that $\forall p\colon P \to G_A \in \mathscr{O}.p \models \overline{ac_G}$ by Sec. 2.2.3 and Def. 2.19. Let $p\colon P \to G_A \in \mathscr{O}$. Then, assumption G generally satisfies $\overline{ac_G}$ implies that $i \circ p \models \overline{ac_G}$, since, $i \circ p \in \mathscr{O}$ by $p, i \in \mathscr{O}$ and $\mathscr{O}$-composition in Lem. 3.8 and Item 1. By Lem. 3.14 with ac_G being in $\mathscr{M}$-normal form by assumption, $i \circ p \models \overline{ac_G}$ implies $p \models \overline{ac_G}$. Thus, G_A generally satisfies $\overline{ac_G}$.

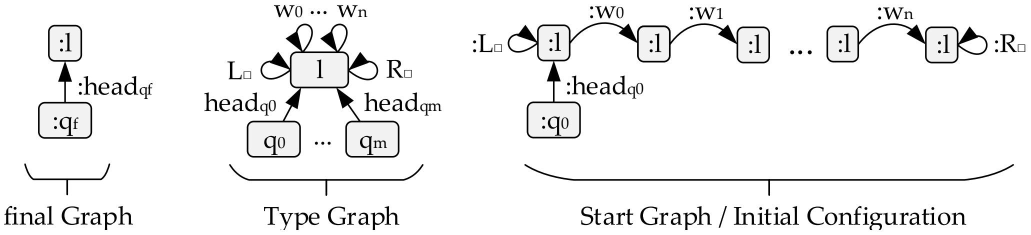
By, $\forall a \in A. a \in EAtoms(C)$ and $\mathscr{L}(GG')$ is C-extension complete it follows that $\forall a \in A. \exists S \in Extensions(a, C_G)$ such that $S \subseteq \mathscr{L}(GG')$ by Def. 3.8. Thus, $\forall a \in A. \exists S_a \in Extensions(a, C_G)$ such that $\forall s \in S_a. \exists$ transformation $\varnothing' \overset{*}{\Rightarrow} s$ via P and almost injective matches by general assumption in Sec. 2.2.4 $^{(*^A)}$. By Def. 3.16, there is a function $f_E \in SELECT_E(A, C_G)$ with $f_E(a) = S_a, \forall a \in A$. By Lem. 3.7, there is a function $f_{a_E} \in SELECT_{a_E}(A, C_G, f_E)$ with $f_{a_E}(a) = s \in S_a, \forall a \in A = (a_i)_{i \in \{1,\dots,n\}}$ (cf. Def. 3.16) such that there exist graphs $(G^E_j)_{1 \leq j \leq n-1}$ and pushouts $(PO^E_k +_{G^E_k} f_{a_E}(a_{k+1}) = PO^E_{k+1})_{k \in \{1,\dots,n-1\}}$ with pushout objects PO^E_{k+1}, $PO^E_1 = f_{a_E}(a_1)$ and injective embeddings $i_{k,1} \colon PO^E_k \to PO^E_{k+1}$ and $i_{k,2} \colon f_{a_E}(a_{k+1}) \to PO^E_{k+1} \in \mathscr{M}$ where $PO^E_n = G_A$ $^{(*^B)}$. For pushout $k = 1$ we conclude as follows. By Lem. 3.2 with $(*^A), (*^B)$ and $m(GG')$ being the set of derived marking rules of GG', it follows that there exists transformations $t_1 \colon PO^E_2 \oplus Att^{\mathbf{F}}_{PO^E_2} \overset{*}{\Rightarrow} PO^E_2 \oplus Att^{\mathbf{T}}_{f_{a_E}(a_1)} \oplus Att^{\mathbf{F}}_{PO^E_2 \setminus f_{a_E}(a_1)}$ and $t_2 \colon PO^E_2 \oplus Att^{\mathbf{F}}_{PO^E_2} \overset{*}{\Rightarrow} PO^E_2 \oplus Att^{\mathbf{T}}_{f_{a_E}(a_2)} \oplus Att^{\mathbf{F}}_{PO^E_2 \setminus f_{a_E}(a_2)}$ via marking rules $m(GG')$. Assumption $m(GG')$ is C-conflict-free implies that all critical pairs over productions $m(GG')$ that are significant w.r.t. $\mathscr{L}(C)$ are strictly confluent by Def. 3.12 – Note that the assumption may also imply that all critical pairs over P that are not C-inconsistent share the same rule and match, respectively. However, the second implication implies the first implication by Remarks. 3.6 and 3.7 and therefore, it is sufficient to continue we the first implication. By local confluence theorem Thm. 2.43 in [EEGH15], transformation system $m(GG')$ is locally confluent and also terminating by Sec. 3.2 and Rem. 3.5 with all productions in GG being non-trivial by assumption, implying further that $m(GG')$ is confluent by Lem. 3.32 in [EEPT06]. Thus, since, PO^E_2 is the pushout object, it follows that embeddings $i_{1,1} \colon f_{a_E}(a_1) \to PO^E_2$ and $i_{1,2} \colon f_{a_E}(a_2) \to PO^E_2$ are jointly surjective and therefore, there are transformations $PO^E_2 \oplus Att^{\mathbf{T}}_{f_{a_E}(a_1)} \oplus Att^{\mathbf{F}}_{PO^E_2 \setminus f_{a_E}(a_1)} \overset{*}{\Rightarrow} PO^E_2 \oplus Att^{\mathbf{T}}_{PO^E_2}$ and $PO^E_2 \oplus Att^{\mathbf{T}}_{f_{a_E}(a_2)} \oplus Att^{\mathbf{F}}_{PO^E_2 \setminus f_{a_E}(a_2)} \overset{*}{\Rightarrow} PO^E_2 \oplus Att^{\mathbf{T}}_{PO^E_2}$ via $m(GG')$ leading to transformation $PO^E_2 \oplus Att^{\mathbf{F}}_{PO^E_2} \overset{*}{\Rightarrow} PO^E_2 \oplus Att^{\mathbf{T}}_{PO^E_2}$. Thus by Lem. 3.2, there exists a transformation $\varnothing' \overset{*}{\Rightarrow} PO^E_2$ via P and almost injective matches with injective embedding $id_{PO^E_2}$. Analogously, we iterate over all pushouts for $k = (1, \dots, n-1)$ and obtain a transformation $\varnothing' \overset{*}{\Rightarrow} PO^E_n = G_A$ via P and almost injective matches. By Lem. 3.15 with instance morphism $i \colon G_A \to G$ and all application conditions in P being in $\mathscr{M}$-normal form by assumption, there exists a transformation $p \colon \varnothing'' \overset{*}{\Rightarrow} G$ via P and almost injective matches with instance morphism $i' \colon \varnothing' \to \varnothing''$. By Def. 3.17, $i' \in \mathscr{E}, \mathscr{O}$, i.e., graph part i'_S is an isomorphism by Sec. 2.2.2 and Rem. 2.3, and furthermore, derived span $der(p) \colon \varnothing'' \to G \in \mathscr{M}$ by Sec. 2.2.4 and Rem. 2.15, i.e., data part $der(p)_D$ is an isomorphism by Sec. 2.2.2 and Rem. 2.3, and therefore, $\varnothing'' = \varnothing$ by general assumption where graphs $G \in \mathscr{L}(C)$ and start graph $\varnothing$ of GG share the same algebra up to isomorphism. Thus, there is a transformation $\varnothing \overset{*}{\Rightarrow} G$ via P and almost injective matches, i.e., $G \in \mathscr{L}(GG)$. Therefore, $\mathscr{L}(C) \subseteq \mathscr{L}(GG)$. □

A.12 Proof of Sec. 3.4 and Prop. 3.4

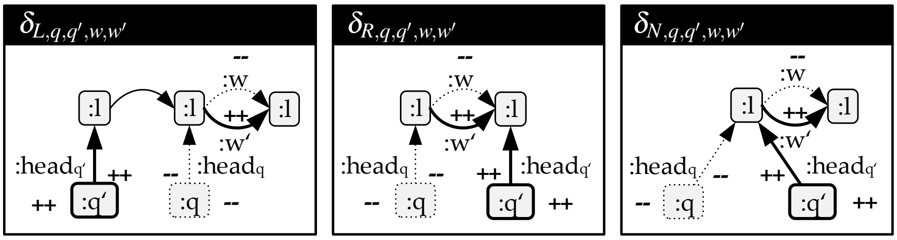
1. For partial reachability, we only have to check if there is $m \colon S \to G \in \mathscr{M}$, since, if there is $m' \colon G' \to G \in \mathscr{M}$ for some $t \colon S \overset{*}{\Rightarrow} G'$ then there is $m = m' \circ der(t) \colon S \to G \in \mathscr{M}$ by $\mathscr{M}$-composition and Rem. 2.15. Thus, $\nexists m \Rightarrow \nexists m'$.

2. The undecidability is shown by a reduction from an undecidable variant of the halting problem over turing machines (Does a deterministic turing machine (TM) with exactly one final/accepting state accept on a given input?). The reduction is given by a mapping from TMs and inputs to graph grammars and final graphs. Configurations are encoded as typed

graphs and each transition of the transition function δ is encoded as a graph transformation rule over typed graphs leading to a graph grammar with the initial configuration as start graph. The type graph is given below (middle) and defined by (a) a node q_i for each state $q_i \in Q, i = (0 \dots m)$ of the TM, (b) auxiliary node l which links together the tape symbols to a word as the tape content, (c) a reflexive edge w_j on node l for each symbol $w_j \in \Gamma, j = (0 \dots n)$ of the tape alphabet Γ, (d) an auxiliary edge head_{q_i} from node q_i to l for each q_i which marks the tape position together with the state for each configuration of TM, and (e) reflexive auxiliary egdes $\mathsf{L}_\Box$ and $\mathsf{R}_\Box$ on node l that mark the left and right ends of the tape. The start graph S for the initial configuration with input word $w_0 w_1 \dots w_n$ and initial state q_0 is given below (right).

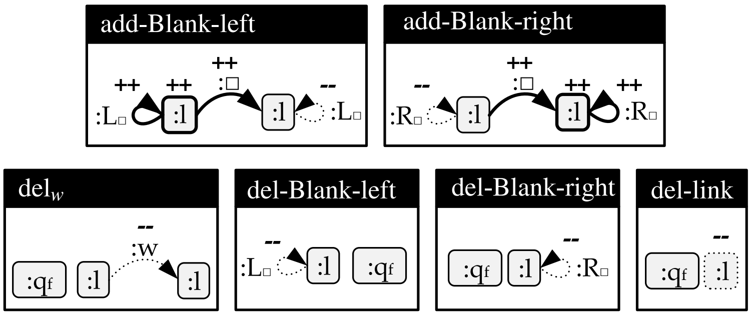


For each transition $(q,w) \to (q',w',\alpha) \in \delta$ with $\alpha \in \{L,N,R\}$, a $\delta_{\alpha,q,q',w,w'}$-rule is defined as below. Each δ-rule deletes symbol w,replaces it by w' and moves the head on the tape one step to the left for $\alpha = L$, to the right for $\alpha = R$ or performs no move for $\alpha = N$ by deleting the old head and state q and replacing them by the new state q' and head pointing to the new tape position.



Furthermore, the following auxiliary rules are defined: (a) Rules $\mathsf{add}-\mathsf{Blank}-\mathsf{left}$ and $\mathsf{add}-\mathsf{Blank}-\mathsf{right}$ that add a new blank symbol $\Box$ to the right and left ends of the tape, (b) for each symbol $w \in \Gamma$, a rule del_w that deletes the symbol from the tape if a final state q_f is reached, (c) rules $\mathsf{del}-\mathsf{Blank}-\mathsf{left}$ and $\mathsf{del}-\mathsf{Blank}-\mathsf{right}$ that delete the markers $L_\Box$ and $R_\Box$ for the left and right ends of the tape if a final state q_f is reached, and (d) rule $\mathsf{del}-\mathsf{link}$ which delets l nodes if a final state q_f is reached.

Therefore, a TM with input $w_0 w_1 \dots w_n$ is encoded as start graph S together with the set P of production rules from above forming graph grammar $GG = (S,P)$ and together with the final graph from above (left) where q_f is the final state. Moreover, P is finite by Q and Γ being finite for $\delta\colon (Q \setminus q_f) \times \Gamma \to Q \times \Gamma \times \{L,N,R\}$ and the final graph is finite.



Obviously, if the TM reaches the final state q_f, then the final graph is (partially) reachable from the start graph. This holds, since, (a) for each transition there exists a dedicated δ-rule simulating the transition, (b) new blank symbols can be added to the ends of the tape at any time by auxiliary rules del − Blank − left(right) simulating an infinite tape, and (c) auxiliary rules del... finally delete all nodes and edges except the final graph due to the gluing condition. Conversely, if the final graph is (partially) reachable from the start graph, then the TM reaches the final state q_f. This holds, since, each graph which is reachable from the start graph follows the structure of the start graph with exactly one state node and head edge, at most one L□ edge and at most one R□ edge at the ends of the tape and at most one symbol edge between each two l nodes forming a linear tape without branchings via edges. Therefore, there is at most one δ-rule applicable to each graph via at most one ($\mathscr{M}$-)match exactly representing the corresponding transition of the TM by the determinism of TM with δ being a function. Thus, each sequence of mixed δ- and add − Blank − left(right)-rule applications from the start graph to a graph containing (final) state node q_i represents a computation of TM from the initial configuration to a configuration with (final) state q_i. Finally, when assuming the decidability of the (partial) reachability problem, we could also decide the halting problem leading to a contradiction.

3. (a) There are at most $|M|!$ acyclic (terminating) match-paths in GS (that start in S) where each is composed of at most $|M|$ matches.

$$\begin{tikzcd}
L_p \arrow[dd, "n_1 \circ m"] \arrow[ddddd, bend right, "n_2 \circ m"'] & & K_p \arrow[ll, hook', "l_p"'] \arrow[rr, hook, "r_p"] \arrow[dd, "n_1''"] \arrow[ddd, dotted, bend left=10, "n''"] \arrow[ddddd, bend left=40, "n_2''"] & & R_p \arrow[dd, "n_1'"] \arrow[ddd, dotted, bend left=20, "n'"] \arrow[ddddd, bend left=50, "n_2'"] \\
& (1) & & (4) & \\
A_1 \arrow[ddd, hook, "i"] & (2) & O_1 \arrow[ll, hook', "l'_{p,1}"'] \arrow[rr, hook, "r'_{p,1}"] \arrow[d, dotted, "o"] & (5) & B_1 \arrow[d, dotted, "b"] \\
& & O \arrow[ull, dotted, hook', "l'"] \arrow[rr, dotted, hook, "r'"] \arrow[dd, dotted, "\underline{n''}"] & & B \arrow[dd, dotted, "\underline{n'}"] \\
& (3) & & (6) & \\
A_2 & & O_2 \arrow[ll, hook', "l'_{p,2}"'] \arrow[rr, hook, "r'_{p,2}"] & & B_2 \arrow[d, hook, "m''"] \\
& & & & G
\end{tikzcd}$$

Furthermore, given a match-path $path$, a graph G and two recursive transformations $A \xRightarrow{path}_{GS} B$ and $A \xRightarrow{path}_{GS} B'$, then $\exists m'\colon B \to G \in \mathscr{M}$ if and only if $\exists m''\colon B' \to G \in \mathscr{M}$ $^{(*^1)}$, i.e., $\nexists m'$ implies $\nexists m''$ for all other recursive transformations w.r.t. $path$. Therefore, for verifying all acyclic, terminating recursive transformations that start in S whether m exists, it is sufficient to verify only one recursive transformation for each match-path. By a finite set of acyclic, terminating match-paths in GS that start in

S, each path being finite and G being finite, the verification terminates. For showing $(*^1)$, we show it analogously for a single recursive transformation step which directly implies $(*^1)$ by induction over the recursive transformations. Thus, given a match m, a production $p = (L_p \xleftarrow{l_p} K_p \xrightarrow{r_p} R_p)$, a graph G and two recursive transformation steps $A_1 \xRightarrow{(p,m,n_1)}_{GS,n'_1} B_1$ and $A_2 \xRightarrow{(p,m,n_2)}_{GS,n'_2} B_2$ with $i\colon A_1 \rightarrow A_2 \in \mathscr{M}$ and $i \circ n_1 \circ m = n_2 \circ m$, if $\exists m''\colon B_2 \rightarrow G \in \mathscr{M}$, then $\exists m' = m'' \circ \underline{m'} \in \mathscr{M}$ with $\underline{m'}\colon B_1 \rightarrow B_2 \in \mathscr{M}$ and $\underline{m'} \circ n'_1 = n'_2$. By the restriction theorem with $i \in \mathscr{M}$ and $i \circ n_1 \circ m = n_2 \circ m$, there exists pushouts $(1)+(2), (3), (4)+(5)$ and (6) with $\underline{n''} \circ n'' = n''_2$ and $\underline{n'} \circ n' = n'_2$. By $(\mathbf{AGraphs}_{ATGI}, \mathscr{M})$ being $\mathscr{M}$-adhesive, uniqueness of pushout complements and (1) as well as $(1)+(2)$ being pushouts, there is induced isomorphism $o\colon O_1 \rightarrow O$ with $o \circ n''_1 = n''$. By pushout decomposition with $(1)+(2)+(3)$ and (1) being pushouts, $(2)+(3)$ is a pushout. By $\mathscr{M}$ being closed under pushouts, $\underline{n''} \circ o \in \mathscr{M}$. By (4) and $(4)+(5)$ being pushouts, there is induced morphism $b\colon B_1 \rightarrow B$ with $b \circ n'_1 = n'$ by universal pushout property. By pushout decomposition with $(4)+(5)+(6)$ and (4) being pushouts, $(5)+(6)$ is a pushout. Thus, $\underline{m'} = \underline{n'} \circ b \in \mathscr{M}$ with $\underline{m'} \circ n'_1 = n'_2$, since, $\mathscr{M}$ is closed under pushouts. Finally, $m'' \circ \underline{m'} \in \mathscr{M}$ by $\mathscr{M}$-composition.

(b) For cyclic, terminating recursive transformations that start in S we conclude as follows. By $(*^1)$, the verification of a single recursive transformation for each match-path again encloses the verification of all recursive transformations. The verification procedure of whether $m \in \mathscr{M}$ exists is as follows: Given $t\colon A \xRightarrow{path}_{GS,n'} B$ with $n'\colon R \rightarrow B \in \mathscr{M}$ - we start at S with the empty path *path* and $n' = id_S\colon S \rightarrow S$. Then, there are at most $(|M|^{|M|})^k \cdot |M|$ match-paths for some $k \in \mathbb{N}$ that may extend *path* by up to $|M| \cdot k + 1$ matches. For each possible *path* extension via matches $(m_i)_{i \in \{1 \ldots l \leq |M| \cdot k+1\}}$, t is extended match-wise via recursive transformation steps $s_i\colon (A_i \xRightarrow{(s_{GS}(m_i), m_i, n'_{i-1})}_{GS,n'_i} B_i)_{i \in \{1 \ldots l\}}$ with $n'_0 = n'$ and $A_0 = B$. If there is some m_i in an extension such that the corresponding step s_i does not exist due to a violation of the gluing condition, then we stop for this specific extension. If there is some $x \in n'_{i-1}(\_)$ that is preserved along $der(s_i)$ but $x \notin n'_i(\_)$ (i.e., B is strongly extended by element x), then we recursively proceed with the verification procedure for this specific extension with $t = t'$ where t' is t extended up to s_i and $n' = n'_i$. Note that x will never be touched (deleted) in any further extensions. Thus, by this monotonicity in recursive transformations we can stop for a specific extension if we reach a graph in that extension that is composed out of x's and is larger than G. If we have applied the last possible step $s_{|M| \cdot k+1}$ in an extension, then it is true that there is a match $m_i\colon L \rightarrow C \in M$ which was visited at least $k+1$ times in that extension via some step s_i with $k = |C|!$ and $|C|$ being the sum of graph elements (nodes and edges) in C. Furthermore, the graph which is obtained by s_i is isomorphic to each graph which was obtained by any of the previous steps via m_i, since, the graphs where not strongly extended by some element and we have already obtained all possible $k = |C|!$ combinations of mapping the elements of C to C. Thus, it is guaranteed that we repeat one of these combinations with $k+1$ and therefore, can stop for this specific extension. Therefore, the overall procedure terminates. □

A.13 Proof of Sec. 3.4 and Prop. 3.7

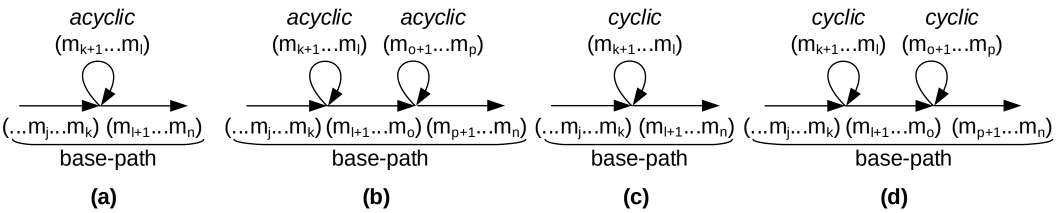
The base-path $b_{path'}$ of a match-path $path' = (m_1 \ldots m_n)$ and the partial mapping $\rightharpoonup_{path'} \colon \{1 \ldots n\} \to \{1 \ldots n\}$ of positions of matches from $path'$ to $b_{path'}$ are given by $b_{path'} := b(\varnothing, 0, 1, path')$ with

$$\text{for } j \leq n, b(m,i,j,path' = (m_1 \ldots m_i, m_{i+1} \ldots m_j \ldots m_k \ldots m_n)) :=$$
$$\begin{cases} b(m,i,j+1,path') & \text{if } i > 0 \text{ and } (m_{i+1} \ldots m_j \ldots m_k) \text{ is a match-cycle,} \\ b((m,m_j),j,j+1,path') & \text{otherwise with } \rightharpoonup_{path'} (j) := |m|+1 \end{cases}$$
$$\text{and for } j > n, b(m,i,j,path' = (m_1 \ldots m_n)) := m$$

Obviously, if $path'$ is acyclic, then $b_{path'} = path'$ $^{(*^1)}$. Moreover, if $path'$ is cyclic, terminating, starts in A or ends in B, then $b_{path'}$ is acyclic, terminating, starts in A or ends in B $^{(*^2)}$. We show for general terminating match-paths $path' = (m_1 \ldots m_n)$, sequences of recursive transformation steps $t' \colon (A_{i-1} \xRightarrow{(s_{GS}(m_i),m_i,n_{i-1})}_{GS,n_i} A_i)_{i \in \{j \ldots n\}}$ over $(m_j \ldots m_n)$ and morphisms $ac' \colon A \to A_{j-1}$ for $1 \leq j \leq n$ with $(j, \_) \in \rightharpoonup_{path'}$ that there is $(ac \colon A \to B, n) \in \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$ such that there is $i \colon B \to A_n \in \mathscr{M}$ with $i \circ ac = der(t') \circ ac'$ and $i \circ n = n_n$ $^{(*^3)}$. Moreover, if $(m_j \ldots .m_n)$ is acyclic, then $ac = der(t') \circ ac'$ and $n = n_n$ which does also hold for non-terminating $(m_1 \ldots m_j \ldots .m_n)$ $^{(*^4)}$.

Case $((m_j \ldots m_n)$ **is acyclic)** By assumption $(j, \_) \in \rightharpoonup_{path'}$, $b_{path'} = (m_1 \ldots m_{j'} \ldots m_{n'})$ with $n - j = n' - j'$, $(m_{j+i} = m_{j'+i})_{i \in \{0 \ldots n-j\}}$ and $(\rightharpoonup_{path'} (j+i) = j'+i)_{i \in \{0 \ldots n-j\}}$ $^{(*^5)}$, since by the construction of $b_{path'}$, assuming the opposite for some $j+i$ with $i > 0$ implies a match-cycle contradicting with assumption $(m_j \ldots m_n)$ being acyclic. By induction over t': **Basis.** Let $j = n$, i.e., t' be given by a single recursive transformation step $t' \colon A_{n-1} \xRightarrow{(s_{GS}(m_n),m_n,n_{n-1})}_{GS,n_n} A_n$. **Case** $(\mathscr{P} = \varnothing$ **or** $k = 2)$ By construction of $\underline{\mathscr{L}}_w$ and $(*^5)$ with $\rightharpoonup_{path'} (n) = n'$, $b_{path'} = (m_1 \ldots m_{n'})$ and $m_n = m_{n'}$, $(der(t') \circ ac', n_n) \in \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$. Finally, $i = id_{A_n} \in \mathscr{M}$ with $id_{A_n} \circ der(t') \circ ac' = der(t') \circ ac'$ and $id_{A_n} \circ n_n = n_n$. **Case** $(\mathscr{P} \neq \varnothing)$ By $(ac', n_{j-1}) \in \mathscr{C}$, it leads to the previous case with switch $k = 2$. **Hypothesis.** Assume that $(*^4)$ holds for t' consisting of $1 \leq k < n$ steps with $j = (n+1) - k$. **Step.** Let $j = (n+1) - (k+1) = n - k$, i.e., t' consists of $k+1$ steps. Analogously to the basis by $(*^5)$ with $m_{n-k} = m_j = m_{j'} = m_{\rightharpoonup_{path'}(j)} \in b_{path'}$, there is some invocation $\underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (n-k)+1, der(\bar{t}) \circ ac', n_{n-k}, \_) \overset{(*^5)}{=} \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} ((n+1)-k), der(\bar{t}) \circ ac', n_{n-k}, \_)$ from $\underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$ with $\bar{t} \colon A_{n-k-1} \xRightarrow{(s_{GS}(m_{n-k}),m_{n-k},n_{n-k-1})}_{GS,n_{n-k}} A_{n-k}$, $\bar{t}' \colon (A_{i-1} \xRightarrow{(s_{GS}(m_i),m_i,n_{i-1})}_{GS,n_i} A_i)_{i \in \{(n+1)-k \ldots n\}}$ and $der(\bar{t}') \circ der(\bar{t}) = der(t')$. By hypothesis and Prop. 3.2 together with acyclic $(m_j \ldots m_n)$ implying $(m_{j+1} \ldots m_n) = (m_{(n+1)-k} \ldots m_n)$ being acyclic, $(der(\bar{t}') \circ der(\bar{t}) \circ ac', n_n) = (der(t') \circ ac', n_n) \in \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} ((n+1)-k), der(\bar{t}) \circ ac', n_{n-k}, \_)$ and therefore, $(der(t') \circ ac', n_n) \in \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$. Finally, $i = id_{A_n} \in \mathscr{M}$ with $id_{A_n} \circ der(t') \circ ac' = der(t') \circ ac'$ and $id_{A_n} \circ n_n = n_n$.

Case $((m_j \ldots m_n)$ **is cyclic)**



By induction over the structure of t' based on Basis (a) and Steps (b)-(d) from above: Sub-paths $(m_{l+1} \dots m_n)$ or $(m_{p+1} \dots m_n)$ with $l, p < n$ in (a)-(d) are guaranteed to be in the base-path of $path'$, respectively, since, assuming they were part of a match-cycle contradicts with the assumption that $path'$ is terminating. Furthermore by assumption $(j, \_) \in \rightharpoonup_{path'}$, sub-paths $(m_j \dots m_k)$ with $j \leq k$ in (a)-(d) are guaranteed to be in the base-path of $path'$, respectively. Let $(m_j \dots m_n)$ be composed of sub-paths $(m_j \dots m_k), (m_{k+1} \dots m_l), (m_{l+1} \dots m_n)$ or $(m_j \dots m_k), (m_{k+1} \dots m_l), (m_{l+1} \dots m_o), (m_{o+1} \dots m_p), (m_{p+1} \dots m_n)$, respectively, with $(m_{k+1} \dots m_l) \in Paths_{t_{GS}(m_{l+1}), t_{GS}(m_{l+1})}(GS)$ and $(m_{o+1} \dots m_p) \in Paths_{t_{GS}(m_{p+1}), t_{GS}(m_{p+1})}(GS)$ being (a)cyclic match-cycles and $b_{path'} = (\dots m_j \dots m_k, m_{l+1} \dots m_n)$ or $b_{path'} = (\dots m_j \dots m_k, m_{l+1} \dots m_o, m_{p+1} \dots m_n)$ being the base-paths of $path'$. Thus, the sequence t' of steps is given by sub-sequences $t'_{(y,z)} : (A_{i-1} \xRightarrow{(s_{GS}(m_i), m_i, n_{i-1})}_{GS, n_i} A_i)_{i \in \{y \dots z\}}$ for $(y,z) \in \{(j,k), (k+1,l), (l+1,n)\}$ or $(y,z) \in \{(j,k), (k+1,l), (l+1,o), (o+1,p), (p+1,n)\}$, respectively. **Basis.** By Prop. 3.2 and $b_{path'}$ being acyclic by $(*^2)$, $(m_j \dots m_k, m_{l+1} \dots m_n)$ and therefore, $(m_j \dots m_k)$ are acyclic and furthermore, $b_{path'} \overset{(*^1)}{=} b_{b_{path'}}$. Thus, by $(*^4)$, $(der(t'_{(j,k)}) \circ ac', n_k) \in \underline{\mathscr{L}}_w((\dots m_j \dots m_k), \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$. Therefore, by the match-wise construction of $\underline{\mathscr{L}}_w$, there is an invocation $\underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (l+1), der(t'_{(j,k)}) \circ ac', n_k, \_)$ from $\underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$. Assume that match-cycle $m = (m_{k+1} \dots m_l) \in \gg_{GS} (b_{path'})$, then $m_n \in m$ by the definition of $\gg_{GS}$ and therefore, $b_{path'}$ is not terminating contradicting with the direct implication of assumption $path'$ being terminating via $(*^2)$. Thus, from $m \notin \gg_{GS} (b_{path'})$ and m being acyclic by assumption, we conclude that $m \in \mathscr{P} \subseteq Paths_{t_{GS}(m_{\rightharpoonup_{path'}(l+1)}), t_{GS}(m_{\rightharpoonup_{path'}(l+1)})}(GS) \setminus \gg_{GS} (b_{path'})$. We inductively conclude over sequence $t'_{(k+1,l)}$ as follows:

$$\begin{tikzcd}
 & & L_{k+2} \arrow[rrr, hook] \arrow[d, hook, "m_{k+2}"'] \arrow[drr, phantom, "(3)"] & & & R_{k+2} \arrow[dl, hook, dotted, "l_{k+2}"'] \arrow[ddd, hook, "n_{k+2}"] \\
L_{k+1} \arrow[rr, hook] \arrow[d, hook, "m_{k+1}"'] \arrow[dr, phantom, "(1)"] & & R_{k+1} \arrow[rr, hook, dotted] \arrow[dl, hook, dotted, "r_{k+1}"'] \arrow[dd, hook, "n_{k+1}"] \arrow[dr, phantom, "(4)"] & & L'_{k+2} \arrow[dl, hook, dotted, "r_{k+2}"'] & \\
R_k \arrow[r, hook, dotted, "g_{k+1}"] \arrow[d, hook, "n_k"'] \arrow[dr, phantom, "(2)"] & R'_{k+1} \arrow[rr, hook, dotted, "g_{k+2}"] \arrow[dr, hook, dotted, "a_{k+1}"'] \arrow[drr, phantom, "(5)"] & & R'_{k+2} \arrow[drr, hook, dotted, "a_{k+2}"'] & & \\
A_k \arrow[rr, hook] & & A_{k+1} \arrow[rrr, hook] & & & A_{k+2}
\end{tikzcd}$$

$$\begin{tikzcd}
 & & L_i \arrow[rrr, hook] \arrow[d, hook, "m_i"'] & & & R_i \arrow[ddl, hook, dotted, "l_i"'] \arrow[dddd, hook, "n_i"] \\
 & & R_{i-1} \arrow[dl, hook, "l_{i-1}"'] \arrow[ddd, hook, "n_{i-1}"] & (6_i) & & \\
 & L'_{i-1} \arrow[rrr, hook, dotted] \arrow[dl, hook, "r_{i-1}"'] \arrow[drr, phantom, "(7_i)"] & & & L'_i \arrow[dl, hook, dotted, "r_i"] & \\
R'_{i-1} \arrow[rrr, hook, dotted, "g_i"] \arrow[drr, hook, "a_{i-1}"'] & & & R'_i \arrow[drr, hook, dotted, "a_i"] & & \\
 & & A_{i-1} \arrow[rrr, hook] \arrow[ur, phantom, "(8_i)"] & & & A_i
\end{tikzcd}$$

All given morphisms from above are in $\mathscr{M}$ by the definition of non-deleting productions and recursive transformation steps. Therefore, we can construct pushouts over them resulting again in $\mathscr{M}$-morphisms only, since, $\mathscr{M}$ is closed under pushouts. In particular, we construct pushout (1) with induced morphism a_{k+1} and $a_{k+1} \circ r_{k+1} = n_{k+1}$. By pushout decomposition with $(1) + (2)$ and (1) being pushouts, (2) is a pushout. We construct pushouts (3) and (4) resulting in pushout $(3) + (4)$ by pushout composition with induced morphism a_{k+2} and $a_{k+2} \circ r_{k+2} \circ l_{k+2} = n_{k+2}$. Again, by pushout decomposition with $(3)+(4)+(5)$ and $(3)+(4)$ being pushouts, (5) is a pushout. Analogously, for each subsequent step i with $k+2<i\leq l$ in $t'_{(k+1,l)}$, we construct pushouts (6_i) over $l_{i-1}\circ m_i\in\mathscr{M}$ by $\mathscr{M}$-composition and (7_i) resulting again in pushouts $(6_i)+(7_i)$ and (8_i) with induced morphism a_i and $a_i\circ r_i\circ l_i=n_i$. Thus, by $(*^4)$ for $m\overset{(*^1)}{=}b_m$, $(g_l\circ\ldots\circ g_{k+1},r_l\circ l_l)\in B_m=\underline{\mathscr{L}}_w(m,1,id_{R_k},id_{R_k},2)$ in Def. 3.28 and Item 2a. Furthermore, for $t'_{(k+1,l)}$ by pushout composition, we obtain pushouts $(1)+(4)+(7_{k+3})+\ldots+(7_l)$ and $(2)+(5)+(8_{k+3})+\ldots+(8_l)$ and the composition of both. Therefore, the pushout complement exists in Def. 3.28 and Item 2c with induced morphism $l_l\in\mathscr{M}$. Since, the pushout complement and object in Def. 3.28 and Item 2b are unique up to isomorphism, w.l.o.g. we conclude that $(der(t'_{(k+1,l)})\circ der(t'_{(j,k)})\circ ac',n_l)\in A_k\oplus B_m\subseteq\mathscr{C}$ for the trivial $i_{L_{k+1}}$-$\mathscr{M}$-decomposition $d=(I_{L_{k+1}}\xrightarrow{i_{L_{k+1}}}L_{k+1}\xrightarrow{id_{L_{k+1}}}L_{k+1})\in\mathscr{D}$ in Def. 3.28 and Item 1 with $m=m_{k+1}$ in Def. 3.28. Note that for $der(t)=n'=id_{R_k}$ in Def. 3.28 and Item 2d, Def. 3.28 and Item 2(d)i holds, since, o' is an isomorphism by preservation of isomorphism $id_{L_{k+1}}$ along the constructed pushout (o,o') with inverse isomorphism $a''=o'^{-1}$ - Def. 3.28 and Items 2(d)ii and 2(d)iii follow directly from a'' being an isomorphism. Therefore, there is an invocation $\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(l+1),der(t'_{(k+1,l)})\circ der(t'_{(j,k)})\circ ac',n_l,2)$ from $\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(j),ac',n_{j-1},\_)$. By Prop. 3.2 and $b_{path'}$ being acyclic by $(*^2)$, $(m_{l+1}\ldots m_n)$ is acyclic. Therefore by $(*^4)$ with $b_{b_{path'}}\overset{(*^1)}{=}b_{path'}$, $(der(t'_{(l+1,n)})\circ der(t'_{(k+1,l)})\circ der(t'_{(j,k)})\circ ac',n_n)=(der(t')\circ ac',n_n)\in\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(l+1),der(t'_{(k+1,l)})\circ der(t'_{(j,k)})\circ ac',n_l,2)$. Thus, $(der(t')\circ ac',n_n)\in\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(j),ac',n_{j-1},\_)$. **Hypothesis.** Assume that $(*^4)$ and therefore $(*^3)$ with i being the corresponding identity morphism hold for case (a). **Step.** Analogously to the base case, for case (b), there is an invocation $\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(l+1),der(t'_{(k+1,l)})\circ der(t'_{(j,k)})\circ ac',n_l,2)$ from $\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(j),ac',n_{j-1},\_)$. By induction hypothesis, $(der(t')\circ ac',n_n)\in\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(l+1),der(t'_{(k+1,l)})\circ der(t'_{(j,k)})\circ ac',n_l,2)$ and therefore, $(der(t')\circ ac',n_n)\in\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(j),ac',n_{j-1},\_)$. Analogously, we inductively proceed with arbitrary acyclic match-cycles that are reachable from $b_{paths'}$. For case (c) with cyclic match-path $(m_{k+1}\ldots m_l)$, let $(m_{l'}\ldots m_l)$ in $(m_{k+1}\ldots m_{l'}\ldots m_l)$ with $t_{GS}(m_{l'})=t_{GS}(m_{k+1})$ and $\forall i\in\{l'+1\ldots l\}.t_{GS}(m_i)\neq t_{GS}(m_{l'})$ $^{(*^6)}$ be the last match-cycle in $(m_{k+1}\ldots m_l)$ that starts and ends in $t_{GS}(m_{l+1})$. Analogously to the base case, there is an invocation $\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(l+1),der(t'_{(j,k)})\circ ac',n_k,\_)$ from $\underline{\mathscr{L}}_w(b_{path'},\rightharpoonup_{path'}(j),ac',n_{j-1},\_)$. Based on the construction in Def. 3.28 we conclude as follows:

$$\begin{tikzcd}
 & & & L_{l'-1} \arrow[r, hook] \arrow[d, hook, "m_{l'-1}"'] & R_{l'-1}=R_k \arrow[dd, hook, "n'=r_{l'-1}\circ l_{l'-1}"'] \arrow[ddd, bend left, hook, "n_{l'-1}"] \\
L_{k+1} \arrow[r, hook] \arrow[d, hook, "m_{k+1}"'] & R_{k+1} \arrow[d, hook] & \cdots & R_{l'-2} \arrow[d, hook] & \\
der(t):R_k \arrow[r, hook, "g_{k+1}"] \arrow[d, hook, "n_k"'] \arrow[dr, phantom, "(k+1)"] & \circ \arrow[d, hook] & & \circ \arrow[r, hook, "g_{l'-1}"] \arrow[d, hook] \arrow[dr, phantom, "(l'-1)"] & R'_{l'-1}=R'' \arrow[d, hook, "a_{l'-1}"] \\
A_k \arrow[r, hook] & A_{k+1} & \cdots & A_{l'-2} \arrow[r, hook] & A_{l'-1} \\
 & & der(t'_{(k+1,l'-1)}) & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & L_l \arrow[r, hook] \arrow[d, hook, "l_{l-1}\circ m_l"'] & R_l=R_k \arrow[d, hook, "l_l"] \arrow[ddd, bend left, hook, "n_l"] \\
L_{l'} \arrow[r, hook, "b_{l'}"] \arrow[d, hook, "m_{l'}"'] \arrow[dr, phantom, "(l'_1)"] & R_{l'} \arrow[d, hook] & \cdots & \circ \arrow[r, hook, "b_l"] \arrow[d, hook] \arrow[dr, phantom, "(l_1)"] & \circ \arrow[d, hook, "r_l"] \\
R_{l'-1} \arrow[r, hook, "g_{l'}" description] \arrow[d, hook, "n_{l'-1}"'] \arrow[dr, phantom, "(l'_2)"] & \circ \arrow[d, hook] & & \circ \arrow[r, hook, "g_l" description] \arrow[d, hook] \arrow[dr, phantom, "(l_2)"] & \circ \arrow[d, hook, "a_l"] \\
A_{l'-1} \arrow[r, hook, "a'_{l'}" description] & A_{l'} & \cdots & A_{l-1} \arrow[r, hook, "a'_l" description] & A_l \\
 & & der(t'_{(l',l)}) & &
\end{tikzcd}$$

$$\begin{tikzcd}
I_{L_{l'}} \arrow[r, hook, "i_{L_{l'},1}"] & I'_{L_{l'}} \arrow[r, hook, "i_{L_{l'},2}"] \arrow[dd, hook, "m"] \arrow[ddr, phantom, "(PB)"] & L_{l'} \arrow[r, hook, "\overline{ac}'"] \arrow[d, hook, "m_{l'}"] & \circ \arrow[ddd, hook, "c \circ \overline{n}_2"] \arrow[dddd, hook, dotted, bend left, "a'_1"] \\
 & & R_k \arrow[d, hook, "n'"] \arrow[dd, bend left, "n_{l'-1}"] \arrow[r, phantom, "(1)+(2)"] & \\
 & der(t): R_k \arrow[r, hook, "g_{l'-1} \circ \ldots \circ g_{k+1}"] \arrow[d, hook, "n_k"] \arrow[dr, phantom, "(k+1)+\ldots+(l'-1)"] & R'' \arrow[d, hook, "a_{l'-1}"] & \\
 & A_k \arrow[r, hook, "der(t'_{(k+1,l'-1)})"] \arrow[drr, hook, dotted, bend right, "a'_2"] & A_{l'-1} \arrow[r, hook, "d"] & C \\
 & & & \circ \arrow[u, hook, dotted, "e_1"]
\end{tikzcd}$$

$$\begin{tikzcd}
L_{l'} \arrow[r, hook, "\overline{ac}'"] \arrow[rrr, hook, bend left, "b_l \circ \ldots \circ b_{l'}"] \arrow[d, hook, "m_{l'}"] \arrow[dr, phantom, "(1)"] & \circ \arrow[d, hook, "\overline{n}_2"] & R_l \arrow[l, hook, "\overline{n}_1"'] \arrow[r, hook, "l_l"] & \circ \arrow[d, hook, "r_l"] \\
R_{l'-1} \arrow[r, hook, "\overline{ac}"] \arrow[rrr, hook, bend left, "g_l \circ \ldots \circ g_{l'}"] \arrow[d, hook, "n_{l'-1}"] \arrow[dr, phantom, "(2)"] & \circ \arrow[rr, hook, dotted, "\overline{i}"] \arrow[d, hook, "c"] \arrow[drr, phantom, "(3)"] & & \circ \arrow[d, hook, "a_l"] \arrow[dl, hook, dotted, "c'"] \\
A_{l'-1} \arrow[r, hook, "d"] \arrow[rrr, hook, bend left, "a'_l \circ \ldots \circ a'_{l'}"] & C \arrow[r, hook, dotted, "\overline{i}_1"] & C' \arrow[r, hook, dotted, "\overline{i}_2"] & A_l
\end{tikzcd}$$

Analogously to the base case by assumption $path'$ being terminating, $(m_{k+1} \ldots m_{l'-1}) \in \overline{\mathscr{P}} = Paths_{t_{GS}(m_{\rightharpoonup_{path'}(l+1)}),t_{GS}(m_{\rightharpoonup_{path'}(l+1)})}(GS) \setminus \gg_{GS} (b_{path'})$ and furthermore, by $(*^1),(*^2)$ with $b_{(m_{l'} \ldots m_l)}$ being acyclic, $b_{(m_{l'} \ldots m_l)} \in \mathscr{P} \subseteq \overline{\mathscr{P}}$. Thus, for $A_k \oplus B_{b_{(m_{l'} \ldots m_l)}}$ we conclude as follows. Although $(m_{l'} \ldots m_l)$ is not terminating, we can apply the induction hypothesis and steps to it, since, from $(m_{l'} \ldots m_l)$ being the last cycle in $(m_{k+1} \ldots m_{l'} \ldots m_l)$ from and to $t_{GS}(m_{l+1})$ it follows that (a) sub-path $(\ldots m_l)$ is in the base-path $b_{(m_{l'} \ldots m_l)}$ of $(m_{l'} \ldots m_l)$, (b) for all cycles c in $(m_{l'} \ldots m_l)$ that are reachable from $b_{(m_{l'} \ldots m_l)}$ it is true that $c \not\in \gg_{GS} (b_{(m_{l'} \ldots m_l)})$ and furthermore, (c) sub-path $(m_{l'} \ldots)$ is in the base-path $b_{(m_{l'} \ldots m_l)}$ by definition (i.e., $b_{(m_{l'} \ldots m_l)} = (m_{l'}, \ldots, m_l)$). For (a) : Assume that $(\ldots m_l)$ is not in $b_{(m_{l'} \ldots m_l)}$, then it is part of a match-cycle $(m_{l''} \ldots m_l)$ in $(m_{l'} \ldots m_{l''} \ldots m_l)$ with $l'' > l'$ and $t_{GS}(m_{l''}) = t_{GS}(m_{l'})$ which contradicts with assumption $(*^6)$. For (b) : Assume that $c \in \gg_{GS} (b_{(m_{l'} \ldots m_l)})$, then by (a), $m_l \in c$ by definition and therefore, $\exists m_{l''}$ in $(m_{l'} \ldots m_{l''} \ldots m_l)$ with $l'' > l'$ and $t_{GS}(m_{l''}) = t_{GS}(m_{l'})$ by the definition of match-paths which contradicts with assumption $(*^6)$. Moreover, analogously to the base case via pushout (de)-compositions for sequence $t'_{(k+1,l'-1)}$, there is a recursive transformation t w.r.t. $(m_{k+1} \ldots m_{l'-1})$ from R_k to $R'_{l'-1} = R''$ with derived span $der(t) = g_{l'-1} \circ \ldots \circ g_{k+1}$ and co-match $n' = r_{l'-1} \circ l_{l'-1} : R_{l'-1} = R_k \to R'_{l'-1}$. Furthermore, $(k+1) + \ldots + (l'-1)$ (is a) are pushout(s) and $a_{l'-1} \circ n' = n_{l'-1}$. This also holds for sequence $t'_{(l',l)}$ with $(l'_1) + \ldots + (l_1)$ and $(l'_2) + \ldots + (l_2)$ being pushouts and $a_l \circ r_l \circ l_l = n_l$. By induction hypothesis and steps for $(m_{l'} \ldots m_l)$, there is $(\overline{ac}, \overline{n}_2 \circ \overline{n}_1) \in \underline{\mathscr{L}}_w(b_{(m_{l'} \ldots m_l)}, 1, id_{R_k}, id_{R_k}, 2) = B_{b_{(m_{l'} \ldots m_l)}}$ in Def. 3.28 and Item 2a with $\overline{i} \in \mathscr{M}$ and $\overline{i} \circ \overline{ac} = g_l \circ \ldots \circ g_{l'} \circ id_{R_k} = g_l \circ \ldots \circ g_{l'}$ as well as $\overline{i} \circ \overline{n}_2 \circ \overline{n}_1 = r_l \circ l_l$ and $\overline{ac} \in \mathscr{M}$ by $\mathscr{M}$-decomposition with $g_l \circ \ldots \circ g_{l'} \in \mathscr{M}$ by $\mathscr{M}$-composition $^{(*^7)}$. By construction in Def. 3.28 and pushout (de)-composition, again we obtain pushout (1) with pushout complement $(\overline{ac}' \in \mathscr{M}, \overline{n}_2 \in \mathscr{M})$ and induced morphism $\overline{n}_1 \in \mathscr{M}$ in Def. 3.28 and Item 2c and construct pushouts (2) and (3) with $\overline{i}_1 \in \mathscr{M}$ by $\mathscr{M}$-preservation along (3) and $(2)+(3)$ being a pushout by composition. From $(l'_2) + \ldots + (l_2)$ being a pushout and $(*^7)$, $(\overline{i} \circ \overline{ac}, n_{l'-1})$ over $(a'_l \circ \ldots \circ a'_{l'}, a_l)$ is also a pullback. Thus by effective pushouts with $(2)+(3)$ being a pushout, there is $\overline{i}_2 \in \mathscr{M}$ with $\overline{i}_2 \circ c' = a_l$ and $\overline{i}_2 \circ \overline{i}_1 \circ d = a'_l \circ \ldots \circ a'_{l'} = der(t'_{(l',l)})$ $^{(*^8)}$. Therefore, $\overline{i}_2 \circ \overline{i}_1 \circ c \circ \overline{n}_2 \circ \overline{n}_1 \overset{(3),(*^8)}{=} a_l \circ \overline{i} \circ \overline{n}_2 \circ \overline{n}_1 \overset{(*^7)}{=} a_l \circ r_l \circ l_l = n_l$ $^{(*^9)}$. By pushout composition $(1)+(2)$ is a pushout with $n_{l'-1} \circ m_{l'} \in \mathscr{M}$ by $\mathscr{M}$-composition and therefore, $(1)+(2)$ is also a pullback. Analogously, pushout $(k+1) + \ldots + (l'-1)$ with $n_k \in \mathscr{M}$ is also a pullback. We construct pullback (PB) over $(n' \circ m_{l'}, g_{l'-1} \circ \ldots \circ g_{k+1})$ with $i_{L_{l'},2}, m \in \mathscr{M}$, since, $\mathscr{M}$ is closed under (PB). Let $[i_{L_{l'},1} \in \mathscr{M}]$ be the initial $\mathscr{M}$-subobject of $I'_{L_{l'}}$. By Prop. 3.6, $[i_{L_{l'},2} \circ i_{L_{l'},1} \in \mathscr{D}]$ is the initial $\mathscr{M}$-subobject of $L_{l'}$ and its decomposition in Def. 3.28 and Item 1. Furthermore for Def. 3.28 and Item 2d, we construct pushout (o, o')

over $(i_{L_{l'},2}, m)$ with induced morphism $a'' \in \mathscr{M}$ by effective pushout (cf. Item 2(d)i)and $a'' \circ o = n' \circ m_{l'}$ (cf. Item 2(d)ii) as well as $a'' \circ o' = der(t)$ (cf. Item 2(d)iii). Finally, we construct pushout (a'_1, a'_2) over $(\overline{ac}' \circ i_{L_{l'},2}, n_k \circ m)$ in Def. 3.28 and Item 2b with $(a'_2 \circ der(t'_{(j,k)}) \circ ac', a'_1 \circ \overline{n}_1) \in A_k \oplus B_{b_{(m_{l'} \ldots m_l)}} \subseteq \mathscr{C}$. Thus, there is an invocation $\underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (l+1), a'_2 \circ der(t'_{(j,k)}) \circ ac', a'_1 \circ \overline{n}_1, 2)$ from $\underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$. Moreover by pullback composition, $(PB) + (k+1) + \ldots + (l'-1) + (1) + (2)$ is a pullback and therefore by effective pushouts, there is $e_1 \in \mathscr{M}$ with $e_1 \circ a'_1 = c \circ \overline{n}_2$ and $e_1 \circ a'_2 = d \circ der(t'_{(k+1,l'-1)})$ $^{(*^{10})}$. Thus by $\mathscr{M}$-composition, there is $i' = \bar{i}_2 \circ \bar{i}_1 \circ e_1 \in \mathscr{M}$ with $i' \circ a'_2 \circ der(t'_{(j,k)}) \circ ac' \overset{(*^{10})}{=} \bar{i}_2 \circ \bar{i}_1 \circ d \circ der(t'_{(k+1,l'-1)}) \circ der(t'_{(j,k)}) \circ ac' \overset{(*^{8})}{=} der(t'_{(l',l)}) \circ der(t'_{(k+1,l'-1)}) \circ der(t'_{(j,k)}) \circ ac' = der(t'_{(k+1,l)}) \circ der(t'_{(j,k)}) \circ ac'$ and $i' \circ a'_1 \circ \overline{n}_1 \overset{(*^{10})}{=} \bar{i}_2 \circ \bar{i}_1 \circ c \circ \overline{n}_2 \circ \overline{n}_1 \overset{(*^{7}),(3)}{=} \bar{i}_2 \circ c' \circ r_l \circ l_l \overset{(*^{8})}{=} a_l \circ r_l \circ l_l = n_l$ $^{(*^{11})}$. With the remaining sequence over match-path (m_{l+1}, m_n) we proceed as follows.

$$\begin{tikzcd}
\circ \arrow[rrr, dotted, hook, bend left, "i_{i,2}"] \arrow[from=1-2, to=1-3, phantom, "(=)"] & \circ \arrow[l, dotted, hook', "i_{i,1}"'] & R_i \arrow[l, dotted, hook', "n'_i"'] \arrow[r, hook, "n_i"] & A_i \\
 & & L_i \arrow[u, hook] \arrow[d, hook, "m_i"] & \\
A_{i-1} \arrow[uu, dotted, hook, "a''_i"] \arrow[from=1-1, to=3-2, phantom, "(3)"] & \circ \arrow[l, hook', "i_{i-1}"'] \arrow[uu, dotted, hook, "a'_i"] \arrow[rr, hook, bend right, "i_{i-1}"'] \arrow[from=1-2, to=3-3, phantom, "(1)"] & R_{i-1} \arrow[l, hook', "n'_{i-1}"'] \arrow[r, hook, "n_{i-1}"] \arrow[from=1-3, to=3-4, phantom, "(2)"] \arrow[d, phantom, "(=)"] & A_{i-1} \arrow[uu, hook, "a_i"'] \\
 & & &
\end{tikzcd}$$

Beside the existing sequence $t'_{(l+1,n)}$ of steps with pushouts (2) for $i \in \{l+1 \ldots n\}$ and $der(t'_{(l+1,n)}) = a_n \circ \ldots a_{l+1}$, we construct a corresponding sequence $t''_{(l+1,n)}$ over $(m_{l+1} \ldots m_n)$ with pushouts (1), $der(t''_{(l+1,n)}) = a'_n \circ \ldots \circ a'_{l+1}$ and $n'_l = a'_1 \circ \overline{n}_1$, $i_l = i'$. For each step i we construct pushout (3) over $i_{i-1} \in \mathscr{M}$ with $i_{i,1} \in \mathscr{M}$ and pushout $(1) + (3)$ by pushout composition and preservation of $\mathscr{M}$ along (3). By the definition of recursive transformation steps and $\mathscr{M}$-composition with $n_{i-1} \circ m_i \in \mathscr{M}$, (2) is also a pullback and therefore by effective pushouts with pushout $(1) + (3)$ and $(*^{11})$, there is $i_{i,2} \in \mathscr{M}$ with $i_i = i_{i,2} \circ i_{i,1} \in \mathscr{M}$ by $\mathscr{M}$-composition, $i_i \circ n'_i = n_i$ and $i_{i,2} \circ a''_i = a_i \Rightarrow i_{i,2} \circ a''_i \circ i_{i-1} = a_i \circ i_{i-1} \overset{(3)}{\Leftrightarrow} i_i \circ a'_i = a_i \circ i_{i-1}$. Thus, $i_n \circ n'_n = n_n$ and $i_n \circ der(t''_{(l+1,n)}) = i_n \circ a'_n \circ \ldots \circ a'_{l+1} = a_n \circ \ldots \circ a_{l+1} \circ i_l = der(t'_{(l+1,n)}) \circ i_l$ $^{(*^{12})}$. Analogously to the base case, $(m_{l+1} \ldots m_n)$ is acyclic and therefore by $(*^4)$, $(der(t''_{(l+1,n)}) \circ a'_2 \circ der(t'_{(j,k)}) \circ ac', n'_n) \in \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (l+1), a'_2 \circ der(t'_{(j,k)}) \circ ac', a'_1 \circ \overline{n}_1, 2)$, i.e., $(der(t''_{(l+1,n)}) \circ a'_2 \circ der(t'_{(j,k)}) \circ ac', n'_n) \in \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$. Furthermore, there is $i_n \in \mathscr{M}$ with $der(t') \circ ac' = der(t'_{(l+1,n)}) \circ der(t'_{(k+1,l)}) \circ der(t'_{(j,k)}) \circ ac' \overset{(*^{11})}{=} der(t'_{(l+1,n)}) \circ i' \circ a'_2 \circ der(t'_{(j,k)}) \circ ac' \overset{(*^{12})}{=} i_n \circ der(t''_{(l+1,n)}) \circ a'_2 \circ der(t'_{(j,k)}) \circ ac'$ and $i_n \circ n'_n \overset{(*^{12})}{=} n_n$. Analogously to case (c), for case (d), there is an invocation $\underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (l+1), ac_1, n_1, 2)$ from $\underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$ with $i_1 \in \mathscr{M}$ and $i_1 \circ ac_1 = der(t'_{(k+1,l)}) \circ der(t'_{(j,k)}) \circ ac'$ as well as $i_1 \circ n_1 = n_l$ $^{(*^{13})}$. Furthermore analogously to (c), we construct a sequence $t''_{(l+1,n)}$ over $(m_{l+1} \ldots m_n)$ and n_1 with co-match n'_n, morphism $i_n \in \mathscr{M}$, $i_n \circ n'_n = n_n$

and $i_n \circ der(t''_{(l+1,n)}) = der(t'_{(l+1,n)}) \circ i_1$ $^{(*^{14})}$. By $(*^3)$ for case (c), there is $(ac_2,n_2) \in \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (l+1), ac_1, n_1, 2)$ with $i_2 \in \mathscr{M}$, $i_2 \circ ac_2 = der(t''_{(l+1,n)}) \circ ac_1$ and $i_2 \circ n_2 = n'_n$ $^{(*^{15})}$. Thus, there is $(ac_2,n_2) \in \underline{\mathscr{L}}_w(b_{path'}, \rightharpoonup_{path'} (j), ac', n_{j-1}, \_)$ with $i_n \circ i_2 \in \mathscr{M}$ by $\mathscr{M}$-composition, $i_n \circ i_2 \circ ac_2 \overset{(*^{15})}{=} i_n \circ der(t''_{(l+1,n)}) \circ ac_1 \overset{(*^{14})}{=} der(t'_{(l+1,n)}) \circ i_1 \circ ac_1 \overset{(*^{13})}{=} der(t'_{(l+1,n)}) \circ der(t'_{(k+1,l)}) \circ der(t'_{(j,k)}) \circ ac' = der(t') \circ ac'$ and $i_n \circ i_2 \circ n_2 \overset{(*^{15})}{=} i_n \circ n'_n \overset{(*^{14})}{=} n_n$. Analogously, we inductively proceed with arbitrary cyclic match-cycles that are reachable from $b_{path'}$.

1. Let $t\colon S \overset{path}{\Longrightarrow}_{GS} G$ be terminating, acyclic and starting in S. By Def. 3.20, $path$ is terminating, acyclic and starts in S, therefore, $b_{path} \overset{(*^1)}{=} path \in \underline{Paths}_S(GS)$. Thus, by the construction of $\mathscr{L}_w(GS)$ and $(*^4)$, $[der(t)] \in \mathscr{L}_w(GS)$ for $ac' = n_0 = id_S$.

2. Let $t\colon S \overset{path}{\Longrightarrow}_{GS} G$ be terminating, cyclic and starting in S. By Def. 3.20, $path$ is terminating, cyclic and starts in S. Thus by $(*^2)$, $b_{path} \in \underline{Paths}_S(GS)$ and furthermore, $(*^3)$ implies that there is $[ac\colon S \to G'] \in \mathscr{L}_w(GS)$ such that there is $i\colon G' \to G \in \mathscr{M}$ with $i \circ ac = der(t)$ for $ac' = n_0 = id_S$.

3. By induction over the construction of $\underline{\mathscr{L}}_w$ in Def. 3.28: **Basis.** Let $(ac,n) \in \underline{\mathscr{L}}_w(m = (m_1\colon L \to R, \ldots, m_n), 1, id_R, id_R, \_)$ be obtained without recursive calls of $\underline{\mathscr{L}}_w$ for $\mathscr{C}$. Then, by the construction in Def. 3.28, there is $t\colon R \overset{m}{\Longrightarrow}_{GS,n} A$ with $ac = der(t)$. Thus, there is identity $id_A \in \mathscr{M}$ with $id_A \circ ac = ac = der(t)$ and $id_A \circ n = n$. **Hypothesis.** For $(ac,n') \in \underline{\mathscr{L}}_w(m = (m_1\colon L \to R, \ldots, m_n), 1, id_R, id_R, \_)$ there is $t\colon R \overset{m}{\Longrightarrow}_{GS,n} A$ and $i \in \mathscr{M}$ with $i \circ ac = der(t)$ and $i \circ n' = n$. **Step.** Let $(ac,n) \in \underline{\mathscr{L}}_w(m = (m_1\colon L \to R \ldots m_j \ldots m_n), 1, id_R, id_R, \_)$ be obtained by recursive calls $\underline{\mathscr{L}}_w(m_i, 1, id_{R_i}, id_{R_i}, 2)$ for $\mathscr{C}$ starting at position j for i in m with $1 < j \leq n$. Note that there does not exist such recursive calls at position 1, since, either $\underline{\mathscr{L}}_w$ is called with $R = S$ (i.e., $\mathscr{P} = \varnothing$) or $\underline{\mathscr{L}}_w$ is called with switch $k = 2$ for position 1 $^{(A)}$. By the construction of $\underline{\mathscr{L}}_w$ in Def. 3.28, there is $t_1\colon R \xRightarrow{(m_1 \ldots m_{j-1})}_{GS,n_{j-1}} A$ and invocation $\underline{\mathscr{L}}_w(m, j, ac_1 = der(t_1), n_1 = n_{j-1}, \_)$ with $i'_1 = id_A \in \mathscr{M}$ and $i'_1 \circ ac_1 = der(t_1)$ as well as $i'_1 \circ n_1 = n_{j-1}$ $^{(*^1)}$. Furthermore for recursive calls for $\mathscr{C}$ by construction with pushouts (o,o') over $(i_{L,2},m)$ (cf. Def. 3.28 and Item 2d) and (a'_1,a'_2) over $(\overline{ac}' \circ i_{L,2}, n_1 \circ id_{R_i} \circ m)$ (cf. Def. 3.28 and Item 2b) we conclude as follows (all morphisms given below are in $\mathscr{M}$ by definition and preservation of $\mathscr{M}$ along pushouts and $\mathscr{M}$-composition):

$$\begin{tikzcd}
 & \circ \arrow[d, hook, "\overline{m}'_1 \circ \overline{n}'"'] \arrow[rd, hook, "n''"] \arrow[d, phantom, shift left=6ex, "(=)"] & & \\
R_i \arrow[r, hook, "\overline{ac}"] \arrow[d, hook, "d_1 \circ n'"'] \arrow[rr, hook, bend left, "der(t'_3)"] \arrow[rd, phantom, "(1)"] & R' \arrow[r, hook, "i'_3"] \arrow[d, hook, "g_1 \circ b_1"] \arrow[rd, phantom, "(2)"] & \circ \arrow[d, hook, dotted, "r_1"] \arrow[rdd, hook, dotted, bend left, "q_1"] & \\
B \arrow[r, hook, "g_2"] \arrow[rrrd, hook, dotted, bend right, "q_2"'] & C \arrow[r, hook, dotted, "r_2"] & \circ \arrow[rd, hook, dotted, "\bar{i}'_3"] \arrow[d, phantom, "(=)"] \arrow[r, phantom, shift left=3ex, "(=)"] & \\
 & & & D
\end{tikzcd}$$

$$\begin{tikzcd}
I'_L \arrow[r, hook, "i_{L,2}"] \arrow[rd, hook, "m"'] & L \arrow[rr, hook, "o"] \arrow[rrrrr, hook, bend left, "\overline{ac}'"] & & O \arrow[rr, hook, dotted, "c_2"] \arrow[d, phantom] \arrow[ddd, hook, dotted, bend right=10, "e_1"'] \arrow[dd, hook, "a''"] & & \circ \arrow[lddd, hook, dotted, "f_1"] \arrow[rdd, hook, dotted, "c_3"] & L' \arrow[l, hook', dotted, "c_1"'] \arrow[dd, hook, dotted, "b_1 \circ \overline{m}'_1"] \arrow[rdddd, hook, "a'_1"] & \\
 & R_i \arrow[rru, hook, "o'"'] \arrow[d, hook, "id_{R_i}"'] & & & & & & \\
 & R_i \arrow[rr, hook, "der(t'_2)"] \arrow[dd, hook, "n_1"'] & & R'' \arrow[rrr, hook, dotted, "b_2"] \arrow[dd, hook, dotted, "d_1"] & & & \circ \arrow[dd, hook, dotted, "g_1"] & \\
 & & \circ \arrow[rr, hook, dotted, "f_2"] \arrow[rd, hook, dotted, "e_3"] & & \circ \arrow[rd, hook, dotted, "i_2"] & & & \\
 & A \arrow[ru, hook, dotted, "e_2"] \arrow[rr, hook, dotted, "d_2"] \arrow[rrrrrr, hook, bend right, "a'_2"'] & & B \arrow[rr, hook, dotted, "h_2"] \arrow[rrr, hook, dotted, bend right, "g_2"'] & & \circ \arrow[r, hook, dotted, "i_3"] & C \arrow[llu, hook, dotted, "i_1"'] \arrow[r, hook] & A'
\end{tikzcd}$$

For the pushout in Def. 3.28 and Item 2c, we construct a pushout (b_1, b_2) over $(n', \overline{ac})$ and by pushout composition and Def. 3.28 and Item 2(d)ii obtain pushout $(b_1 \circ \overline{m}'_1, b_2)$ over $(\overline{ac}', a'' \circ o)$. We construct pushout (c_1, c_2) over $(\overline{ac}', o)$ with induced morphism c_3 and $c_3 \circ c_1 = b_1 \circ \overline{m}'_1$ $^{(*^3)}$ and obtain pushout (c_3, b_2) over (c_2, a'') by pushout decomposition. We construct pushout (d_1, d_2) over $(der(t'_2), n_1)$, i.e., over $(a'' \circ o', n_1 \circ id_{R_i})$ by Def. 3.28 and Item 2(d)iii, and construct pushout (e_1, e_2) over $(o', n_1 \circ id_{R_i})$ with induced morphism e_3, (e_3, d_1) being a pushout over (a'', e_1) by pushout decomposition and therefore by $\mathscr{M}$-preservation along pushouts, $e_3 \in \mathscr{M}$ with $e_3 \circ e_1 = d_1 \circ a''$ and $e_3 \circ e_2 = d_2$ $^{(*^2)}$. We construct pushout (f_1, f_2) over (c_2, e_1). By pushout composition, $(f_1 \circ c_1, f_2 \circ e_2)$ is a pushout over $(\overline{ac}' \circ i_{L,2}, n_1 \circ id_{R_i} \circ m)$ and therefore also a pullback, since by $\mathscr{M}$-composition $\overline{ac}' \circ i_{L,2} \in \mathscr{M}$. Therefore by effective pushouts, there is $i_1 \in \mathscr{M}$ with $i_1 \circ a'_1 = f_1 \circ c_1$ and $i_1 \circ a'_2 = f_2 \circ e_2$ $^{(*^4)}$. We construct pushout (g_1, g_2) over (b_2, d_1) and pushout (i_2, h_2) over (f_2, e_3). By pushout composition, $(i_2 \circ f_1, h_2)$ and $(g_1 \circ c_3, g_2)$ are pushouts over $(c_2, d_1 \circ a'') \overset{(*^2)}{=} (c_2, e_3 \circ e_1)$ and also pullbacks by $c_2 \in \mathscr{M}$. Therefore again by effective pushouts, we obtain $i_3 \in \mathscr{M}$ with $i_3 \circ i_2 \circ f_1 = g_1 \circ c_3$ and $i_3 \circ h_2 = g_2$ $^{(*^5)}$. Thus, $i_3 \circ i_2 \circ i_1 \circ a'_1 \overset{(*^4)}{=} i_3 \circ i_2 \circ f_1 \circ c_1 \overset{(*^5)}{=} g_1 \circ c_3 \circ c_1 \overset{(*^3)}{=} g_1 \circ b_1 \circ \overline{m}'_1$ and $i_3 \circ i_2 \circ i_1 \circ a'_2 \overset{(*^4)}{=} i_3 \circ i_2 \circ f_2 \circ e_2 \overset{(PO)}{=} i_3 \circ h_2 \circ e_3 \circ e_2 \overset{(*^2)}{=} i_3 \circ h_2 \circ d_2 \overset{(*^5)}{=} g_2 \circ d_2$ $^{(*^6)}$. By induction hypothesis, for $(\overline{ac}, \overline{m}'_1 \circ \overline{n}')$ there is $t'_3 \colon R_i \overset{*}{\Rightarrow}_{GS,n''} \text{-}$ and $i'_3 \in \mathscr{M}$ with $i'_3 \circ \overline{ac} = der(t'_3)$ and $i'_3 \circ \overline{m}'_1 \circ \overline{n}' = n''$ $^{(*^7)}$. By pushout composition of pushouts (b_1, b_2) and (g_1, g_2), (1) is a pushout. We construct pushout (2) with (1) + (2) being a pushout by composition. Furthermore we construct pushout (q_1, q_2) over $(der(t'_3), d_1 \circ n')$ which is also a pullback by $d_1 \circ n' \in \mathscr{M}$ and therefore, we obtain a sequence t_2 of recursive transformation steps from A via t'_2 and t'_3 over n_1 with derived span $q_2 \circ d_2$ and co-match $q_1 \circ n''$ by pushout (de)-compositions analogously to proving the base case for Prop. 3.7 and Items 1 and 2. By effective pushouts, there is $\bar{i}'_3 \in \mathscr{M}$ with $\bar{i}'_3 \circ r_1 = q_1$ and $\bar{i}'_3 \circ r_2 \circ g_2 = q_2$ $^{(*^8)}$. Thus, $\bar{i}'_3 \circ r_2 \circ i_3 \circ i_2 \circ i_1 \circ a'_1 \circ \overline{n}' \overset{(*^6)}{=} \bar{i}'_3 \circ r_2 \circ g_1 \circ b_1 \circ \overline{m}'_1 \circ \overline{n}' \overset{(2)}{=} \bar{i}'_3 \circ r_1 \circ i'_3 \circ \overline{m}'_1 \circ \overline{n}' \overset{(*^7)}{=} \bar{i}'_3 \circ r_1 \circ n'' \overset{(*^8)}{=} q_1 \circ n''$ and $\bar{i}'_3 \circ r_2 \circ i_3 \circ i_2 \circ i_1 \circ a'_2 \overset{(*^6)}{=} \bar{i}'_3 \circ r_2 \circ g_2 \circ d_2 \overset{(*^8)}{=} q_2 \circ d_2$ $^{(*^9)}$.

$$\begin{tikzcd}
 & \circ \arrow[d, hook, "n_{j-1}"] \arrow[rdd, bend left, "n_1"] & & & D' & \circ \arrow[d, hook, "q_1 \circ n''"] \arrow[rdd, bend left, "a'_1 \circ \overline{n}'"] & \\
R \arrow[r, hook, "der(t_1)"] \arrow[rrd, hook, "ac_1"'] & \circ \arrow[rrru, dotted, hook, "s_2"] & & (3) & & D \arrow[lu, dotted, hook', "s_1"'] & \\
 & & A \arrow[lu, "i'_1"] \arrow[rrru, "q_2 \circ d_2"] \arrow[rrrr, hook, "a'_2"'] & & & & A' \arrow[lu, hook', "\bar{i}'_3 \circ r_2 \circ i_3 \circ i_2 \circ i_1"']
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & \circ \arrow[rr, hook, "p"] \arrow[d, hook, "m_j"] & & \circ \arrow[dddd, hook, "n'"] \\
 & & \circ \arrow[ld, hook', "s_1 \circ q_1 \circ n''"'] \arrow[ddd, hook, "a'_1 \circ \overline{n}'"] & (4) & \\
 & D' \arrow[rr, dotted, hook, "ac'_2"] & & D'' & \\
 & & & (5) & \\
R \arrow[ruu, hook, "der(t)"] \arrow[rr, hook, "a'_2 \circ ac_1"'] & & A' \arrow[luu, hook', "i"] \arrow[rr, hook, "ac_2"'] & & \circ \arrow[luu, dotted, hook', "i'"']
\end{tikzcd}$$

Finally, we extend t_1 from $(*^1)$ by t_2 to a recursive transformation $t \colon R \overset{*}{\Rightarrow}_{GS, s_1 \circ q_1 \circ n''} D'$ with $der(t) = s_2 \circ der(t_1)$ by constructing pushout (3). Therefore for $(a'_2 \circ ac_1, a'_1 \circ \overline{n}') \in \mathscr{C}$ with succeeding invocation $\underline{\mathscr{L}}_w(m, j, a'_2 \circ ac_1, a'_1 \circ \overline{n}', 2)$, there is $i = s_1 \circ \bar{i}'_3 \circ r_2 \circ i_3 \circ i_2 \circ i_1 \in \mathscr{M}$ by $\mathscr{M}$-composition and recursive transformation t with $i \circ a'_2 \circ ac_1 \overset{(*^9)}{=} s_1 \circ q_2 \circ d_2 \circ ac_1 \overset{(3)}{=} s_2 \circ i'_1 \circ ac_1 \overset{(*^1)}{=} s_2 \circ der(t_1) = der(t)$ and $i \circ a'_1 \circ \overline{n}' \overset{(*^9)}{=} s_1 \circ q_1 \circ n''$. The case for $der(t'_2) = n' = id_{R_i}$ in Def. 3.28 and Item 2d is shown analogously by omitting recursive transformation t'_2. For succeeding invocation $\underline{\mathscr{L}}_w(m, j, a'_2 \circ ac_1, a'_1 \circ \overline{n}', 2)$ with switch $k = 2$, pushout (4) over $(p, a'_1 \circ \overline{n}' \circ m_j)$ is obtained from the recursive step of the construction.

We construct pushout (5) over (ac_2, i) and obtain pushout $(4)+(5)$ by composition which extends recursive transformation t to a recursive transformation $t'\colon R \overset{*}{\Rightarrow}_{GS,i'\circ n'} D''$ with $der(t') = ac'_2 \circ der(t)$. Thus for succeeding invocation $\underline{\mathscr{L}}_w(m, j+1, ac_2 \circ a'_2 \circ ac_1, n', 1)$, there is $i' \in \mathscr{M}$ by $\mathscr{M}$-preservation of i along pushout (5) and recursive transformation t' with $i' \circ ac_2 \circ a'_2 \circ ac_1 \overset{(5)}{=} der(t')$ and $i' \circ n' = i' \circ n'$. For $(ac, n) \in \underline{\mathscr{L}}_w(m = (m_1\colon L \to R \dots m_j \dots m_n), 1, id_R, id_R, \_)$, by induction over the construction for match-path m we finally obtain that there is $t''\colon R \overset{*}{\Rightarrow}_{GS} A''$ and $i'' \in \mathscr{M}$ with $i'' \circ ac = der(t'')$. Note that if m is terminating and starts in S then also t'' is terminating and starts in S, since, by (A) and the construction of $\underline{\mathscr{L}}_w$, a step $((4)+(5))$ via match m_1 (m_n) is the first (last) step in t''. Thus, by the definition of $\mathscr{L}_w(GS)$ with m being terminating and starting in S, for each $[ac] \in \mathscr{L}_w(GS)$ there is terminating $t''\colon S \overset{*}{\Rightarrow}_{GS} \_$ which starts in S and morphism $i'' \in \mathscr{M}$ with $i'' \circ ac = der(t'')$. □

A.14 Proof of Sec. 3.5 and Lem. 3.16

Let $Restr_t(G)$ be the restriction of G along t with injection $t'\colon Restr_t(G) \to G$. We first show for positive conditions ac_P that $\exists p\colon Restr_t(P) \to G \in \mathscr{M}$ with $p \models Restr_t(ac_P)$ if and only if $\exists p_R\colon Restr_t(P) \to Restr_t(G) \in \mathscr{M}$ with $p_R \models Restr_t(ac_P)$ such that $t' \circ p_R = p$ $(*^A)$.

$$\begin{tikzcd}
TG & & G \arrow[ll, "t_G"'] & & Restr_t(ac_C) \arrow[d, phantom, "\bigtriangledown"] & & \\
TG_R \arrow[u, "t"] \arrow[urr, phantom, "(1)"] & & Restr_t(G) \arrow[ll, "t_{G_R}"] \arrow[u, "t'"] & & Restr_t(C) \arrow[ll, dotted, "q_R"] \arrow[ull, "q"'] \arrow[llll, bend left, "t_{C_R}"] & & Restr_t(P) \arrow[ll, "a_R"] \arrow[ullll, bend right, "p"'] \arrow[llll, dotted, bend left, "p_R"]
\end{tikzcd}$$

“$\Rightarrow$” By induction over the structure of conditions: **Basis.** For $Restr_t(ac_P) = \textbf{true}$, by the construction of restricted conditions in Def. 3.32, p being a morphism from object $Restr_t(P)$ typed over TG via morphism $t \circ t_{C_R} \circ a_R$ to object G typed over TG via morphism t_G implies that $t \circ t_{C_R} \circ a_R = t_G \circ p$. Therefore, by the universal property of pullback (1) from the construction of $Restr_t(G)$ (cf. Def. 3.32), there is a morphism $p_R\colon Restr_t(P) \to Restr_t(G)$ with $t' \circ p_R = p$ and $p_R \models \textbf{true} = Restr_t(ac_P)$. By $\mathscr{M}$-decomposition with $p \in \mathscr{M}$ and $t' \in \mathscr{M}$ (since, $\mathscr{M}$-morphisms $t \in \mathscr{M}$ are closed under pullbacks (1)), $p_R \in \mathscr{M}$. **Hypothesis.** For restricted conditions $Restr_t(ac_C)$ and $Restr_t(ac_{P,i}), i \in I$ the assumption holds. **Step.** For $Restr_t(ac_P) = \exists(a_R\colon Restr_t(P) \to Restr_t(C), Restr_t(ac_C))$, $p \models Restr_t(ac_P)$ implies that there is $q\colon Restr_t(C) \to G \in \mathscr{M}$ with $q \circ a_R = p$ $(*^1)$. From q being a morphism it follows that $t_G \circ q = t \circ t_{C_R}$. Thus, by the universal property of pullback (1) there is $q_R\colon Restr_t(C) \to Restr_t(G)$ with $t' \circ q_R = q$ $(*^2)$. By $\mathscr{M}$-decomposition with $q, t' \in \mathscr{M}$, $q_R \in \mathscr{M}$. Analogously from p being a morphism, it follows that there is p_R with $t' \circ p_R = p$ $(*^3)$ and furthermore, from $p, t' \in \mathscr{M}$ by $\mathscr{M}$-decomposition, $p_R \in \mathscr{M}$. Thus, $t' \circ p_R \overset{(*^3)}{=} p \overset{(*^1)}{=} q \circ a_R \overset{(*^2)}{=} t' \circ q_R \circ a_R$. By $t' \in \mathscr{M}$ being a monomorphism, it follows that $q_R \circ a_R = p_R$. By induction hypothesis and assumption $q \models Restr_t(ac_C)$, there is q'_R with $q'_R \models Restr_t(ac_C)$ and $t' \circ q'_R = q \overset{(*^2)}{=} t' \circ q_R$. By $t' \in \mathscr{M}$ being a monomorphism, $q'_R = q_R$. Thus, $p_R \models Restr_t(ac_P)$. For $Restr_t(ac_P) = \wedge_{i\in I}(Restr_t(ac_{P,i}))$ we obtain morphisms $p_{R,i} \in \mathscr{M}$ with $p_{R,i} \models Restr_t(ac_{P,i})$ and $t' \circ p_{R,i} = p$. By $t' \in \mathscr{M}$ being a monomorphism, $p_{R,i_1} = p_{R,i_2} = \underline{p}$ for all $i_1, i_2 \in I$ and therefore, $\underline{p} \models Restr_t(ac_P)$. Analogously, we prove the step for $Restr_t(ac_P) = \vee_{i\in I}(Restr_t(ac_{P,i}))$.

"$\Leftarrow$" By induction over the structure of conditions: **Basis.** For $Restr_t(ac_P) = \textbf{true}$, there is $t' \circ p_R \colon Restr_t(P) \to G \in \mathscr{M}$ by $\mathscr{M}$-composition with $t' \circ p_R \models \textbf{true} = Restr_t(ac_P)$. **Hypothesis.** For restricted conditions $Restr_t(ac_C)$ and $Restr_t(ac_{P,i}), i \in I$ the assumption holds. **Step.** For $Restr_t(ac_P) = \exists(a_R \colon Restr_t(P) \to Restr_t(C), Restr_t(ac_C))$, by assumption $p_R \models Restr_t(ac_P), p_R \in \mathscr{M}$ there is $q_R \in \mathscr{M}$ with $q_R \circ a_R = p_R$ and $q_R \models Restr_t(ac_C)$. By $\mathscr{M}$-composition, there are $t' \circ p_R, t' \circ q_R \in \mathscr{M}$ with $t' \circ q_R \circ a_R = t' \circ p_R$. By induction hypothesis, $t' \circ q_R \models Restr_t(ac_C)$. Thus, $t' \circ p_R \models Restr_t(ac_P)$. For $Restr_t(ac_P) = \wedge_{i \in I}(Restr_t(ac_{P,i}))$ we obtain morphisms $p_i \in \mathscr{M}$ with $p_i \models Restr_t(ac_{P,i})$ and $t' \circ p_R = p_i$. Thus, $p_{i_1} = p_{i_2} = \underline{p}$ for all $i_1, i_2 \in I$ and therefore, $\underline{p} \models Restr_t(ac_P)$. Analogously, we prove the step for $Restr_t(ac_P) = \vee_{i \in I}(Restr_t(ac_{P,i}))$.

For non-positive conditions $Restr_t(\neg ac'_P) = \neg Restr_t(ac'_P)$ with negations we conclude as follows. We already have shown that there exists $p, p_R \in \mathscr{M}$ with $t' \circ p_R = p$ for both directions. For $p \models \neg Restr_t(ac'_P)$ assume that $p_R \models Restr_t(ac'_P)$. Then, by the result from before it follows that there is $p' = t' \circ p_R = p$ with $p' = p \models Restr_t(ac'_P)$ contradicting with assumption $\neg(p \models Restr_t(ac'_P))$. Therefore, $p_R \models \neg Restr_t(ac'_P)$, i.e., $p \models Restr_t(\neg ac'_P)$ implies $p_R \models Restr_t(\neg ac'_P)$. The other direction is shown analogously. The main difference is that we obtain a morphism p'_R with $p'_R \models Restr_t(ac'_P)$ and $t' \circ p'_R = p = t' \circ p_R$. By $t' \in \mathscr{M}$ being a monomorphism (as already shown before), $p'_R = p_R$ and therefore, $p_R \models Restr_t(ac'_P)$ contradicts with assumption $\neg(p_R \models Restr_t(ac'_P))$. Therefore, $p_R \models Restr_t(\neg ac'_P)$ implies $p \models Restr_t(\neg ac'_P)$.

Now, we prove the main results of Lem. 3.16. For direction "$\Leftarrow$", we use the uniqueness of initial morphisms, i.e., for initial morphisms $i_{P_R} \colon I \to Restr_t(P)$, $i_G \colon I \to G$ and $p \circ i_{P_R} \colon I \to G$ from initial object I and morphism $p \colon Restr_t(P) \to G$ it holds that $p \circ i_{P_R} = i_G$ $^{(*^1)}$. The same is true for direction "$\Rightarrow$" and initial morphisms $i_{P_R}, i_{G_R} \colon I \to Restr_t(G)$ and $p_R \circ i_{P_R} \colon I \to Restr_t(G)$ with $p_R \circ i_{P_R} = i_{G_R}$ $^{(*^2)}$.

"$\overset{I}{\models}$" $G \overset{I}{\models} Restr_t(ac_P) \overset{(*^1)}{\Leftrightarrow} \exists p \colon Restr_t(P) \to G \in \mathscr{M}$ with $p \models Restr_t(ac_P) \overset{(*^A)}{\Leftrightarrow} \exists p_R \colon Restr_t(P) \to Restr_t(G) \in \mathscr{M}$ with $p_R \models Restr_t(ac_P) \overset{(*^2)}{\Leftrightarrow} Restr_t(G) \overset{I}{\models} Restr_t(ac_P)$.

"$\models$" $G \models Restr_t(ac_P) \overset{(*^1)}{\Leftrightarrow} \neg\exists p \colon Restr_t(P) \to G \in \mathscr{M}$ with $p \models Restr_t(\neg ac_P) \overset{(*^A)}{\Leftrightarrow} \neg\exists p_R \colon Restr_t(P) \to Restr_t(G) \in \mathscr{M}$ with $p_R \models Restr_t(\neg ac_P) \overset{(*^2)}{\Leftrightarrow} Restr_t(G) \models Restr_t(ac_P)$. □

A.15 Proof of Sec. 3.5 and Thm. 3.10

Let ac'_P be an extension of a plain condition $ac_P \in C_I \cup C_G$ via C_G and $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ be an object. For morphism $p \colon P \to G$ we prove that $p \models ac_P$ if and only if $p \models ac'_P$. By induction over plain positive conditions:

"$\Rightarrow$" **Basis.** For $ac_P = \textbf{true}$, $p \models ac'_P = \textbf{true}$. For $ac_P = \exists(a \colon P \to C, \textbf{true})$, let $(m_i \colon P_i \to E_i \in \mathscr{M}, ac_{P_i,i} \in C_G)_{i \in \{1,\ldots,n\}}$ be the matches m_i and conditions $ac_{P_i,i} \equiv \vee_{j \in J} \exists(a_j \colon P_i \to C_j, \textbf{true})$ that are used for the step-wise extension ac'_P according to Sec. 3.2 and Def. 3.5. By $p \models ac_P$, there exists $q_1 \colon C \to G \in \mathscr{M}$. For each step i of the extension we conclude as follows. Assumption $G \in \mathscr{L}(C_G)$, i.e., $G \models ac_{P_i,i}$, implies that there exists $q'_{i+1} \colon C_j \to G \in \mathscr{M}$ with $q_i \circ m_i = q'_{i+1} \circ a_j$ for some $j \in J$ according to Sec. 2.2.3 and Rem. 2.10 and match $q_i \circ m_i \in \mathscr{M}$ by $\mathscr{M}$-composition. We create pullback (m'_i, a'_j) over (q_i, q'_{i+1}) and subsequent pushout (1) where all morphisms are in $\mathscr{M}$, since, $\mathscr{M}$-morphisms (q_i and

q'_{i+1}) are closed under pushouts and pullbacks. By the universal pullback property, there is $p_i \colon P_i \to P'_i$ with $m_i = m'_i \circ p_i$ and $a_j = a'_j \circ p_i$ implying further by $\mathscr{M}$-decomposition and $m_i, m'_i \in \mathscr{M}$ that $p_i \in \mathscr{M}$. Furthermore by effective pushouts, there is $q_{i+1} \colon E_{i+1} \to G \in \mathscr{M}$ with $q_i = q_{i+1} \circ e_i$. By definition Sec. 3.2 and Def. 3.4, E_{i+1} is not C_G-inconsistent, since, $G \models c, \forall c \in C_G$ by assumption $G \in \mathscr{L}(C_G)$. Assume that E_{i+1} is C_G-inconsistent, i.e., $E_{i+1} \not\models c$ for some $c \in C_G$ that is violation stable under embedding, then also $G \not\models c$ via inclusion q_{i+1} contradicting with the statement from before. Therefore by constructions Defs. 3.5 and 3.35, $p \models ac'_P$.

$$\begin{tikzcd}
& P_1 \arrow[r, dotted, "p_1"] \arrow[d, "m_1"] \arrow[rr, bend left, "a_j"] & P'_1 \arrow[r, dotted, "a'_j"] \arrow[dl, dotted, "m'_1"] \arrow[dr, phantom, "(1)"] & C_j \arrow[d, dotted, "e'_1"] \arrow[ddll, bend left=60, "q'_2"] \\
P \arrow[r, "a"] \arrow[dr, "p"'] & C = E_1 \arrow[rr, dotted, "e_1"] \arrow[d, "q_1"] & & E_2 \arrow[dll, dotted, "q_2"] \\
& G & &
\end{tikzcd}$$

Hypothesis. The assumption holds for plain conditions $ac_{P,i}, i \in I$. **Step.** For $ac_P = \vee_{i \in I}(ac_{P,i})$, $p \models ac_P$ implies that $p \models ac_{P,i}$ for some $i \in I$. Let $ac'_{P,i}$ be the extension of $ac_{P,i}$ in ac'_P, then by induction hypothesis, $p \models ac'_{P,i}$ and therefore, $p \models ac'_P$ by construction Def. 3.35. Analogously, we conclude for $ac_P = \wedge_{i \in I}(ac_{P,i})$.

"$\Leftarrow$" **Basis.** For $ac_P = \textbf{true}$, $p \models ac_P$. For $ac_P = \exists(a \colon P \to C, \textbf{true})$ and extension $ac'_P = \vee_{i \in \{1,\ldots,n\}}(\exists(P \xrightarrow{e_i \circ a} E_i, \textbf{true}))$, $p \models ac'_P$ implies that $p \models \exists(P \xrightarrow{e_i \circ a} E_i, \textbf{true})$ for some i, i.e., there exists $q_i \colon E_i \to G \in \mathscr{M}$ with $p = q_i \circ e_i \circ a$. By construction Sec. 3.2 and Def. 3.5 and $\mathscr{M}$-composition, $e_i \in \mathscr{M}$, since, $\mathscr{M}$-morphisms a'_j are closed under pushouts (1). Therefore by $\mathscr{M}$-composition, $q_i \circ e_i \in \mathscr{M}$ and furthermore $q_i \circ e_i \models \textbf{true}$, i.e., $p \models ac_P$. **Hypothesis.** The assumption holds for plain conditions $ac_{P,i}, i \in I$. **Step.** For $ac_P = \vee_{i \in I}(ac_{P,i})$, $p \models ac'_P$ implies that $p \models ac'_{P,i}$ for some $i \in I$ and extension $ac'_{P,i}$ of $ac_{P,i}$. By induction hypothesis, $p \models ac_{P,i}$, i.e., $p \models ac_P$. Analogously, we conclude for $ac_P = \wedge_{i \in I}(ac_{P,i})$.

For plain non-positive conditions $ac_P = \neg \underline{ac}_P$, let $ac'_P = \neg \underline{ac}'_P$ be an extension of ac_P with extension $\underline{ac}'_P$ of $\underline{ac}_P$.

"$\Rightarrow$" Assume that $p \not\models ac'_P = \neg \underline{ac}'_P$, i.e., $p \models \underline{ac}'_P$. Therefore, $p \models \underline{ac}_P$ contradicting with assumption $p \models ac_P$, i.e., $\neg(p \models \underline{ac}_P)$. Therefore, $p \models ac'_P$.

"$\Leftarrow$" Assume that $p \not\models ac_P$, i.e., $p \models \underline{ac}_P$. Therefore, $p \models \underline{ac}'_P$ contradicting with assumption $p \models ac'_P = \neg \underline{ac}'_P$, i.e., $\neg(p \models \underline{ac}'_P)$. Therefore, $p \models ac_P$.

Based on the fact from above, in the following, we proof that $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ if and only if $G \in \mathscr{L}_I(C_I \cup C'_I) \cap \mathscr{L}(C_G \cup C'_G)$.

"$\Rightarrow$" $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$ implies that $G \overset{I}{\models} ac_{P,I}, \forall ac_{P,I} \in C_I$ and $G \models ac_{P,G}, \forall ac_{P,G} \in C_G$. For each extension $ac'_{P,I} \in C'_I$ of some $ac_{P,I} \in C_I$ we conclude as follows. By Sec. 2.2.3 and Rem. 2.10, $G \overset{I}{\models} ac_{P,I}$ implies that there exists $p \colon P \to G$ with $p \models ac_{P,I}$ implying further that $p \models ac'_{P,I}$, i.e., $G \overset{I}{\models} ac'_{P,I}$ according to Sec. 2.2.3 and Rem. 2.10. Thus, $G \in \mathscr{L}_I(C_I \cup C'_I)$. For each extension $ac'_{P,G} \in C'_G$ we conclude analogously leading to $G \in \mathscr{L}(C_G \cup C'_G)$. Therefore, $G \in \mathscr{L}_I(C_I \cup C'_I) \cap \mathscr{L}(C_G \cup C'_G)$.

"$\Leftarrow$" $G \in \mathscr{L}_I(C_I \cup C'_I) \cap \mathscr{L}(C_G \cup C'_G)$ implies that $G \overset{I}{\models} c_I, \forall c_I \in C_I$ and $G \models c_G, \forall c_G \in C_G$ implying further that $G \in \mathscr{L}_I(C_I) \cap \mathscr{L}(C_G)$. □

Bibliography

[AGG16] AGG 2.1 - The Attributed Graph Grammar System. http://www.user.tu-berlin.de/o.runge/agg/, 2016.

[AHS90] Jiří Adámek, Horst Herrlich, and George Strecker. *Abstract and Concrete Categories*. Wiley-Interscience, New York, NY, USA, 1990.

[ALS15] Anthony Anjorin, Erhan Leblebici, and Andy Schürr. 20 years of triple graph grammars: A roadmap for future research. *ECEASST*, 73, 2015.

[AN00] Peter R.J. Asveld and Anton Nijholt. The inclusion problem for some subclasses of context-free languages. *Theoretical Computer Science*, 230(12):247 – 256, 2000.

[ASU06] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[BBG05a] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer, 2005.

[BBG05b] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer-Verlag, Berlin/Heidelberg, 2005.

[BDK$^+$12] Nathalie Bertrand, Giorgio Delzanno, Barbara König, Arnaud Sangnier, and Jan Stückrath. On the decidability status of reachability and coverability in graph transformation systems. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, pages 101–116, 2012.

[BEGG10] Benjamin Braatz, Hartmut Ehrig, Karsten Gabriel, and Ulrike Golas. Finitary m-adhesive categories. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr, editors, *Proceedings of Intern. Conf. on Graph Transformation (ICGT' 10)*, volume 6372, pages 234–249, 2010.

[Béz05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[BW13] Alexander Bergmayr and Manuel Wimmer. Generating metamodels from grammars by chaining translational and by-example techniques. In *Proceedings of the*

First International Workshop on Model-driven Engineering By Example co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29, 2013., pages 22–31, 2013.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, CA, USA, 2003.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[CHH$^+$12] Andrea Corradini, Reiko Heckel, Frank Hermann, Susann Gottmann, and Nico Nachtigall. Transformation systems with incremental negative application conditions. In *Recent Trends in Algebraic Development Techniques, 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers*, pages 127–142, 2012.

[DHJ$^+$06] Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. *Adaptive Star Grammars*, pages 77–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[DR15] Brahma Dathan and Sarnath Ramnath. *Object-Oriented Analysis, Design and Implementation: An Integrated Approach*, chapter Modelling with Finite State Machines, pages 275–322. Springer International Publishing, Cham, 2015.

[DXC10a] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations. In *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, pages 61–76, 2010.

[DXC10b] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. Specifying overlaps of heterogeneous models for global consistency checking. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Oslo, Norway, October 2-8, 2010, Reports and Revised Selected Papers*, pages 165–179, 2010.

[DXC$^+$11] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, pages 304–318, 2011.

[EEE$^+$07] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information preserving bidirectional model transformations. In *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 72–86, 2007.

[EEGH15] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation: General Framework and Applications*. Springer, 2015.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EKMR99] H. Ehrig, H.J. Kreowski, U. Montanari, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific, 1999.

[EM90] H. Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[Fet99] James H. Fetzer. The role of models in computer science. *The Monist*, 82(1):20–36, 1999.

[GdLK+13] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. Engineering model transformations with transML. *Software & Systems Modeling*, 12(3):555–577, 2013.

[GEH11] Ulrike Golas, Hartmut Ehrig, and Frank Hermann. Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. *ECEASST*, 2011.

[GHN+13] Susann Gottmann, Frank Hermann, Nico Nachtigall, Benjamin Braatz, Claudia Ermel, Hartmut Ehrig, and Thomas Engel. Correctness and completeness of generalised concurrent model synchronisation based on triple graph grammars. In *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, 2013.

[GLEO12] Ulrike Golas, Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. *Theor. Comput. Sci.*, 424:46–68, 2012.

[GNE+16] Susann Gottmann, Nico Nachtigall, Claudia Ermel, Frank Hermann, and Thomas Engel. Towards the propagation of model updates along different views in multi-view models. In *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.*, pages 45–60, 2016.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[HEGO10] Frank Hermann, Hartmut Ehrig, Ulrike Golas, and Fernando Orejas. Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In *Proceedings of the First International Workshop on Model-Driven Interoperability*, MDI '10, pages 22–31, New York, NY, USA, 2010. ACM.

[Hen15] HenshinTGG. https://github.com/de-tu-berlin-tfs/Henshin-Editor, 2015.

[HEO$^+$11] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of model synchronization based on triple graph grammars - extended version. Technical Report TR 201107, TU Berlin, Fak. IV, 2011.

[HEO$^+$15] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software and System Modeling*, 14(1):241–269, 2015.

[HEOG10] F. Hermann, H. Ehrig, F. Orejas, and U. Golas. Formal analysis of functional behaviour for model transformations based on triple graph grammars - extended version. Technical Report 2010/08, 2010.

[HGN$^+$13] Frank Hermann, Susann Gottmann, Nico Nachtigall, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, and Thomas Engel. On an automated translation of satellite procedures using triple graph grammars. In *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, pages 50–51, 2013.

[HGN$^+$14] Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, and Claudia Ermel. Triple graph grammars in the large for translating satellite procedures. In *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, pages 122–137, 2014.

[HKP05] David Harel, Hillel Kugler, and Amir Pnueli. *Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday*, chapter Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements, pages 309–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[HLH$^+$13] Stephan Hildebrandt, Leen Lambers, Giese Holger, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A Survey of Triple Graph Grammar Tools. In *Bidirectional Transformations*, volume 57, pages 1–18. EC-EASST, 2013.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[HNB$^+$14] Frank Hermann, Nico Nachtigall, Benjamin Braatz, Thomas Engel, and Susann Gottmann. Solving the fixml2code-case study with henshintgg. In *Proceedings of the 7th Transformation Tool Contest part of the Software Technologies: Applications and Foundations (STAF 2014) federation of conferences, York, United Kingdom, July 25, 2014.*, pages 32–46, 2014.

[HP05] Annegret Habel and Karl-Heinz Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and Systems*

Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday, pages 293–308, 2005.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[HS04] Zhaoxia Hu and Sol M. Shatz. Mapping uml diagrams to a petri net notation for system simulation. In Frank Maurer and Gnther Ruhe, editors, *SEKE*, pages 213–219, 2004.

[HSGP13] Brian Henderson-Sellers and Cesar Gonzalez-Perez. *Domain Engineering: Product Lines, Languages, and Conceptual Models*, chapter Multi-Level Meta-Modelling to Underpin the Abstract and Concrete Syntax for Domain-Specific Modelling Languages, pages 291–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[JABK08] Frdric Jouault, Freddy Allilaire, Jean Bzivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(12):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[JR15] Michael Johnson and Robert D. Rosebrugh. Spans of delta lenses. In *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015.*, pages 1–15, 2015.

[KLR$^+$12] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. *Conceptual Modelling and Its Theoretical Foundations: Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday*, chapter Model Transformation By-Example: A Survey of the First Wave, pages 197–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings*, chapter The Epsilon Transformation Language, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[LAD$^+$14] Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani, and Manuel Wimmer. Model transformation intents and their properties. *Software & Systems Modeling*, pages 1–38, 2014.

[LS99] Stephen Lack and Pawel Sobocinski. Adhesive and quasiadhesive categories, 1999.

[Mah09] Bernd Mahr. Information science and the logic of models. *Software and System Modeling*, 8(3):365–383, 2009.

[Mah10] Bernd Mahr. Position statement: Models in software and systems development. *ECEASST*, 30, 2010.

[Men13] Tom Mens. *Model Transformation: A Survey of the State of the Art*, pages 1–19. John Wiley Sons, Inc., 2013.

[MFBC12] Pierre-Alain Muller, Frédéric Fondement, Benoît Baudry, and Benoît Combemale. Modeling modeling modeling. *Software & Systems Modeling*, 11(3):347–359, 2012.

[MG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)Graph and Model Transformation 2005.

[NBE13] Nico Nachtigall, Benjamin Braatz, and Thomas Engel. Symbolic execution of satellite control procedures in graph-transformation-based EMF ecosystems. In *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2013, co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, October 1st, 2013.*, pages 61–66, 2013.

[NHBE14] Nico Nachtigall, Frank Hermann, Benjamin Braatz, and Thomas Engel. Towards domain completeness for model transformations based on triple graph grammars. In *Proceedings of the Third International Workshop on Verification of Model Transformations co-located with Software Technologies: Applications and Foundations, VOLT@STAF 2014, York, UK, July 21, 2014.*, pages 46–55, 2014.

[NMN$^+$92] R. De Nicola, U. Montanari, M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96(1):3 – 33, 1992.

[Put01] Janis Putman. *Architecting with RM-ODP*. Prentice Hall, 2001.

[Qua98] Terry Quatrani. *Visual Modeling with Rational Rose and UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[QVT15] QVT, 2015.

[Rei85] Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[RHM$^+$14] Louis M. Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp, Sebastian Buchwald, Tassilo Horn, Elina Kalnina, Andreas Koch, Kevin Lano, Bernhard Schätz, and Manuel Wimmer. Graph and model transformation tools for model migration. *Software & Systems Modeling*, 13(1):323–359, 2014.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[SC12] Jess Snchez Cuadrado. Towards a family of model transformation languages. In Zhenjiang Hu and Juan de Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 176–191. Springer Berlin Heidelberg, 2012.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, pages 151–163, 1994.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25, September 2003.

[SEM$^+$12] Hanna Schölzel, Hartmut Ehrig, Maria Maximova, Karsten Gabriel, and Frank Hermann. Satisfaction, restriction and amalgamation of constraints in the framework of m-adhesive categories. In *Proceedings Seventh ACCAT Workshop on Applied and Computational Category Theory, ACCAT 2012, Tallinn, Estonia, 1 April 2012.*, pages 83–104, 2012.

[SK08] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, pages 411–425, 2008.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, New York, 1973.

[Str08] Michael Striewe. Using a triple graph grammar for state machine implementations. pages 514–516. 2008.

[SW13] James Stanier and Des Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*, 45(3):26:1–26:27, July 2013.

[TA15] Frank Trollmann and Sahin Albayrak. Extending model to model transformation results from triple graph grammars to multiple models. In *Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pages 214–229, 2015.

[Tha11] Bernhard Thalheim. The theory of conceptual models, the theory of conceptual modelling and foundations of conceptual modelling. In David W. Embley and Bernhard Thalheim, editors, *Handbook of Conceptual Modeling: Theory, Practice and Research Challenges*, pages 543–577. Springer, Berlin, Heidelberg, 2011.

[UML15] Object Management Group Unified Modeling Language (UML) Version 2.5, 2015.

[WHR14] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *Software, IEEE*, 31(3):79–85, May 2014.

[xte16] Xtext Framework, 2016.

Index