Composite Modeling based on Distributed Graph Transformation and the Eclipse Modeling Framework

Dissertation

vorgelegt von
Dipl.-Inf. Stefan Jurack

Vom Fachbereich 12 – Mathematik und Informatik –
der Philipps-Universitaet Marburg
zur Erlangung des akademischen Graden

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Promotionsausschuß:
Dekan: Prof. Dr. Manfred Sommer
Referent: Prof. Dr. Gabriele Taentzer, Philipps-Universitaet Marburg
Referent: Prof. Dr. Reiko Heckel, University of Leicester

Marburg, Oktober 2012

Abstract

Model-driven development (MDD) has become a promising trend in software engineering for a number of reasons. Models as the key artifacts help the developers to abstract from irrelevant details, focus on important aspects of the underlying domain, and thus master complexity. As software systems grow, models may grow as well and finally become possibly too large to be developed and maintained in a comprehensible way. In traditional software development, the complexity of software systems is tackled by dividing the system into smaller cohesive parts, so-called components, and let distributed teams work on each concurrently. The question arises how this strategy can be applied to model-driven development.

The overall aim of this thesis is to develop a formalized modularization concept to enable the structured and largely independent development of interrelated models in larger teams. To this end, this thesis proposes component models with explicit export and import interfaces where exports declare what is provided while imports declare what it needed. Then, composite model can be connected by connecting their compatible export and import interfaces yielding so-called composite models. Suitable to composite models, a transformation approach is developed which allows to describe changes over the whole composition structure.

From the practical point of view, this concept especially targets models based on the Eclipse Modeling Framework (EMF). In the modeling community, EMF has evolved to a very popular framework which provides modeling and code generation facilities for Java applications based on structured data models.

Since graphs are a natural way to represent the underlying structure of visual models, the formalization is based on graph transformation. Incorporated concepts according to distribution heavily rely on distributed graph transformation introduced by Taentzer. Typed graphs with inheritance and containment structures are well suited to describe the essentials of EMF models. However, they also induce a number of constraints like acyclic inheritance and containment which have to be taken into account. The category-theoretical foundation in this thesis allows for the precise definition of consistent composite graph transformations satisfying all inheritance and containment conditions.

The composite modeling approach is shown to be coherent with the development of tool support for composite EMF models and composite EMF model transformation.

Zusammenfassung

Die modellgetriebenen Softwareentwicklung (MSD) ist ein vielversprechender Trend aus unterschiedlichsten Gründen. Modelle als die Schlüsselelemente in der MSD erlauben Entwicklern von irrelevanten Details zu abstrahieren, sich auf wichtige Aspekte der Zieldomäne zu konzentrieren und damit die Komplexität von Softwaresystemen zu meistern. Eine stetige Steigerung der Komplexität führt jedoch auch zu stetig wachsenden Modellen. Dies geht so lange gut, bis die Modelle dem Anspruch der guten Wartbarkeit und Verständlichkeit selbst nicht mehr genügen. In herkömmlicher Softwareentwicklung wird der Komplexität mit Verteilung begegnet, d.h. das Gesamtsystem wird eine Menge stark zusammengehörender Teile, den Softwarekomponenten, zerteilt. An diesen können dann verteilte Teams parallel arbeiten. Die Frage ist nun, ob und wie diese Strategie auch auf die modellgetriebene Softwareentwicklung angewendet werden kann.

Entsprechend ist das große Ziel dieser Arbeit die Entwicklung eines formal fundierten Modularisierungskonzepts, welches die strukturierte und weitgehend unabhängige Entwicklung von miteinander verknüpften Modellen in großen Entwicklerteams ermöglicht und unterstützt. Diesen Zweck sollen Komponentenmodelle mit expliziten Import- und Exportschnittstellen erfüllen. Exporte identifizieren die Teile die veröffentlicht werden, währende Importe die Teile benennt, welche von außen benötigt und konsumiert werden. Letztendlich werden Komponentenmodelle nur durch das Verbinden ihrer zueinander kompatiblen Export- und Importschnittstellen verbunden. Ein solch entstehender Modellverband wird *composite model* genannt. Passend zu solchen Verbänden wird ein Transformationsansatz entwickelt, welcher das Beschreiben von Modelländerungen über einzelne Modellgrenzen hinweg erlaubt und damit das parallele Ändern beliebiger Modelle eines Verbands.

Dieses Konzept wird von praktischer Seite mit besonderem Fokus auf eine bestehende Technologie entwickelt, dem Eclipse Modeling Framework (EMF). EMF ist in der MSD Welt weit verbreitet und erfreut sich weiter wachsender Beliebtheit. Beispielsweise bietet EMF, neben den Erweiterungen durch zahlreiche andere Eclipse-Projekte, eine vollständige Infrastruktur für das Generieren von Java Anwendungen basierend auf strukturierten Datenmodellen.

Da Graphen auf natürlich Weise die unterliegende Struktur von (visuellen) Modellen darstellen können, basiert der Formalismus in dieser Arbeit auf den Theorien und Konzepte der Graphtransformation. Dabei ist der Verteilungsaspekt insbesondere durch die Arbeiten von Taentzer zu Verteilter Graphtransformation geprägt. Die wesentlichen Strukturelemente von EMF Modellen werden durch getypte Graphen beschrieben, die zusätzliche Strukturen zur Darstellung von Vererbungs- und Enthaltenseinsbeziehungen (Containment) bereitstellen. Gerade diese zusätzlichen Strukturen führen im Formalismus zu einer Reihe von Nebenbedingungen, denen auf geeignete Weise Rechnung getragen werden muss. So erlaubt das kategoriale Fundament in dieser Arbeit

die Transformation auf Modellverbänden so zu definieren, dass auch nach der Transformation einer Vielzahl miteinander verbundener Modelle die Bedingungen bezüglich Vererbung und Containment überall erfüllt werden.

Die Arbeit rundet die Implementierung der Konzepte in die Eclipse Werkzeuge CompoEMF und CompoHenshin ab, welche die Machbarkeit und Kohärenz zeigen.

This thesis is dedicated to my family in the first place which gave me the room for numerous weekends of work without complaining.

It is also dedicated to my grandparents in honor of their wholehearted absolute pride taken in all their grandchildren.

Acknowledgements

Special thanks go to my supervisor Gabriele Taentzer for her promotion and countless afternoon sessions of fun with category theory and the theory of graph transformation. I really enjoyed these sessions whereas sometimes they gave some extra gray hair to me. I also thank Reiko Heckel who immediately agreed to be my co-supervisor and who gave me a lot of valuable and encouraging comments.

Furthermore, sincere thanks are given to the following friends and colleagues for proof reading and their valuable comments. Christian Krause especially checked the formal foundation in Part I. Thorsten Arendt read the first two chapters of this thesis while the last three have been read by Sebastian Laue. The implementation part (Part II) was cross-read by Daniel Strüber and Tim Schäfer who also put great effort in the implementations.

Thank you all!

CONTENTS

Chapter 1

Introduction

Software engineers have to cope with a continuously increasing complexity in software development. A promising paradigm addressing this issue is model-driven software development (MDD). Here, the main artifacts are models being ideal means to master complexity by abstraction and thus enable developers to focus on application-specific aspects of a software system. All implementation details are added by a code generator which is a highly reused part of a model-driven infrastructure.

1.1 Motivation

Model-driven software development has an increasing importance in software engineering. Nevertheless, the proceeding growth of software systems may yield very large models as well being possibly too large to be developed and maintained in a comprehensible way. In traditional software development, it is common practice to partition the target system into some kind of components and let distributed development teams work on each. This *divide and conquer* strategy does not only meet complexity related concerns but also offers the chance to work on components concurrently.

The question arises how this strategy can be applied to model-driven development, i.e., how models can be treated by teams being physically distributed and dealing with different logical aspects of software systems. An obvious idea is to set up a central repository for models which can be used by all teams. This solution is straightforward to implement. However, central repositories support the work of physically distributed modelers who still deal with logically undistributed models. This is not always adequate: for instance, in model-driven open source development, software components may be developed by truly independent teams. In this case, composite models where each team is working on its model component would be more adequate. Furthermore, designer teams might not want to expose their full model to everybody for a

couple of reasons. On the one hand, they may want to hide sensitive information on model level and indeed information hiding is a well-known and highly appreciated concept in the object-oriented paradigm. On the other hand, a small but essential subset of the original model may ease to carry-over parts and reuse them. Consequently, a rather modular approach is needed here.

Being the central artifacts in the model-driven development, models are subject to direct model modifications. They are usually performed by model transformations, one of the essential activities in MDD. As editing operations, model transformations are naturally performed directly on the model, called in-place. As for a set of distributed models, the question arises anew: how can such models be transformed in a distributed manner? And since a set of distributed and interconnected artifacts is always a synchronization challenge, how can they be transformed in a coordinated way? Obviously, according to a modular modeling approach, a transformation concept is needed able to deal with distributed interconnected models.

In the modeling community, the Eclipse Modeling Framework (EMF) [27, 74] has evolved to a well-known and widely used technology. EMF provides modeling and code generation capabilities based on so-called structural data models. As they describe structural aspects only, they are mainly used to specify domain-specific languages. EMF complies with Essential MOF (EMOF) as part of OMG's Meta Object Facility (MOF) 2.0 specification [63].

Interestingly, EMF does already support a plain model distribution mechanism where each model element can refer directly to elements contained in a remote resource, i.e., stored elsewhere. This concept is fine as long as an undistributed model is expected since technically arbitrary connected EMF models appear as a single model in memory. Moreover, each element can be referred to in an unstructured manner which is not always desired and which opposes to the concept of information hiding.

There are a number of in-place EMF model transformation approaches available too, e.g., Kermeta [51], EWL [55], EMF Tiger [9], Moment2 [13], and Henshin [37, 3].

Altogether, this sounds like a promising base to build up on.

1.2 Goals

The central goal of this thesis is the development of a concept for composite EMF modeling which is in particular formally funded. Composite EMF modeling subsumes here, on the one hand, a modularization concept which allows for a logical distribution of EMF model parts into interconnected EMF components models. EMF component models shall be connected in a structured way, i.e., the connection between two EMF models shall only be established if they are compatible with each other. In the following, such composed EMF component models are called *composite EMF models*.

On the other hand, composite EMF modeling also subsumes a transformation concept for composite EMF models, called composite EMF model transformation, which appropriately handles synchronization issues. This includes a closer consideration which synchronization issues may occur at all and how to tackle them.

Both concepts, the composite EMF models and composite EMF model transformation, deserve a formal foundation to allow for reasoning and analyzes. The reasoning shall especially assure that the transformation of distributed EMF models performs in a consistent way, i.e., that transformations yield composite EMF models again. Further elaborations on analysis are beyond the objectives of this thesis but are, however, enabled by the formal foundation.

As proof of concept, this thesis shall also comprise a prototypical implementation of composite EMF models and a tool dedicated to their transformation.

1.3 Main Results

The main result of this thesis is a concept for composite EMF modeling which allows for the logical distribution of EMF models and their transformations. Composite models consist of component models each being equipped with a number of export and import interface models. Export interfaces specify model parts that are provided to the environment while import interfaces specify model parts being required. A composite model is build up from component models being connected via their interfaces. The explicit declaration of interfaces allows for an independent definition of component models such that they can be connected later. Furthermore, explicit interfaces support information hiding in a straightforward way since model elements are either shared explicitly or not at all.

The composite EMF modeling approach is given twofold, i.e., a formal foundation is provided as well as tool implementations.

Formal Foundation. To provide a formal foundation of composite EMF modeling with explicit import and export interfaces, a number of approaches are combined. Since graphs are an ideal means to represent the underlying structure of (visual) models, the composite modeling approach is based on graph theory and algebraic graph transformation [23]. EMF model specific structural properties such as inheritance and containment edges are captured by additional dedicated graph structures yielding graphs with inheritance and containment structures. This is essentially incorporated from the work of Biermann et al. [11]. The notion of distribution is added by taking up and adapting the concepts of distributed graph transformation in [78]. However, in [78] Taentzer does not consider explicit import and export interfaces which are fundamental to the composite modeling approach as proposed in this thesis.

As a key concept in the modeling world, meta modeling is taken into account in the formalization as well yielding typed composite graphs with inheritance and containment. For ordinary graphs this approach is quite common [20]. However, the approach of the present thesis uses this technique also to predefine possible shapes of compositions, i.e., possible interrelations between interfaces are specified on meta level already. This approach is believed to be a straightforward continuation of the ordinary meta modeling concept when applied to model composition.

The formalization of typed composite graphs with inheritance and containment as well as their transformations is based on category theory. Formal reasoning shows that composite graphs and graph morphisms form a category which is later used to define composite graph transformations as double pushouts in this category. Properties of the category of typed graphs with node type inheritance are shown by Hermann et al. in [38] and are reused in this context. EMF related properties concerning containment edges, however, impede the general existence of pushouts (see also [11]). Therefore, a restricted form of rules is defined, called consistent rules, and it is shown that the application of consistent rules to composite graphs lead to composite graphs again. With the help of [26] it is shown that composite graph transformations can be constructed component-wise.

The most recent enhancement, called weak composite models, allows export interfaces to occur without their bodies. This underlines the vision of truly independent component models in the sense that components may be really hidden somehow except of their export interfaces. It is additionally shown, that the correspondingly defined transformation can be led back to ordinary composite graph transformation.

Implementation and Tooling According to the main formal results, tool support has been developed.

A tool called CompoEMF [73, 72] comes as Eclipse [22] plugin and represents a prototypical implementation of the formal concepts of composite EMF models. Its main intend is high reuse of existing EMF models, i.e., ordinary EMF models may easily become parts of component models. Interface data are consequently stored in separate interface models, being EMF models as well. While the formalization of typed composite graphs with inheritance and containment structures does not explicitly consider attributes, CompoEMF supports attributes anyhow in a coherent way, i.e., sharing attributes requires their explicit export and import. In practice, the access of imported attributes leads to a delegate request along the interface chain in order to gather the actual attribute value. CompoEMF is also shipped with a tree-based editor and a graphical editor which support the user with features beyond basic CRUD operations.

Composite EMF model transformations are implemented by CompoHenshin [76, 75], a tool which builds up on the transformation engine of Henshin [37, 3]. It exploits the fact, that composite model transformations can be implemented as a number of coordinated local model transformation. A dedicated editor supports the creation of composite model rules. Their application is supported by means of the CompoHenshin API yet.

CompoEMF and CompoHenshin are prototype implementations of the formal concepts. While they leave a number of improvements open, they already show the coherence of the concepts.

Note that none of these tools provide implementations in accordance to the recently elaborated weak composite graphs.

1.4 Related Publications of the Author

This section gives are short summary of own publications strongly related to this thesis.

In [44], a classification of component concepts is given whereas the one with explicit import and export interfaces is focused on. It especially sketches composite models with explicit and implicit interfaces using concepts of distributed graph transformation and outlines different kinds of composite model transformations.

While [42] outlines the thesis project in general, the concept of typed composite graphs with explicit import and export interfaces is formalized in detail in [45]. These definitions are further improved in [46] which also provides more technical considerations towards composite graph transformation. Finally, the journal paper [47] formalizes the transformation of typed composite graphs with inheritance and containment structures. This thesis continues that research and clarifies the conditions a match has to satisfy to perform composite transformations. This includes the definition of pushout complements in the category of composite graphs and morphisms. Moreover, the concepts of weak composite graphs and their transformation are new and published with this thesis at first.

In the meantime, [3] introduces Henshin, an official Eclipse incubation project and tool for EMF model transformation, at which the author participates as initial committer.

1.5 Organization of the Chapters

The remainder of this thesis is structured as follows (compare Fig. 1.1): Chapter 2 gives an informal overview of composite modeling. This includes a general survey of existing composition approaches as well as an informal introduction into composite EMF models with explicit import and export interfaces and their transformation. This explicit structuring of component models is not

trivial and deserves careful considerations of the engineers. To this end, a structured design process may help which is also outlined in that chapter. Last, composite modeling is further motivated by some examples where one of them is the running example throughout the thesis.

Formalizations and implementations are placed in separate parts, Part I and Part II, respectively. Part I starts with a general introduction to algebraic graph transformation in Chap. 3 whose basic definitions are extended by inheritance and containment structures in Chap. 4 to capture specialties of EMF. With the local graph structures being properly defined, Chap. 5 goes for their composition by distributed graphs. So far, only simple graphs are composed. Chapter 6 then combines all concepts and introduces typed composite graphs with inheritance and containment and their corresponding transformation.

Part II introduces the implementation and tooling for composite EMF models, first. For sake of a fair judgment of CompoEMF, essentials of EMF are presented beforehand. CompoEMF is then described according to its implementation and editor support. CompoHenshin is presented in a similar way: At first, Henshin is outlined stand-alone which shall serve as a good basis for the subsequent introduction of CompoHenshin.

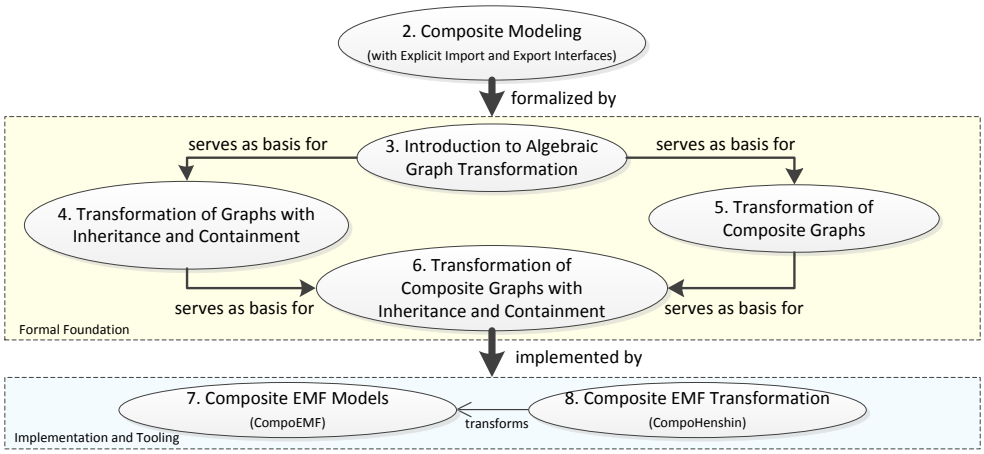


Figure 1.1: Illustration of the main content of this thesis.

A discussion of related work and concluding remarks can be found in Chap. 9 and Chap. 10, respectively.

Chapter 2

Composite Modeling

The model-driven development of complex software systems rapidly leads to large models. An intuitive way of mastering complexity is to develop a number of smaller models being interrelated somehow. Such a modularization facilitates common desired properties, e.g., the readability and understandability of models, their reusability within as well as beyond a software system, and the flexibility to develop them independently by possibly truly distributed developers.

Section 2.1 presents an overview of state-of-the-art approaches of model composition. It especially becomes apparent that, so far, model composition research solely focuses on techniques in order to merge or integrate individual models. In contrast, the approach proposed in this thesis rather elaborates on the independent and consistent development based on interrelated component models, called composite models. To this end, composite models with explicit import and export interfaces are introduced whose modifications are described by a corresponding model transformation approach. They are presented in Sec. 2.2 and Sec. 2.3, respectively. Note that the remainder of this chapter and this thesis consider composite modeling with explicit import and export interfaces only.

Obviously, the power of being able to define arbitrary interrelated component models comes with a great responsibility. Developers are required to early define which kinds of components shall exist and how they shall be connected. This deserves a general process for collaborative and distributed modeling. Therefore, Sec. 2.4 sketches a possible process which, however, requires further elaboration and work in the future.

Last but not least, in Sec. 2.5 example scenarios for composite modeling are presented at which one of them is to be the running example throughout this thesis.

2.1 Composition of Models: State of the Art

The composition of models is generally characterized by multiple component models and interrelation between the models describing some sort of overlapping or correspondence. In the following, a classification of model composition approaches is given [44]. Afterwards, a selection of existing approaches and tools are outlined.

2.1.1 Classification

For an explicit consideration of composition concepts for models, the composition structure, also called *network structure*, is considered separately from the internal structure of models, called *object structure*. Figure 2.1 illustrates a general classification on network level in three schemes.

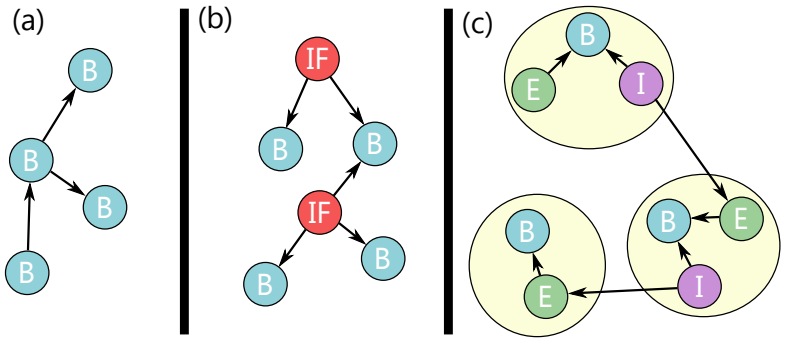


Figure 2.1: Schemas of component models with implicit interfaces (a), with common interfaces (b), and with explicit import/export interfaces (c).

Component models with implicit interfaces are illustrated in (a). All models are interpreted as so-called body models (`B`) which carry the data of a certain domain. Elements in a body model may directly refer to arbitrary elements in other body models. This technique does not need intermediaries/interfaces to connect with each other and thus can be considered efficient, i.e., there is no overhead by additional interfaces. Nevertheless, it is also very plain since this kind of composition is possible only if all elements are visible to the environment. Due to the lack of interfaces, modularization is supported rather physically. That means, the kind of interrelation between models may be arbitrary and is not limited somehow in favor of the definition of logical components. Furthermore, important modularization concepts like information hiding are not supported. With regard to interfaces, the elements referred to by the source model can be considered as implicit import while they constitute the implicit export interface in the context of the target model.

Component models with common interfaces as depicted in (b) can also be described by bipartite network graphs. Network nodes are either body nodes (`B`) or interface nodes (`IF`) at which network edges run from interfaces to bodies. This structure describes component models whose corresponding objects are identified over a common interface. While interfaces are explicitly given, again information hiding is not supported, i.e., all elements are accessible from the outside. The notion of a component is still vague since interfaces may connect an arbitrary number of body models and body models might be connected by more than one interface. The usage of additional interface models yields some overhead.

This schema is the most common and does often appear in conjunction with the integration of uncoupled models which then yields a new single assembled model. In the past, the research on model composition especially focuses on this topic. The integration is usually done in two steps: First, some kind of matching is performed which finds or completes correspondences between models. Correspondences may be kept in form of an interface model. The matching is often specified over a dedicated composition language not being related to the actual domain. Second, models are merged with respect to the correspondences found. This is often implemented by model transformation or at least can be viewed as such [6]. Prior to the integration, however, models are usually treated as individuals without any correlation. Apparently, this may easily lead to inconsistencies.

Component models with explicit import and export interfaces have received less attention in the modeling community yet. Their general schema is illustrated in (c) where each component is constituted by exactly one body model (`B`) and an arbitrary number of import (`I`) and export interface models (`E`). Exports identify those elements of their body which are provided to the environment while imports identify those being consumed. Accordingly, components can only be connected via import-export relations. This schema is the only one providing an explicit definition of components and offering information hiding since all elements to be exposed have to explicitly occur in the export interface. Due to the interfaces which serve as kind of facade [31], developers are enabled to work with their models independently. Obviously, interfaces have to remain stable but the developer is aware of existing interfaces and what parts of the component may (or may not) be safely modified. Anyhow, export interfaces may be extended and import interfaces may be reduced without running into inconsistencies. The major shortcoming of this approach is the comparatively high overhead of additional interface models which can be, however, compensated to some extend by interfaces being simpler than their body.

2.1.2 Survey of Existing Approaches

The Eclipse Modeling Framework [27] is a famous representative of schema (a). EMF models can be structured in separate resources (mostly files) which may be interconnected by so-called remote references, i.e., references from local model elements to remote model elements. The look-up of remote model elements is realized by proxy model elements which resolve remote references on demand. EMF supports meta modeling, i.e., meta models and instances over them can be created. The physical distribution may take place on both meta level and instance level. See [74] and Sec. 7.1 for further details on EMF.

One of the popular tools of schema (b) is the EMF based *Atlas Model Weaver* (AMW) [7, 2], an Eclipse project at incubation state. It allows establishing relationships between elements of different independent models yielding a weaving model, the interface model. Linking model elements can be performed manually and alternatively be inferred by means of a match operator. Weaving models must conform to an extensible weaving meta model which declares permitted relationships. The term *weaving* model does already indicate that related models are to be woven into one big model. This is done with the help of the ATL Transformation Language (ATL) [40, 5], a model transformation language, toolkit, and also an Eclipse project. Note that the weaving process does not destroy the original models but creates a new one.

Kompose [29, 30] is a generic model composition tool belonging to schema (b) as well and also focuses in model integration. It exploits so called signatures to automatically infer relationships between model elements. In fact, signatures are property values of model elements involved. Kompose is built on top of Kermeta [51] being a meta modeling environment.

The *Modeling Aspects Using a Transformation Approach* approach, short MATA [82, 81], follows the aspect-oriented paradigm on model level. That means, besides a base model there is an aspect model which specify crosscutting concerns in the targeted software system, however, in an independent manner. In contrast to *AMW* and *Kompose*, the composition of base model and aspect model is specified by model transformation rules whose application leads to an integrated model. Transformation rules do not only allow the definition of direct correspondences but may describe more complex correlations, e.g., between various elements. Rules may also introduce new elements in the integrated model which do not occur in the base model nor in the aspect model. Considering the set of model transformation rules as interface, MATA is a special instantiation of schema (b).

An approach more related to the management of component models than to their integration is *Virtual EMF* [18, 17], a young project for the virtual composition of EMF models. In terms of schema (b), the interface model is represented by a so-called virtual model which is, roughly spoken, the union of selected elements of the related models. What is remarkable about Virtual EMF is that the virtual model is not a real model but simulated by the tooling

where request for access and manipulation are redirected to the original models. As a result, the user interacts with the interface models rather than with the body models. Since virtual models may refer to any element in the original models information hiding does not take place. Furthermore, since models may participate in several virtual models, inconsistencies may occur. Unfortunately, Virtual EMF also lacks of a theoretical foundation.

Kelsen and Ma propose another interesting composition technique in [50] basically belonging to schema (b). Body models and interfaces models are represented by so-called modules and fragmentation models, respectively. In addition, a module can be understood as the assembled model resulting from the composition of modules via a fragmentation model. Since composition cycles are forbidden, modules establish a hierarchy. This approach, however, incorporates schema (c) in the following sense: In order to introduce information hiding, modules are additionally equipped with an interface description. This description specifies which nodes are visible to the environment and how they can be accessed by fragmentation models. Nevertheless, this technique is all the same a composition technique focusing on the integration (and decomposition) of models. Operations to allow for a consistent editing of connected models are not captured in [50]. Furthermore, this concept does not provide tooling yet. Each module and the assembled model have to conform to the same meta model. Note that this approach provides a formalization which proofs that the assembled models also conforms to the meta model of the modules.

To the best of the author's knowledge, except for the concept propose in this thesis (see next section), there are no other concepts available for models which declare import and export interfaces explicitly (cf. Fig. 2.1 schema (c)). In the following, composite models with explicit import and export interfaces are considered only.

2.2 Composite Models with Explicit Import and Export Interfaces

In the previous section, the concept of component models with explicit import and export interfaces has already been outlined. Composite models are constituted by interconnected component models where each component model consists of exactly one body model and an arbitrary number of export and import interface models. Body, import, and export are also called component parts. The body model represents the actual data of a component, export interfaces identify those parts of the body being visible to the environment, and import interfaces identify those parts being required from other components. Interconnections between components are exclusively established by connecting an import interface with an export interface. Needless to say that interfaces must be compatible in order to be connected. As a consequence,

the development of modular systems is not only supported but enforced. In contrast to other approaches, interconnections are understood as sharing information with each other rather than establishing correspondences. The amount of information to be shared must be explicitly defined in the interfaces of each component. This enables information hiding.

Figure 2.2 illustrates a composite model with explicit import and export interfaces comprising two interconnected component models. They are shown on network level and object level. The network level represents the topology of a composite model, i.e., components are represented by nodes (`Body`, `Import`, `Export`) and relations between them. Dashed edges denote interface-body relations while dotted edges denote relations between import and export interfaces. As a kind of refinement of network nodes and edges, the object layer represents the actual models of the component parts and the mappings between model elements, respectively. Intuitively, mappings on object level must be compatible in the sense that source and target nodes of mapped edges must be mapped to source and target of the image edge.

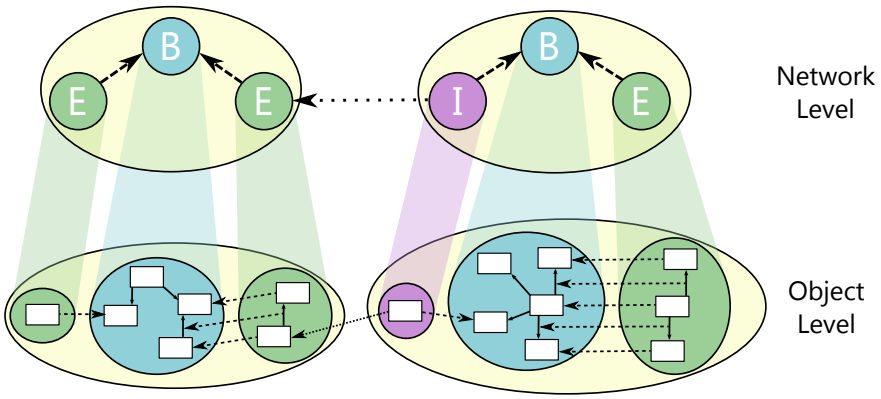


Figure 2.2: Composite model with explicit import and export interfaces on network level and object level.

Note that the element in the import interface is connected to an element in the export interface. Cases where import elements are not connected to export elements are conceivable but may denote some kind of inconsistency. The approach proposed in thesis requires complete mappings, i.e., on the one hand, all elements in interfaces must be connected to a corresponding body element, and on the other hand, all elements in import interfaces must also be connected to export elements.

2.2.1 Meta Modeling

The present approach focuses on object-oriented model structures at which meta modeling is a common technique. Meta models provide an abstract description and generalization of the structure of possible instances, i.e., an instance model conforms to its meta model if it satisfies the structural constraints given. To distinguish model elements, nodes and edges in meta models are generally called classes and associations (or relations) while their counterparts in instance models are called objects and links.

Composite models with explicit interfaces support meta modeling as well. Figure 2.3 shows a simple configuration on network level with a composite model as meta model specified above the line. Contained components may be considered as component meta models. Below the line, a possible instance conforming to this composite meta model is given. As usual in the Unified Modeling Language (UML) [65] and adapted here, the typing of network nodes is given right after the colon. The typing on object level follows from ordinary meta modeling and is omitted here in order to focus on structural aspects of components. Note that each component meta model provides its own meta models, one for each of its component parts.

The example instance shows that instances of component meta models are free to instantiate an arbitrary number of interfaces as long they conform to the interface meta model. One component instance at the left decides not to have an interface at all while the component instance at the right instantiates even two import interfaces. Each of these import interfaces points to an export interface. In the present approach, imports must point to exactly one export. However, approaches are also conceivable which allow imports to point to multiple exports or to no export at all. Export interfaces, on the contrary, are allowed to be referred to by arbitrary many import interfaces.

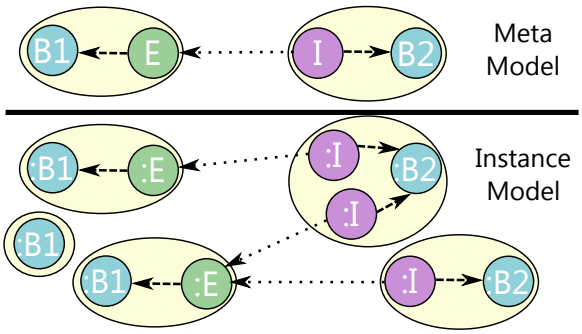


Figure 2.3: Meta modeling with composite model with explicit import and export interfaces.

2.2.2 Inheritance and Containment

The present approach focuses not only on object-oriented model structures but in particular on Essential MOF [62] based modeling technologies like EMF [27]. On meta level, they offer features beyond simple classes and associations. Prominent aspects are attributes, inheritance including its notion of abstract classes, and containment structures. Inheritance relations have no correspondences on object level and describe a parent-child relationship or sub-typing. Instances of abstract node types are forbidden. A containment relation between two classes defines an ownership relation between instances of them: a class B being contained by class A means that objects of type B must not exist separately but each of them has to be contained by exactly one object of type A. Containment relations are of particular meaning to EMF models (see 7.1 for details). Under the conditions that there is a single root object containing all other objects transitively and that containment is acyclic, the overall containment structure spans a tree that allows the serialization of instance models in a structured manner.

Incorporated into the concept of composite models with explicit interfaces, inheritance offers an interesting feature: Import and export interfaces may have simpler inheritance structures and do not need to expose the complete inheritance structures of their component bodies. Attributes are treated analogously to nodes and edges, i.e., if they shall be shared they have to explicitly occur in interfaces. Since containment edges have a special meaning, the present approach of composite models requires them to be mapped to containment edges again.

Fig. 2.4 shows a simple meta model (left) and instance model (right) of a component on object level. The component meta model consists of a body which carries a structure using inheritance. Complying with standard UML notation, containment edges are denoted by diamonds at the containers end and inheritance is denoted by a rectangle at the parents end. Italic letters, e.g., `B`, indicate an abstract class. The export interface exploits that in the body the classes `C` and `D` inherit the containment edge to `E` such that it offers the containment directly in its interface. Especially note that the two containment edges in the interface are both mapped to one edge in the body. A straightforward example component instance model at the right of Fig. 2.4 shows objects and links. Object typing is denoted after the colon while link typing can be uniquely deduced and is not denoted explicitly here. Please refer to Sec. 2.5 for concrete examples.

Further features, e.g., multiplicities, operations, and enumerations are not considered in the present approach as this thesis especially focuses on structural aspects of the modularization. They belong to future work.

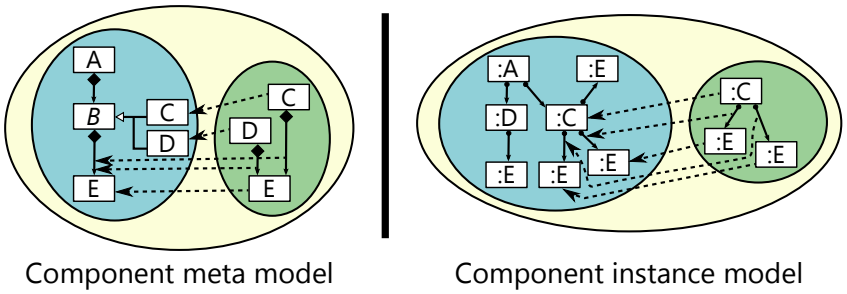


Figure 2.4: Component meta model with inheritance and containment structures (left) and an instance (right).

2.3 Transformation of Composite Models with Explicit Interfaces

Model-based development and especially model-driven development heavily rely on model transformations. Multiple interconnected models, however, deserve a more sophisticated transformation technique in order to master the synchronization challenge and avoid incompatible mappings. For instance, consider the cases where bodies are modified without adapting related export interfaces and where exports are modified without taking dependent import interfaces into account.

Independent of any concrete model transformation approach, this section considers composite transformations as partial mappings of composite models. That means, one composite model is partially mapped to another one. The source composite model describes the initial state while the target composite model describes the resulting state. They can describe the major effects of model transformations which can be the creation, deletion or update of model elements and their references.

In [44], a classification of composite graph transformations is given which directly corresponds to composite model transformations. Four kinds of composite transformations are distinguished with respect to their impact on the component models and the topology. *Internal transformations* run in bodies only and do not change any interface. *Component transformations* are especially concerned with manipulations of single components, i.e., body transformations with interface adaptations. Such transformations do not consider any impact on connected components, in contrast to *synchronized transformations* which consist of several parallel component transformations. They allow actions like modifying export interfaces and adapting connected import interfaces accordingly. The transformation kind with the highest possible impact on composite models, *(network) reconfigurations*, may change the overall composite network structure which in turn often requires the adaption of already existing components. In the following, each transformation kind is discussed

and visualized. Note that this is done in an abstract way for clarity, i.e., meta modeling is omitted.

Internal transformations are the kind of composite transformation not having any impact on adjacent components. Actions to be considered are the creation of nodes and edges in bodies as well as arbitrary actions to existing elements in a body that are not part of an interface.

Figure 2.5 shows the transformation of two interconnected component models which are shown on the network level and on the object level as already introduced. The transformation reveals a modification in the body model of the right component only (see bottom right in Fig. 2.5). In detail, the body model is extended by two nodes and two edges. As this has no effect on the interface, this transformation is obviously to be classified as internal transformation.

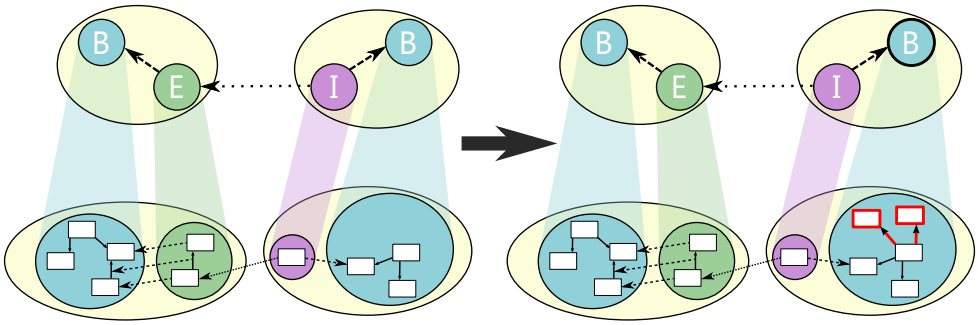


Figure 2.5: Example internal transformation.

Component transformations allow all kinds of actions inside one component including its interfaces that do not affect other components. Apparently, internal transformations are a special case of component transformations since they do not change interfaces at all. Further actions are the deletion of model elements from import interfaces and their creation in export interfaces. These actions do not require any adaption of connected interfaces.

An example component transformation is shown in Fig. 2.6. The right-hand side shows the concurrent extension of the body model and its export interface with additional nodes and edges. The transformation modifies only a single component and has no effect on the right component.

Synchronized transformations overcome the limitations of component transformations and allow performing necessary adaptations of connected components. Again, this transformation kind includes the ones introduced above.

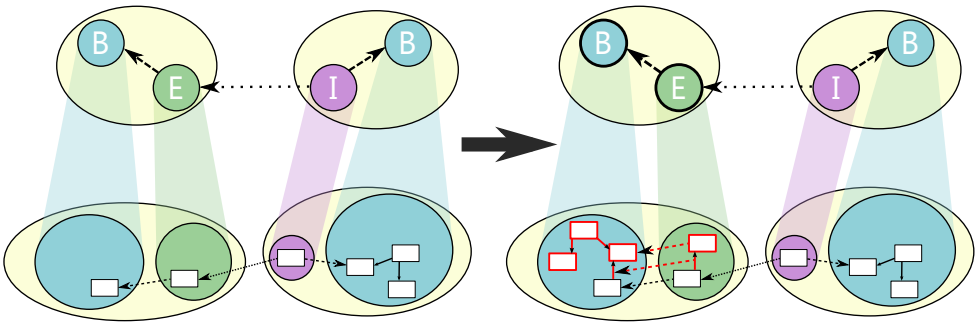


Figure 2.6: Example component transformation.

The transformation in Fig. 2.7 depicts an extension of the right component's body and import interface. Since import interface elements shall always point to a corresponding export interface element, the synchronized extension of an export interface is required. This transformation performs this concurrently as well. The export and the body of the left component are also extended by new elements. A mapping from the new import element to the new export element is added, too.

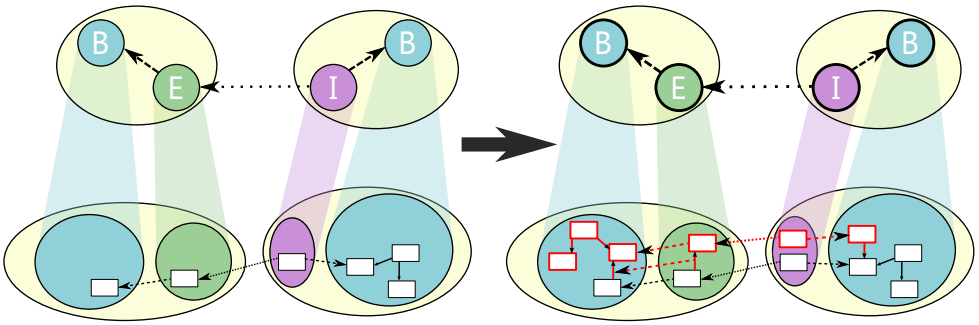


Figure 2.7: Example synchronized transformations.

Reconfigurations may also change network structures in contrast to the transformation kinds above. This includes actions such as the creation of component parts, the deletion of component parts, and the creation/deletion of import-export relation. Such changes may require various adaptations of adjacent component parts which can be considered as synchronized transformations on preserved network nodes. Consequently, synchronized transformations form a subset of reconfigurations.

In Fig. 2.8, the transformation shows a substantial change in the composite model. A new component occurs including its mandatory body, an import

interface and the connection to an export. Analogous changes happen on object level.

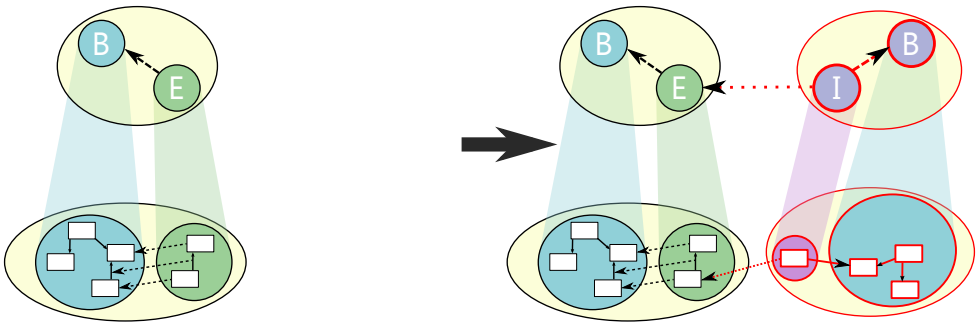


Figure 2.8: Example (network) reconfiguration.

2.4 Towards a Composite Modeling Process

When lifting model-driven development to a distributed setting, a couple of challenges arise: different possible starting points are conceivable, i.e., questions matter like "Do models already exist?" and "If yes, is their nature rather monolithic, i.e., they have not been sufficiently modularized yet?". In the answer of both questions is yes, then well-defined modularization strategies have to be applied. Furthermore, contributors at different locations might be responsible for models that are interconnected in some sense. Thus, clear conditions and conventions for the editing of models are required to avoid the emergence of inconsistencies. Since model-driven development always involves some kind of generation, e.g., code generation, the question arises how composite models can be used as a blueprint for generation facilities later on.

Modularization. Adopting the modularization paradigm to monolithic models deserves a structured procedure of splitting. A splitting can be performed along horizontal and vertical facet: The former captures cases where several domains are represented in a single model; each of which shall then be sourced out in separate domain-specific models. Vertical facets are logical units within a single domain which may be suitably placed in separate models as well. In how far one or the other splitting strategy can be (semi-)automatically performed remains up to future work.

Note that the splitting has to be performed on both meta level and instance level. This is a particular challenge which extends into the research topic of meta model evolution.

Distributed model editing. A crucial challenge of collaborative editing is to preserve the consistency of models while keeping the editing steps as independent as possible. Existing collaborative model editors such as Papyrus [33] and MagicDraw [58] provide an asynchronous approach to editing: a model can be displayed and modified in multiple distributed editors at once. Furthermore, model parts may be exclusively locked to perform own modifications and prevent the editing by other developers in the meantime. As for the use of composite models, composite model transformations are well suitable to also support synchronous editing steps: for instance, consider the construction of two related components in parallel. Asynchronous and synchronous steps correspond to the classification in Sec. 2.3, i.e., asynchronous steps refer the class of *component transformations* and synchronous steps mean *synchronized transformations* and *network reconfigurations*.

Another aspect of distributed model editing concerns security. That means, how do responsibilities and ownership relate to visibilities of component models and their interfaces and permission for certain editing operations.

Generation. Aiming at providing a full life-cycle of model-driven development, code generation semantics for composite models has to be considered. For this purpose different code generation strategies are conceivable. A distributed code generator might allow the successive code generation for each individual component model. This may yield multiple separate collections of component code which may then be merged together to a whole software system. A centralized code generator might perform code generation for the composite model as a whole. This may include some kind of assembling of all component parts to a whole beforehand. It is also conceivable, that a central code generator accesses all component and interweaves the generate code on the fly. However, as for composite models proposed in this thesis, no such generator exists and is future work.

2.5 Example Scenarios

New concepts of composite EMF modeling are now motivated at three different model-driven development scenarios. At first, graphical editor development based on GMF [34] is illustrated which uses a set of domain-specific languages in order to specify graphical editors. Afterwards, model-driven web application development is illustrated which comprises the development of a domain-specific language for web applications. Last but not least, the model-based development of business components by means of a department management component and a project management component is shown. This last scenario also serves as running example throughout Part I and II to illustrate formalisms and tooling.

2.5.1 Graphical Editor Development

The Eclipse Graphical Modeling Framework (GMF) [34] offers the development of graphical editors in a model-driven way using EMF models. It comprises a family of modeling languages concerned with different aspects of the targeted visual language. Figure 2.9 shows the so-called GMF Dashboard which illustrates the GMF approach twofold: On the one hand, boxes represent the models necessary to generate an visual editor. On the other hand, arrows between the boxes denote the general workflow.

GMF models and workflow. As indicated by the name, the *domain model* contains the language description of the target domain of the editor. Starting from here, one can automatically derive a *domain generator model* (Domain Gen Model) actually belonging to EMF for model code generation purposes. Furthermore in a semi-automatic manner, a *graphical definition model* (Graphical Def Model) can be derived which specifies visual elements of the editor. This can also be considered as the alphabet of the editor's visual language . The *tooling definition model* (Tooling Def Model) declares which items shall appear on a graphical action palette. Domain model, graphical definition model, and tooling definition model are altogether completely independent yet, i.e., they do not contain references to each other. They are all combined by the *mapping model* which carries links pointing directly to elements of the other models. Given such a mapping model as input, a so-called *diagram editor generator model* can automatically be generated being the input for the final generation process of the editor code.

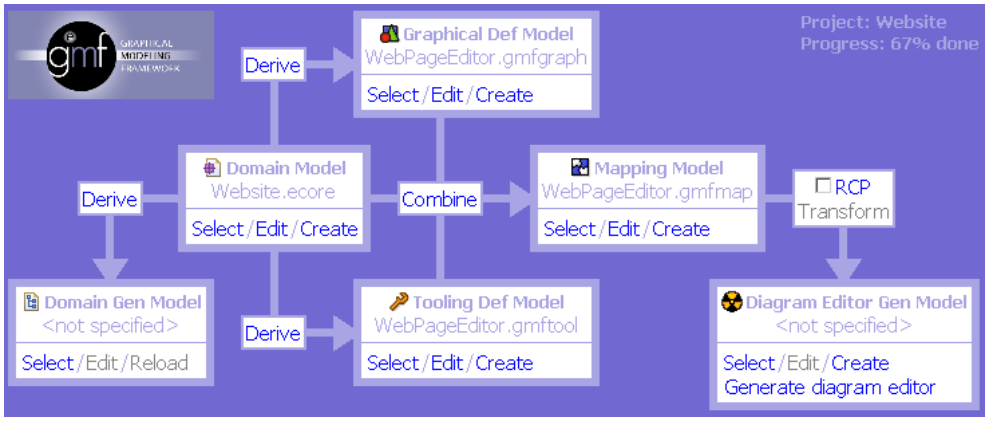


Figure 2.9: GMF Dashboard.

Challenges. In this setting, there are especially two issues which call for composite modeling to be tackled.

First, links between models are created directly and model changes usually occur independently. Needless to say that this easily leads to inconsistent models and GMF is actually very sensitive to inconsistencies. This is also subject of research by other groups. In [68], the authors describe an model evolution process for the GMF based editor development. They analyze model changes and try to deduce co-evolutional editor operations in related models. In contrast, composite modeling as proposed in this thesis follows a more natural "co-evolution" approach where composite model transformations modify all related models concurrently and thus enable to define consistency preserving model changes.

Second, composite modeling is even more suitable with regard to *components* in a GMF based editor development. Assume the case where a domain modeler is responsible for mapping the application domain to the *domain model*. In addition, there are editor designers who develop domain-specific editors showing the domain model in one or more different views. Composite models would then support the decoupling of domain model and editor-specific models by specifying them as two components being interconnected by interfaces. This is outline in [44] by means of a graphical editor for website development. It further presents concrete composite model transformations according to the classification given in Sec. 2.3.

Note that EuGENia [54] meets these issues by enabling the domain model to be equipped with annotations regarding to the desired visual representation. A special generator then infers appropriate GMF models which are always consistent to each other. However, this approach does not support separate developer teams, e.g., one team for the domain model and one team for the editor. Moreover, supplied annotations only capture a very subset of all features GMF actually offers.

2.5.2 Web Application Development

Following the main idea in the previous section, a specification language for simple web applications shall be developed now.

Initial meta model. Figure 2.10 shows the abstract syntax of such a language in form of a meta model depicted in a UML complying class diagram. Attributes are fully omitted in order to concentrate on structure aspects; however, meaningful attributes are obvious, e.g., for names. The class `Website` serves as root element and contains a number of entities (`Entity`) and web pages (`Page`). Entities represent some kind of data to be shown at web pages and may carry attributes (`Attribute`) and references (`Reference`) to other entities. Web pages can be interconnected, i.e., a page may contain links which trigger the navigation to another page. In this simple scenario, pages can either be static (`StaticPage`) or dynamic (`DynamicPage`). While static pages contain fixed content independent of entities, dynamic pages present a list of available

entities (`IndexPage`) or a detailed view for on a specific entity (`DetailsPage`). Note that `Feature`, `Page`, and `DynamicPage` are abstract classes denoted by italic letters.

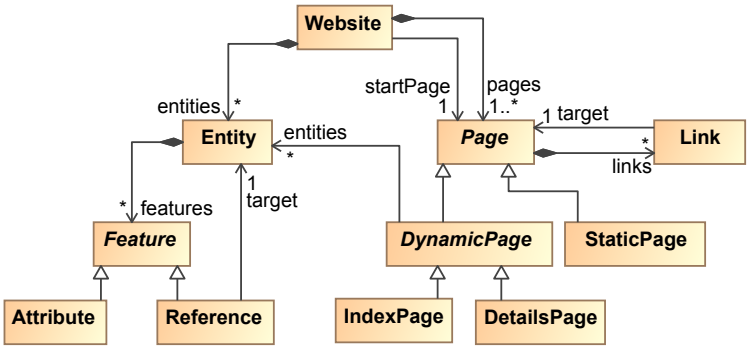


Figure 2.10: Modeling language for the specification of simple web application.

Apparently, two different domains have been encoded in this single model: It encloses the specification of data and the specification of interconnected web pages for data presentation. In favor of modularity, both shall be divided into separate but interconnected models.

Revisited composite meta model. Figure 2.11 shows a revisited meta model in terms of a composite meta model comprising two components [1]. Rounded rectangles wrap component parts and dashed arrows illustrate mappings between component parts. The left component represents the content component, i.e., the one that contains the actual data, while the site map component at the right is able to represent interconnected web pages. Each component body resembles the structure of its corresponding part in the original meta model. Note that the class `Entity` appears in both component's body models as it actually concerns both. The components are connected via their interfaces. In detail, the left-hand component is equipped with an import and export interface able to expose entities and also to let entities of other content models be consumed. On the right-hand side, the site map model imports entities to be presented on web pages. In addition, even web pages of other site map models may be connected. Concrete examples follow below.

Note at this point, that associated attributes and references of `Entity` objects cannot be exported because the classes `Attribute` and `Reference` do not occur in export interface *Export1.1*. They are consequently unknown in site map models. However, in this scenario composite modeling aims at a modularization of the original large model only while expecting instances of the component meta models to have full knowledge of each other. This is

[1] The new components may result from any kind of decomposition algorithm applied to the large original (meta) model.

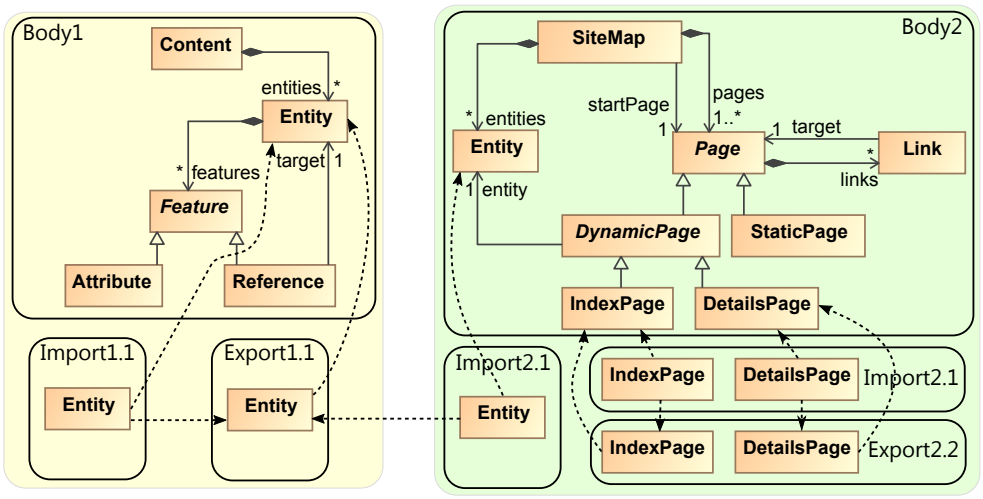


Figure 2.11: Composite (meta) model showing two components as parts of a modular language for the specification of simple web application (compare Fig. 2.10).

comparable with approaches where multiple models are merged at the end and interfaces are designed such that they describe overlaps. For site map models it is therefore assumed to be sufficient to refer to entities in order to gather remaining information if required. A scenario with truly independent components which also hide information is presented in the subsequent section.

Example composite model. An example instance of the composite meta model in Fig. 2.11 is given in Fig. 2.12. Related component part types are denoted by corresponding names within the component parts, e.g., in the top-most left and right the component parts denote to be a body each typed over `Body1`. Objects and links are given in a UML conform manner. Object typing is denoted after the colon; link typing is given with the corresponding type name at the end. Equal numbers in front of columns (only used in `Entity` objects) declare equality and are used in favor of traceability.

Figure 2.12 describes a website structure in the context of a conference. The content component in the upper right describes the ingredient of the website which allows to store data concerning authors and their belonging institution while the upper left specifies the paper related component. The latter particularly associates paper and author entities while the author entity is defined in the other component. Therefore, an export and import is required as shown for the `2:Entity` object.

Arranged at the bottom, there is also a component describing the web page structure. The starting page is a `:StaticPage` which refers to an `:IndexPage`

displaying paper entities and links allowing to look at details of a paper. In that `:DetailsPage` may also be links to have a closer look at details of the authors. Both paper and author entities are imported.

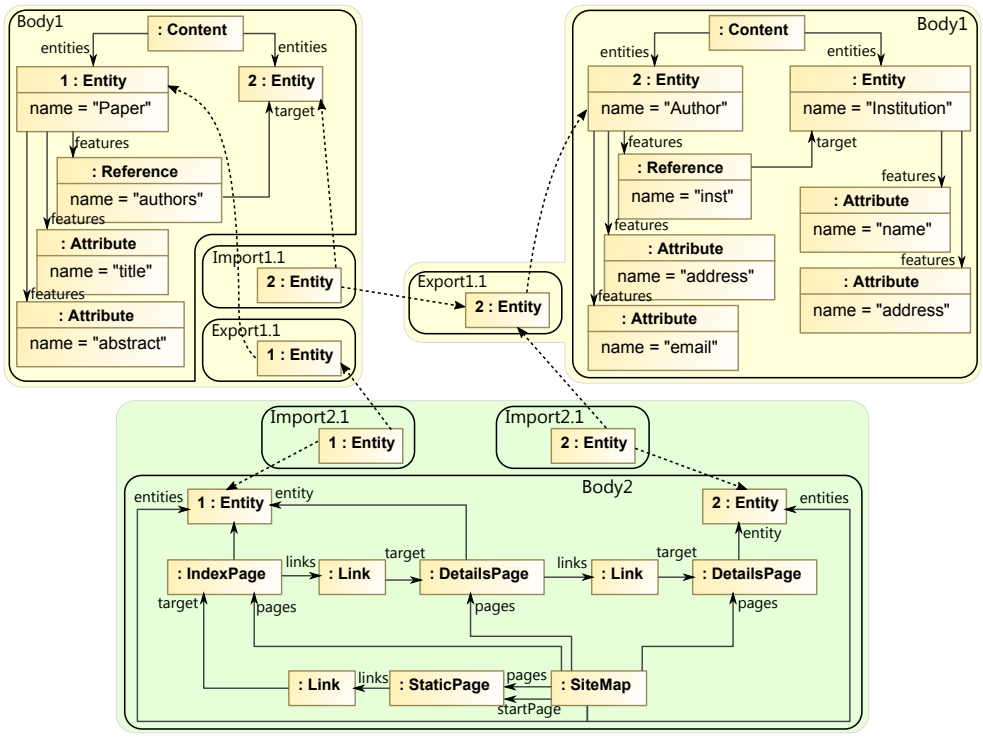


Figure 2.12: Example instance of the composite meta model shown in Fig. 2.11.

At the end, the developers might merge all three components along their interfaces and generate website code out of the resulting big model. Until then, however, each component can be developed independently of each other as long as the interfaces remain stable.

2.5.3 Development of Component-Oriented Business Application

The traditional development of business applications is often characterized be dividing the target system into software components, develop them concurrently and connect them somehow. In the following scenario, a similar approach is followed in the context of model-driven development. Note that this scenario constitutes the running example in this thesis.

Composite meta model. Consider the specification of two simple software components in Fig. 2.13. Both components serve as component meta models.

Body models are arranged at the top while their interfaces are located below. Again, attributes are omitted in favor of readability and to focus on structural aspects.

The component at the left depicts a department management component. The body `DepBody` specifies a department to contain employees at which employees may be assistants, managers, and ordinary employees. All employee sub-types inherit from `Person` which may provide properties such as surname, lastname and so on. Managers may also supervise a number of employees. The set of interfaces only contains export interfaces which are able to expose certain subsets of the body. `DepExport1` focuses on the department hierarchy while `DepExport3` is able to export any kind of employee. However, the latter exposes them without any interrelation. In contrast, `DepExport2` allows to export departments and their related employees and managers. Note in particular, that `DepExport2` does not simply resemble the structure of the body `DepBody` but even simplifies it. This is also shown by the dashed arrows where two of them run from the edges `employees` and `managers` in `DepExport2` to a single one, `employees`, in `DepBody`.

At the right, a project management component is illustrated. A project is accomplished by freelancers and employed members. Employed members are here employees of the company the component is deployed to. Interfaces of this component offer to build project hierarchies. In particular, import interface `PrjImport2` imports (information of) employed member by pointing to exported employee in the department management component. Note that another export would have served as well, e.g., pointing to the employee in `DepExport3`. It is rather a matter of if objects are exported on instance level. The meta level solely opens up the possibility to export or import objects.

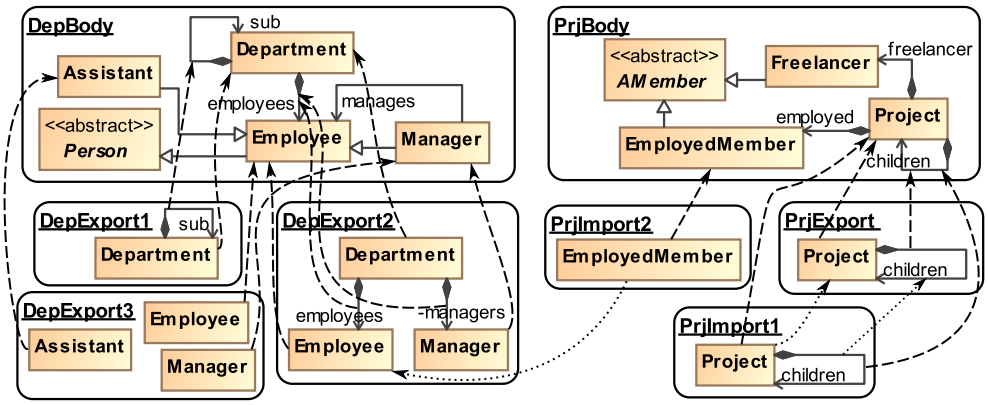


Figure 2.13: Composite meta models of a department management component (left) and a project management component (right).

Composite instance model. A composite instance model typed over the composite meta model in Fig. 2.13 is shown in Fig. 2.14. Rounded rectangles again specify component parts whose typing is denoted by the underlined text. Object typing is given after the colon while link typing is omitted this time to keep the diagram clear. However, their typing can uniquely be deduced. Note that names in front of colons are given for improved traceability (they might as well express the value of a name attribute.)

At the left, the component describes a department concerned with computer science whose head is the manager named Frank. The department is constituted by further sub-departments, software engineering and cloud computing. Exemplarily, the software engineering department also contains employees. The lower export interface partially exposes the department structure and the upper export interface exposes some selected objects of the body. The manager Diane and the department Cloud Computing are not exposed at all, i.e., they are still hidden towards other components.

The project management instance at the right shows a quality assurance project whose sub-project elaborated metrics. Each project is accomplished by some member whereas the employed member object named Felicia is an imported object and actually part of the department management component. Note that there are actually two project management components being interconnected. The project management composite meta model (cf. Fig. 2.13) connects its import interface with its export and therefore enables interconnections between project management component instances.

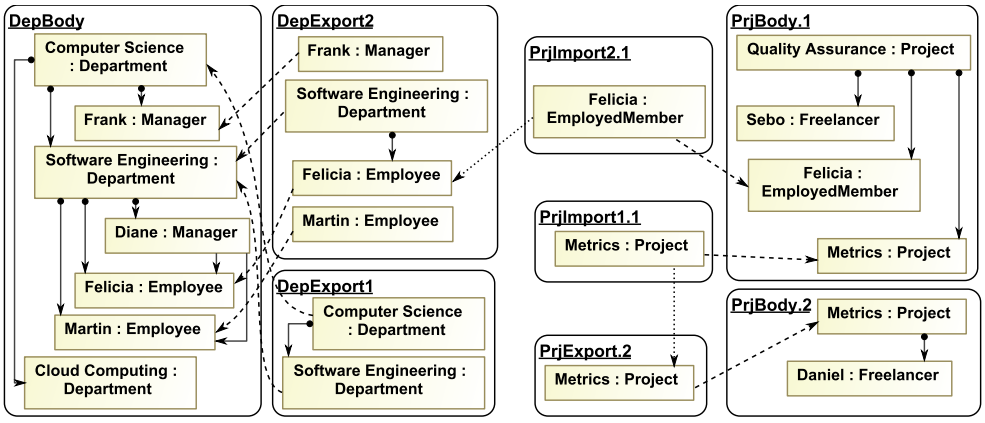


Figure 2.14: Sample instance of the composite meta model depicted in Fig. 2.13.

Assuming attributes in this context, it is conceivable that Felicia's vita stored in the department management component is not exposed to the project management component. Vice versa, within the project management compo-

nent the employed member Felicia may be equipped with additional project-related attributes like hours of work, project role, etc.

Part I

Formal Foundation

This first of two main Parts provides the formal foundation of the composite modeling concepts proposed in the present thesis.

All concepts rely on the theory of algebraic graph transformation and category theory, that is why in Chap. 3 a general introduction to graph transformation is given. Although attributes are common in models and are also well-founded in terms of attributed graphs, they are not explicitly considered throughout this thesis in order to concentrate on structural aspects. However, they can be added in a straightforward way. Chapter 3 also outlines a generalization of the algebraic approach to the categorical level by so-called High-Level Replacement systems (HLR systems) and HLR categories [24]. They enable generic definitions which can then be instantiated in the context of different categories. Of particular interest are weak adhesive categories [56] for which a number of theoretical results are available that may be exploited for related categories in this thesis.

While the concepts in this thesis focus on EMF models in particular, ordinary graphs are not sufficient to represent EMF specifics like *inheritance and containment*. Therefore, Chap. 4 joins these features to the formalism of graph transformation yielding transformation of graphs with inheritance and containment structures.

When local graphs are fully formalized, the formal description of the composition is introduced. It relies on distributed graph transformation introduced by Taentzer [77, 78] and extends the distribution structure notably by explicit import and export interfaces. In order to focus on the new composition concepts, the formalisms are given for ordinary graphs so far. Not until Chapter 6 the composition and graphs with inheritance and containment structures are combined leading to the composite graphs with inheritance and containment structure and their transformation.

These relations are also visualized in Fig. 1.1 in Sec. 1.5.

Chapter 3

Introduction to Algebraic Graph Transformation

This chapter builds a first bridge between the practical modeling world and the theory-oriented world of algebraic graph transformation [23] as applied in this thesis. It provides fundamental concepts of how models are formalized by graphs and how their changes are formalized by graph transformations. For now, graphs and graph transformations as defined in this chapter are only capable of representing basic model structures and their changes. In subsequent chapters, however, these definitions are extended and therefore constitute the basis for more subtle structures in order to suit sophisticated models such as EMF models and distributed/composite models.

In the following, the reader is given a very general approach on how models and their interrelations can be formalized by *graphs* and *graph morphisms*, respectively. Being a key concept in the modeling world, meta modeling is considered as well in the formalization. Distinguished graphs, called *type graphs*, serve as sort of meta models while their instances are represented by so-called *typed graphs* being graphs with appropriate morphisms to their type graphs. Model changes are formalized by *graph transformations* which are, on the one hand, based on the double-pushout approach [19] and, on the other hand, additionally formalized over the general framework of *High-Level Replacement Systems* (HLR Systems) [24]. The latter provides definitions and results on the level of category theory and enables heavy reuse by applying different categories of graph-like structures.

Here and throughout the rest of the paper, all aspects are presented in a formal way accompanied with informal descriptions and examples. Readers familiar with category theory and the theory of graph transformation may just skim through the formalisms, at least in this chapter.

3.1 Typed Graphs

Graphs are a natural means to represent the underlying structure of (visual) models. A graph primarily consists of a set of *nodes* and a set of *edges* which connect nodes by means of a source and target function. Different graphs may relate to each other by means of graph *morphisms* where nodes are mapped to nodes and edges are mapped to edges. Amongst others, morphisms are used to formalize the concept of meta modeling. Here, so-called *instance or typed graphs* map to a so-called *type graph* while their elements are called *instances or typed nodes/edges* and *node/edge types*, respectively. Type graphs correspond to meta models in the modeling world. Graphs may be extended by attributes as, e.g., presented by Ehrig et al. in [23]. However, this thesis concentrates on structural aspects and omits the formalization of attributes. Furthermore, UML-related properties like multiplicities are omitted as well.

Definition 3.1 ((Simple) Graph)**.** A simple *graph* $G = (G_N, G_E, s_G, t_G)$, shortly called graph, consists of a set G_N of nodes, a set G_E of edges, as well as source and target functions $s_G, t_G : G_E \to G_N$. ◇

Remark 3.2. If the distinction between nodes and edges is of no importance, the indices "N" and "E" are omitted, i.e., G instead of G_N. Operations on graphs are defined componentwise, e.g., the difference between two graphs, written $G \setminus H$, is defined by $G_N \setminus H_N$ and $G_E \setminus H_N$. Note that such operation may not necessarily lead to valid graph structures. Individual edges may also be addressed in the form $e\colon n \to m \in G_E$ where $s_G(e) = n \wedge t_G(e) = m$ and $n, m \in G_N$. If the context is clear, the assignment to G_E may be left out. ▽

Example 3.3. Figure 3.1 shows two graphs representing excerpts of a larger system model. On the left-hand side, a simple department management component is depicted which consists of a named `Department` being constituted by a number of `Employee`s and `Manager`s while managers may supervise employees. Managers and employees have a full name and a personnel number. Furthermore, departments may consist of several sub-departments. On the right, a project management component for companies is depicted. A named `Project` with a progress indication ratio may refer to related projects as well as spawned children projects. Projects have a number of members being employees of a company (`EmployedMember`) or being `Freelancer`s. Both provide a number of useful attributes, e.g., a full name, a project role, and a salary. These models are interpreted as graphs in the following way: All boxes are considered as nodes while arrows form the set of edges. Node identities are depicted in bold face. Edge identities are depicted next to arrows which run from source nodes to target nodes. Note that in Fig. 3.1 attributes are shown for comprehensibility only whereas in the examples and formalizations below they are omitted. △

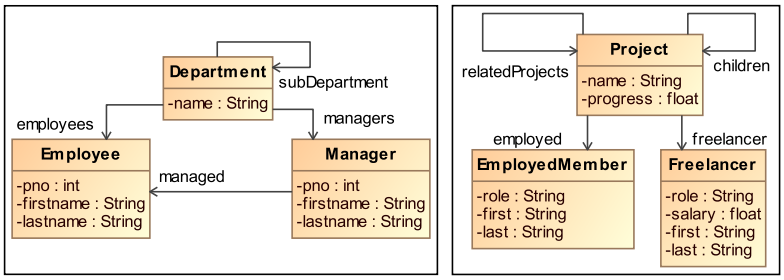


Figure 3.1: Graphs modeling a simple department management (left) and a simple project management (right).

Definition 3.4 (Graph morphism)**.** Given two graphs G and H, a pair of total functions (f_N, f_E) with $f_N : G_N \to H_N$ and $f_E : G_E \to H_E$ forms a graph morphism $f : G \to H$, shortly *morphism*, if it fulfills the following properties:

1. $f_N \circ s_G = s_H \circ f_E$ and (compatible source mappings)
2. $f_N \circ t_G = t_H \circ f_E$. (compatible target mappings)

If both functions f_N and f_E are injective, f is called *injective*. If both functions f_N and f_E are inclusions, i.e., $G_N \subseteq H_N$ and $G_E \subseteq H_E$, f is called *inclusion*. ◇

Graphs may be typed, i.e., a type is assigned to each graph element representing an "instance of" relation in terms of models. This type assignment is accomplished by a total graph morphism which maps typed nodes/edges to their node/edge types provided by a distinguished graph, called type graph.

Definition 3.5 (Typed graph, type graph and typing graph morphism)**.** A graph G is called *typed graph* or *instance graph*, if there exist a distinguished graph TG, called *type graph*, and a total graph morphism $type_G : G \to TG$, called *typing graph morphism*. ◇

Example 3.6. Figure 3.2 shows an example instance graph on the left-hand side and as type graph on the right the department graph of Fig. 3.1 without attributes. To enable proper differentiation, each instance node is equipped with a name and its type behind the colon, e.g., `Frank :Manager`. Corresponding to the given type, there exists a typing mapping to its node type. Edge typings are not explicitly shown by mappings in favor of readability. However, they can be uniquely deduced as long as there is at most one edge type between each two node types which is the case in the type graphs of the running example. On the left-hand side in Fig. 3.2 for example, the edges between `Diane :Manager` and `:Employee` nodes are all typed over the edge type `managed` running from the node type `Manager` to `Employee`. △

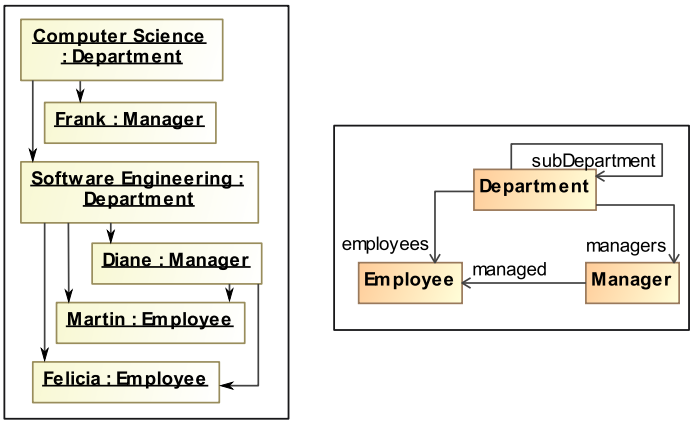


Figure 3.2: Sample instance graph (left) typed over a type graph (right) representing a department management component on instance and type level.

Definition 3.7 (Typed graph morphism)**.** Let G and H be two graphs typed by the typing graph morphisms $type_G \colon G \to TG$ and $type_H \colon H \to TH$, respectively. A *typed graph morphism* $f \colon G \to H$ is a graph morphism typed over a graph morphism $t \colon TG \to TH$, called *type graph morphism*, if the following holds: $type_H \circ f = t \circ type_G$ (cf. Fig. 3.3a). ◇

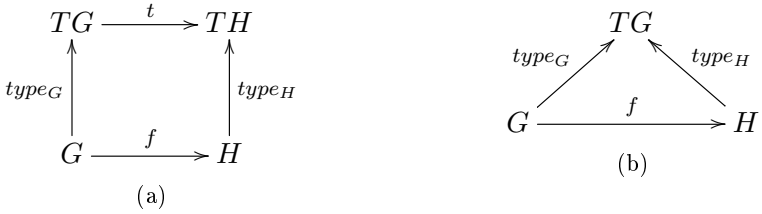


Figure 3.3: A typed graph morphism f (a) typed over a type graph morphism t and (b) typed over type graph TG.

Remark 3.8. Note that G and H being typed over the same type graph TG as illustrated in Fig. 3.3b can be considered as a special case of the definition above. Then, TG and TH are equal with t being the identical morphism. Accordingly, the property to hold reduces to $type_H \circ f = type_G$. In the following, such case is expressed by saying that morphism f is typed over TG. ▽

Now that (typed) graphs and graph morphisms are defined, some category theoretical conclusions follow. They are particularly significant in terms of graph transformations which rely on pushouts as the gluing construction on graphs (see the following section). A number of category theoretical results

relate to pushouts, e.g., in terms of their existence, composition and decomposition.

A category is defined over a class of objects and a class of morphisms, with an identity morphisms for each object and a composition operator on the morphisms. The class of all graphs and the class of all graph morphisms form the category GRAPHS. Analogously, typed graphs and typed graph morphisms form the category GRAPHS_{TG}.

It can be shown that both categories are co-complete, i.e., pushouts exist and consequently a common subgraph can always be constructed at which the gluing of two graphs may take place. For further details the reader may consult [23].

FACT 3.9 (Categories GRAPHS and GRAPHS_{TG})**.** Graphs and graph morphisms form a category, called GRAPHS (cf. [23, Sec. 2.2]). Furthermore, given a type graph TG, typed graphs over TG and typed graph morphisms form a category, called GRAPHS_{TG}. The category GRAPHS_{TG} can be considered as a slice category (see [23, Def. A.5]) of GRAPHS according to a graph TG.

REMARK 3.10. A similar category with injective typed graph morphisms only forms a category as well. The proof of category properties is straightforward relying on the fact that the composition of injective graph morphisms leads to injective graph morphisms again. ▽

FACT 3.11 (GRAPHS and GRAPHS_{TG} are co-complete)**.** In the categories GRAPHS and GRAPHS_{TG} pushouts exist (see [23, Sec. 2.3]). In each case, the initial object is the empty graph (see [23, p. 344]).

The kind of pushouts in GRAPHS_{TG} is illustrated in Fig. 3.4: Consider the commutative diagram with the graphs A, B, C, and D. Let A, B, and C be typed over TG by $type_A$ (not shown), $type_B$, and $type_C$, respectively. Furthermore, let a and c be typed over TG. Then a unique morphism $type_D$ exists such that D, b, and d are typed over TG by $type_D$, due to the universal property of pushouts.

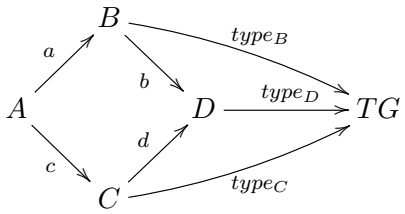


Figure 3.4: Illustration of the kind of pushouts in GRAPHS_{TG}.

In [23, Fact 4.16], the authors show that the categories GRAPHS and GRAPHS_{TG} of (typed) graphs and graph morphisms are *adhesive categories*

[56] along injective morphisms in that category. This allows to apply a number of results such as the Local Church-Rosser and Parallelism Theorems. Particularly, properties related to pushouts and pushout complements will facilitate the reasoning concerning graph transformations in subsequent sections.

Fact 3.12 (Graphs and Graphs_{TG} are adhesive categories.)**.** The categories Graphs and Graphs_{TG} are adhesive categories, where the monomorphisms in Graphs and Graphs_{TG} are the injective graph morphisms and injective typed graph morphisms, respectively.

3.2 Transformation of Typed Graphs

In this section, basic definitions concerning transformations in the categories Graphs and Graphs_{TG} of (typed) graphs and (typed) graph morphisms are provided. For the sake of simplicity, the formalisms below are mainly given without typing information while, however, corresponding typing could be added in a straightforward way analogously to the section above.

The key artifacts of graph transformations are *graph rules*. Rules consist of a left-hand side (LHS) L and a right-hand side (RHS) R, each one being a graph. Furthermore, a gluing graph K indicates corresponding items in L and R, i.e., identities, by morphisms running from K to L and from K to R. Roughly spoken, a rule is applied, called *transformation step*, by finding a morphism from L to a *host* graph G, and replacing the items in G identified by L with those in R. The replacement preserves all correspondences of L being mapped by K, deletes those not being mapped by K, and creates items corresponding to the part of R not being mapped by K.

The following definitions correspond to those in [23, Sec. 3] but are, however, slightly adapted to fit the needs of the current work.

Definition 3.13 (Graph rule)**.** A graph rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of the graphs L, K, R, called the *left-hand side* (LHS), the *gluing graph*, and the *right-hand side* (RHS), respectively, and the two injective graph morphisms $l\colon K \to L$ and $r\colon K \to R$. ◇

Remark 3.14. More general definitions of graph rules as in [23] do not require l and r to be injective. This enables rules to express gluing or splitting of graph elements. In the present approach, however, l and r are required to be injective to ease the reasoning. ▽

Definition 3.15 (Graph transformation step)**.** Given a rule p as defined above, a graph G, called *host* or *start* graph, and an injective graph morphism $m\colon L \to G$, a *transformation step* $G \overset{p,m}{\Longrightarrow} H$ from G to graph H is given, if (1) and (2) in Fig. 3.5 form a double pushout.

In order to conveniently refer to certain parts of the graphs being involved in a transformation step (cf. Fig. 3.5), the following is defined in addition:

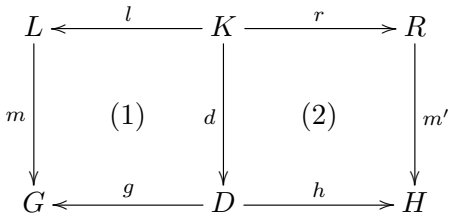


Figure 3.5: Transformation step by a double pushout (DPO).

- $L \setminus l(K) = L^{Del}$ represents the part to be deleted,
- $R \setminus r(K) = R^{Crt}$ represents the part to be created, and
- $G \setminus m(L) = G^{Ind}$ represents the part of G that is not mapped by m, i.e., that is independent of the transformation step.

◇

Remark 3.16. In Fig. 3.5, note that the morphisms g and h are injective since l and r are injective (cf. [23, Fact 2.17]). Furthermore, D is called *context* graph. Also note again that for graphs the set difference is calculated component-wise, i.e., for the sets of nodes and edges separately, which may not necessarily lead to a graph. ▽

A graph transformation step is not applicable in general. Consider for example a graph rule that only intends to delete a node as shown in Fig. 3.6, i.e., L contains a node while K and R are empty. If one tries to apply this rule to a host graph and particularly to a node with edges, the *context graph* D (compare Fig. 3.5) cannot be constructed in a valid fashion, since it then would contain dangling edges.

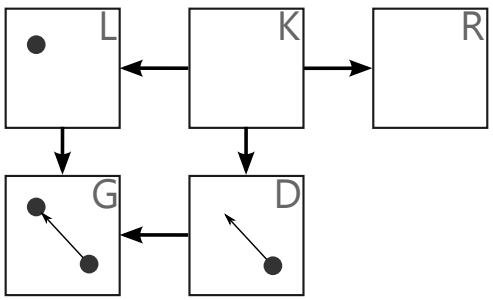


Figure 3.6: Invalid pushout due to a dangling edge.

Consequently, for a graph transformation step to be applicable, D must be constructed such that it leads to a pushout. A corresponding so-called *gluing condition* is given by the authors in [23] and is revisited and adapted below.

Definition 3.17 (Gluing condition for graphs)**.** Let there be a graph rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, a graph G, and an injective morphism $m\colon L \to G$. p and m satisfy the *gluing condition* if $\forall e \in G_E^{Ind} : s_G(e), t_G(e) \notin m(L_N^{Del})$ holds. Then, m is called *match*. ◇

Remark 3.18. The constraint given in the definition above is also known as *dangling condition* or *dangling points* property. In [23], it is defined slightly different but, however, the notation above is equivalent and the author believes that it is more intuitive. Additionally, note that the gluing condition given in [23] contains further properties, e.g., identification points which are only meaningful in a setting with non-injective morphisms. ▽

If the gluing condition is fulfilled then the context graphs *exists* and is *unique*. This is shown to hold below which also leads to the construction of the so-called *pushout complement*, i.e., the context graph D and its adjacent morphisms.

Fact 3.19 (Existence and uniqueness of context graph)**.** Given a graph transformation step $G \overset{p,m}{\Longrightarrow} H$ with a graph rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and a match $m\colon L \to G$, the context graph D exists and is unique up to isomorphism.

Proof. Consider a pushout diagram analog to Fig. 3.5.
"⇒". Given the pushout (1), the gluing condition as defined above follows from the properties of pushouts (cf. [23, Fact 2.17]) in the category Graphs: Consider $e \in G_E^{Ind}$ with $s_G(e) = m_N(n)$. Due to property 2, m and g are jointly surjective, there is an $e' \in D_E$ with $g_E(e') = e$ and $g_N(s_D(e')) = s_G(e)$. From property 3 it follows that there is $n' \in K_N\colon l_N(n') = n$ and $d_N(n') = s_D(e')$ and consequently $n \notin L_N^{Del}$ and $s_G(e) \notin m(L_N^{Del})$. Analogously for $t_G(e)$.
"⇐". If the gluing condition is satisfied, one can construct the *pushout complement* (d, g, D) as follows:

- $D = G \setminus m(L^{Del})$ is the componentwise construction of the *context graph* and
- then $d = m \circ l$ and $g = id_D$.

The uniqueness of pushout complements follows from the properties of adhesive HLR categories (cf. property 4 of [23, Fact 4.26]).

□

Now that conditions are given in order to construct the context graph in a valid fashion, a transformation step can be applied in two stages: At first, the context graph is constructed as given above, i.e., roughly spoken the images of L^{Del} are deleted in G which results in D. In stage two, nodes and edges in R^{Crt} are added to D yielding H.

Fact 3.20 (Existence and uniqueness of transformation step)**.** Given a graph transformation step $G \overset{p,m}{\Longrightarrow} H$ with a graph rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and a match $m\colon L \to G$, the transformation step exists and is unique up to isomorphism.

Proof. A transformation step can be constructed as follows:

1. Construct the context graph as shown in Fact 3.19.
2. Construct H as pushout of (K, r, d), i.e., as componentwise disjoint union $H = D \uplus R^{Crt}$.

The uniqueness of this construction follows from the uniqueness of D (cf. Fact 3.19) and the uniqueness of pushout objects (cf. [23, Fact 2.20]). □

Remark 3.21. The context graph and the transformation step are constructed such that $G \supseteq D \subseteq H$ with $g\colon D \hookrightarrow G$ and $h\colon D \hookrightarrow H$ being inclusions. Analogously, one could easily construct the rules to use inclusions since l and r are injective already. Given a graph rule p, the graph rule p' with inclusions can be constructed as follows: $K' = L \setminus L^{Del}$ and $R' = K' \uplus R^{Crt}$. Since l' and r' are the identical morphisms $id_{K'}$, it is sufficient to write $p' = (L \supseteq K' \subseteq R')$. Obviously, K' and R' are isomorphic to K and R, respectively. Note that rules with inclusions are not limited compared to rules with injective morphisms as defined so far. Both kinds can be easily constructed out of the other. ▽

Remark 3.22. The definitions of *typed graph rules* and *typed graph transformation steps* are straightforward, namely by typed rules and typed transformation steps in the category Graphs_{TG}. Furthermore, in the case of graph rules with inclusions $p = (L \supseteq K \subseteq R)$ there is $type_L \supseteq type_K \subseteq type_R$. ▽

3.3 Transformation in High-Level Replacement Systems

This section lifts the algebraic approach of graph transformation from Sec. 3.2 to the level of category theory. More abstract definitions like the one of rule and transformation step can then be instantiated in subsequent sections which simplifies definitions that recur again and again similarly. Moreover, general results can be applied to all instantiations.

High-level replacement systems, short HLR systems, can be considered as a general and formal framework of graph transformation systems where transformations are defined over objects of HLR categories. An HLR category is defined as a category C and a distinguished class of morphisms $\mathcal{M}$ which specifies those kind of morphisms to be used in transformation rules. In the following, the generic concepts of rules and transformation steps in [23, Chap.

5] are revisited and adapted to fit present needs, e.g., by limiting targeted categories and morphisms.

Rules consist of a left-hand side L, a right-hand side R, each one being an object in a given category C. Furthermore, a gluing object K indicates corresponding items in L and R, i.e., identities, by morphisms running from K to L and from K to R. Roughly spoken, a rule is applied, called transformation step, by finding a morphism from L to an object G, called host object, and replacing the items identified by L with those in R. The replacement deletes all items that occur in L without being mapped by K and creates all items occurring in R without being mapped by K. Those items in K have counterparts in L and R and are therefore preserved.

As pointed out already, typing plays an important role and is implemented by typing morphisms. Needless to say, that this approach is continued below by also providing definitions of transformations which take typing into account.

Definition 3.23 (Rule)**.** Let C be a category and let $\mathcal{M}$ be a distinguished class of injective morphisms in C. A rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ in C consists of the objects L, K, R, called the left-hand side, the gluing object, and the right-hand side, respectively, and the two morphisms $l\colon K \to L$ and $r\colon K \to R$ with $l, r \in \mathcal{M}$. ◇

Definition 3.24 (Transformation step)**.** Given a rule p as defined above, an object G in C, called *host* or *start* object, and a morphism $m\colon L \to G$ in C, a transformation step $G \overset{p,m}{\Longrightarrow} H$ from object G to object H in C is given, if (1) and (2) in Fig. 3.7 form a double pushout in the category C. ◇

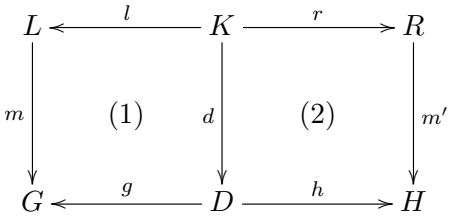


Figure 3.7: Transformation step by a double pushout (DPO).

Remark 3.25. In Fig. 3.7, note that the morphisms g and h are injective since l and r are injective (cf. [23, Fact 2.17]). Furthermore, D is called *context* object. ▽

Definition 3.26 (Applicability of transformation steps)**.** Given a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, an object G and a morphism $m\colon L \to G$ in C, p is called *applicable* to G via m if the following conditions are satisfied:

1. There are an object D and two morphisms $g\colon D \to G$ and $d\colon K \to D$ as illustrated in Fig. 3.7 with g belonging to $\mathcal{M}$, such that (1) becomes a pushout. Then (D, d, g) is called *pushout complement.*

2. There is a pushout (H, m', h) over $d\colon K \to D$ and $r\colon K \to R$ in (2) with h belonging to $\mathcal{M}$.

◇

Now the notion of typing is added to rules and transformation steps by the category C_{TG} with TG being an object in C.

Definition 3.27 (Typed rule)**.** Let C_{TG} be the slice category of a category C and a distinguished object TG in C. A rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ in C_{TG} is given by the tuple $p_{TG} = (p, type)$, where L, K, and R are typed over TG by the triple morphism $type = (type_L\colon L \to TG, type_K\colon K \to TG, type_R\colon R \to TG)$ (cf. Fig. 3.8), such that $type_L \circ l = type_R \circ r = type_K$ and with $type_L, type_R, type_K$ being total. ◇

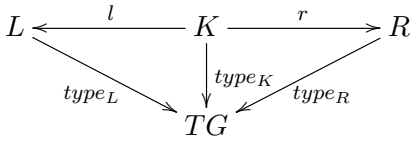


Figure 3.8: Typed rule.

Definition 3.28 (Typed transformation step)**.** A typed transformation step in C_{TG} is given by the transformation step $G \overset{p_{TG},m}{\Longrightarrow} H$ with typed rule p_{TG} and with the objects G, D and H being typed over TG by appropriate morphisms, if (1) and (2) are double pushouts in the category C_{TG} as illustrated in Fig. 3.9. ◇

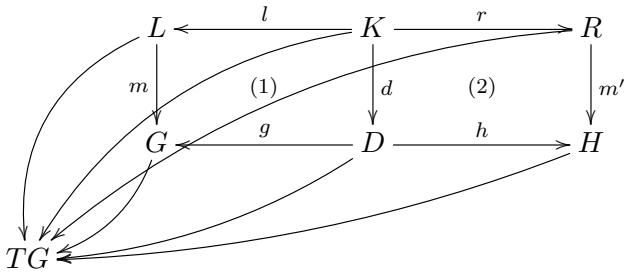


Figure 3.9: Typed transformation step.

Remark 3.29. In order to obtain general results for graph transformation based on weak adhesive HLR categories, additional properties for the class

of injective morphisms have to be satisfied (see [23]) such as compatibility of coproducts with $\mathcal{M}$, existence of initial pushouts over $\mathcal{M}'$-morphisms, and $\mathcal{E}' - \mathcal{M}'-$ pair factorization. These properties hold for finitary categories [14] which are based on $\mathcal{M}$-adhesive categories, a generalization of weak adhesive HLR categories. In terms of the category of typed graphs with $\mathcal{M}$ being the class of injective typed graph morphisms this means, roughly spoken, typed graphs in $(\text{GRAPHS}_{TG}, \mathcal{M})$ are finite if the node and edge sets have finite cardinality while the type graph TG itself may be infinite. The following basic results for graph transformation are valid for the case of finitary graphs: Local Church Rosser Theorem ([23, Thm. 5.12]), Parallelism Theorem ([23, Thm. 5.18]), Concurrency Theorem ([23, Thm. 5.23]), Embedding and Extension Theorem ([23, Thm. 6.14 and Thm. 6.16]), and Local Confluence Theorem ([23, Thm. 6.28 and Lemma 6.22]). $\triangledown$

Chapter 4

Transformation of Graphs with Inheritance and Containment

Graphs as defined so far suit well to describe ordinary model structures. In order to especially represent object-oriented model structures, in particular MOF-based models like EMF ([27]), plain typed graphs are not sufficient as they lack prominent aspects of such models, e.g., *inheritance* with *abstract* types and *containment* structures. Therefore in this chapter, (typed) graphs with inheritance and containment structures, short *IC-graphs*, are introduced. They essentially build upon plain (typed) graphs extended by additional information and constraints. Obviously, these additives demand a reconsideration of related transformations which actually exposes difficulties due to the new structures. This necessitates the definition of *consistent* transformations to circumvent occurring problems.

This chapter brings the work in [11] (extended in [12]) and [38] together which provide a general formalization of EMF model transformation and categorical reasoning in terms of weak adhesive categories, respectively. It is structured as follow: At first, (typed) graph with inheritance and containment structures are introduced. Afterwards, their transformation is elaborated.

4.1 Typed Graphs with Inheritance and Containment Structures

In this section, the concept of typed graphs with inheritance and containment structures is presented. While inheritance describes a parent-child relationship, containment defines an ownership relation between instances of two node types. The notion of abstract types is implemented by a distinguished set A which shares its nodes with the corresponding graph and which declares nodes to be of abstract types. Analogously to common object-oriented languages, instances of abstract node types are forbidden.

The inheritance concept is represented by a relation I which captures inheritance information. The relation *clan* identifies all clan members of a given parent node, i.e., children, grandchildren, etc., including the node itself. This is analogously to [38] and differs from the definition in [11] which represents inheritance in form of an inheritance graph. However, the formalization in [38] appears to be more concise. The semantics of this inheritance concept is analogous to that of object-oriented programming languages, i.e., features of parents are passed on to their children. Since attributes are omitted here, edges are the sole features to inherit. Nevertheless, attributed graphs with node type inheritance are formalized in [20]. An important constraint with regard to inheritance is that inheritance cycles are forbidden. This is quite intuitive since children cannot be their own parents. However, this differs from the definition in [38] where cyclic inheritance is allowed at first and restricted in a later step.

Containment edges are identified by a subset C of edges of the related graph. If a containment edge runs from node n to node n', it is said that n *contains n'*. A corresponding relation *contains* is defined in order to identify all node pairs being connected by such edges. This relation is transitively defined and respects inherited containments. In EMF, it is a desired property for models that at least all non-abstract nodes are transitively contained in one root node, mainly for convenient persistence of EMF models. This leads to instance models with explicit tree structures. However, such a property is not required in the present approach.

Note at this point that the definition of graphs with inheritance and containment below forms the basis for corresponding graphs on type level and instance level. This holds true for the related subsequent definition of graph morphisms with inheritance and containment as well aiming at the centralization and reuse of properties, on the one hand, and the certainty to deal with generally equal morphisms all over, on the other hand.

Definition 4.1 (Graph with inheritance and containment)**.** A tuple $G = (T, I, A, C)$, called *graph with inheritance and containment* or short *IC-graph*, consists of a simple graph $T = (G_N, G_E, s_G, t_G)$, an *inheritance* relation $I \subseteq G_N \times G_N$ with I^* being the reflexive and transitive closure of I, a set $A \subseteq G_N$ of abstract nodes and a set $C \subseteq G_E$ of containment edges. In addition, it is required:

- $\forall n \in G_N$, the inheritance *clan* is defined by $clan_G(n) = \{n' \in G_N \mid (n', n) \in I^*\}$ with
  $\forall n, m \in G_N : n \in clan_G(m) \wedge m \in clan_G(n) \Rightarrow n = m$ (no inheritance cycles).

Furthermore, the following is defined:

- $clan_G(M) = \bigcup_{n \in M} clan_G(n)$.

- a containment relation $contains_G$ corresponding to C:
  Let $contains'_G = \{(n,m) \in G_N \times G_N \mid \exists c \in C \wedge x,y \in G_N : s_G(c) = x \text{ with } n \in clan_G(x) \wedge t_G(c) = y \text{ with } m \in clan_G(y)\}$.
  $contains_G$ is the transitive closure of $contains'_G$.

- An IC-graph G is called *rooted* if there is one non-abstract node $r \in G_N \setminus A$, called *root node*, which contains all other non-abstract nodes transitively: $\forall n \in G_N \setminus (A \cup \{r\})\colon (r,n) \in contains_G$.

- For each simple graph K, its equivalent IC-graph K' is defined by a *straight inheritance extension with containment* as follows: $K' = (K, \emptyset, \emptyset, \emptyset)$.

- G is called *I-graph* if C is empty. For each IC-graph $G = (T, I, A, C)$, its I-graph is defined by $G' = (T, I, A, \emptyset)$.

Tuple elements T, I, A and C of an IC-graph G can also be referred to by $T(G)$, $I(G)$, $A(G)$, and $C(G)$, respectively. Since T specifies the main structure of an IC-graph G, from now on G_N, G_E, s_G, and t_G abbreviate $T(G)_N$, $T(G)_E$, $s_{T(G)}$, and $t_{T(G)}$. ◇

Example 4.2. Figure 4.1 shows two IC-graphs which can be considered to be the refactored[1] versions of the simple graphs in Fig. 3.1. Inheritance is visualized by an arrow with a triangular head at the parent's end, while a containment relation is indicated by a diamond at the container's end. Both are UML-compliant visualizations. In the present formalization, the source node of an edge being in the set C is considered to be the container of the target node. In relation I, each tuple (n,m) is interpreted as parent m with child n. Attributes are shown again just to motivate the setting, although they are not part of the formalization given in this paper.

On the left of Fig. 4.1, for example, the abstract node `Person` has been introduced to encapsulate the attributes `firstname` and `lastname`. This kind of refactoring helps to prevent redundancy. Since `Employee` and `Manager` are (transitive) children of `Person`, i.e., $clan(\texttt{Person}) = \{\texttt{Person}, \texttt{Employee}, \texttt{Manager}\}$, both inherit these two attributes. As child of `Employee`, `Manager` is contained indirectly in `Department` since the incoming containment relation `employees` is inherited as well. That means, for the left IC-graph there is $\{(\texttt{Department}, \texttt{Employee}), (\texttt{Department}, \texttt{Manager})\} \subset contains$. In the project management component on the right-hand side, the abstract node `AMember` encapsulates the attributes `role`, `first`, and `last`. Besides the additional introduction of containments, the rest of the graph corresponds to that in Fig. 3.1.

Considering `Department` and `Project` as root elements, both graphs are rooted as all remaining (non-abstract) nodes are transitively contained. △

[1] Refactoring is the process of structurally improving a model towards certain model quality aspects, e.g., modularity and reusability, while preserving its semantics.

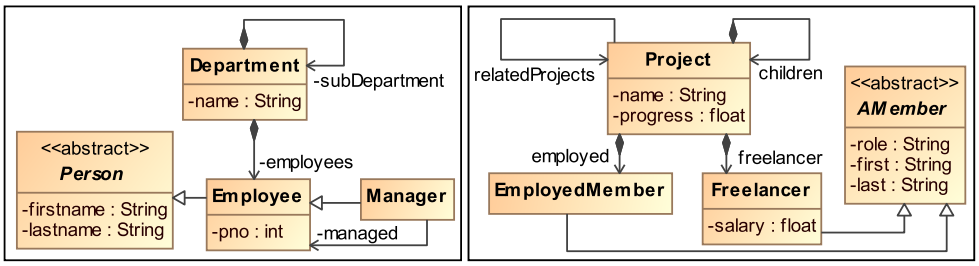


Figure 4.1: IC-graphs modeling a department management (left) and a project management (right).

Analogously to the previous section, interrelations between IC-graphs can be established by so-called IC-morphisms as given in Def. 4.3 below. They are defined over plain graph morphisms while taking inheritance structures into account, too. In order to preserve structural equivalence, they have to satisfy two essential constraints: Constraint (1) ensures that clans are mapped to correspondent clans. From this and the definition of graphs, it also follows that the source (target) node of an origin edge should be mapped either to the source (target) of the image edge directly or, alternatively, to one of its subtypes (see Lemma 4.5 below). The preservation of containment properties is guaranteed by constraint (2), i.e., containment edges have to be mapped to containment edges again.

Definition 4.3 (Graph morphism with inheritance and containment)**.** Given two IC-graphs G and H, a morphism $f\colon G \to H$ is a graph morphism $f_T\colon T(G) \to T(H)$. f is also called *graph morphism with inheritance and containment* or short *IC-morphism*. If the context G is clear, f_{G_E} and f_{G_N} are abbreviated by f_E and f_N. Additionally, the following constraints are required to hold:

1. $\forall n \in G_N : f_N\left(clan_G(n)\right) \subseteq clan_H\left(f_N(n)\right)$ (clan-compatible node mapping), and

2. $f_E(C(G)) \subseteq C(H)$ (containment-compatible edge mapping).

◇

Remark 4.4. An IC-morphism $f\colon G \to H$ does not define explicit mappings between $A(G)$ and $A(H)$ and between $C(G)$ and $C(H)$. However, these mappings can be directly deduced by f_N and f_E, respectively. This is obviously not the case for mappings between $I(G)$ and $I(H)$ since the constraint 1 in Def. 4.3 merely requires mapped original parent and child nodes to be in a transitive parent-child relationship in the image. Accordingly, there is rather

an implicit mapping between the transitive closures $I^*(G)$ and $I^*(H)$ inferred by f_N. ▽

In the definition of IC-morphisms above only nodes are required to be mapped in a clan-compatible way. The question arises whether this should not be the case for edges as well. And indeed, a clan-compatible edge mapping is already induced due to the clan-compatible node mapping, on the one hand, and the conventional source/target compatibility constraints of graph morphisms, on the other hand. This is formally shown in Lemma 4.5 below.

Lemma 4.5. Source and target nodes of a mapped edge are mapped in a clan-compatible way, i.e.,
$\forall e \in G_E : f_N(s_G(e)) \in clan_H\,(s_H(f_E(e)))$ and $f_N(t_G(e)) \in clan_H\,(t_H(f_E(e)))$.

Proof. Since $\forall n \in G_N : n \in clan_G(n)$, there is
$\forall e \in G_E : f_N(s_G(e)) \in f_N(clan_G(s_G(e))) \overset{Def.\ 4.3.1.}{\subseteq} clan_H(f_N(s_G(e))) \overset{Def.\ 3.4}{=} clan_H(s_H(f_E(e)))$. Analogously for t. □

IC-graphs and IC-morphisms form a category. This appears intuitive but deserves a closer look especially due to the rather subtle clan-compatible mappings.

Proposition 4.6 (Category ICGraphs)**.** IC-graphs and IC-morphisms form a category, called ICGraphs.

Proof. The axioms of categories are shown, i.e., the compositionality and associativity of morphisms and the existence of identity morphisms.

Given two IC-morphisms $f\colon G \to H$ and $g\colon H \to K$, it is shown that their composition $g \circ f\colon G \to K$ is an IC-morphism again. That is, the compatibility of node mappings are to be checked. Afterwards, associativity is shown and that containment edges are mapped to containment edges:

1. To show: $\forall n \in G_N : g_N(f_N(clan_G(n))) \subseteq clan_K(g_N(f_N(n)))$
   $g_N(f_N(clan_G(n))) \subseteq g_N(clan_H(f_N(n)))$, since f is an IC-morphism
   $g_N(clan_H(f_N(n))) \subseteq clan_K(g_N(f_N(n)))$, since g is an IC-morphism.

2. The composition of IC-morphisms is associative since it is defined over the composition of graph morphisms that is shown to be associative (see Fact 3.9).

3. To show: $g_E(f_E(C(G))) \subseteq C(K)$
   $g_E(f_E(C(G))) \subseteq g_E(C(H))$, since f is an IC-morphism
   $g_E(C(H)) \subseteq C(K)$, since g is an IC-morphism.

For each IC-graph G, the identity morphism $id_G\colon G \to G$ is chosen which is obviously well-formed according to Def. 4.3. Furthermore, given an IC-morphism $f\colon G \to H$, $f \circ id_G = f$ and $id_H \circ f = f$ are straightforward to show. □

In the object-oriented modeling world, instance models do not carry inheritance structures or containment themselves in contrast to meta models. This is reflected in the formalisms. Instances of *type IC-graphs* can essentially be formalized by simple graphs which have to obey to the structure of their type IC-graphs. For the sake of convenient handling, in the following simple graphs are encoded as IC-graphs with empty inheritance relations and with the set of containment edges inferred by the actual edge typing. Such IC-graphs are then called *typed IC-graphs*. The inferred set of containment edges may appear redundant with the containment edge definitions in the type graph. However, it strongly increases the readability of containment-related constraints in subsequent formalisms. For example, two related constraints can already be found in the definition of typed IC-graphs below which state that instance nodes must have at most one container and that cyclic containment is forbidden. A corresponding definition is given in [11] and is revised in Def. 4.7 below.

Definition 4.7 (Typed IC-graph, type IC-graph, typing IC-morphism and typed IC-morphism)**.** Let there be a distinguished IC-graph TG, called *type IC-graph*, a simple graph $G = (G_N, G_E, s_G, t_G)$ and its straight inheritance extension with containment G'. Furthermore, let $type_{G'} \colon G' \to TG$ be an IC-morphism which also redefines $C(G')$ by $C(G') = \{e \in G'_E | type_{G'_E}(e) \in C(TG)\}$. G' is called *typed IC-graph* and $type_{G'}$ is called *typing IC-morphism* if the following holds:

1. $\forall e1, e2 \in C(G') : t_G(e1) = t_G(e2) \Rightarrow e1 = e2$ (at most one container), and
2. $\forall n \in G'_N : (n, n) \notin contains_{G'}$ (no containment cycles).

For simplicity, G and G' are used synonymously in the following, e.g., $C(G)$ is used in place of $C(G')$ and $type_G \colon G \to TG$ is written for the underlying IC-morphism $type_{G'} \colon G' \to TG$. Furthermore, if the context G is clear, $type_{G_N}$ and $type_{G_E}$ are abbreviated to $type_N$ and $type_E$, respectively.

A typing IC-morphism $type_G$ is called *concrete* if all nodes in G_N are typed over concrete types, i.e., $\forall n \in G_N \colon type_{G_N}(n) \notin A(TG)$. Otherwise it is called *abstract*.

Typed IC-morphisms are defined analogously to Def. 3.7 replacing graphs with IC-graphs and graph morphisms with IC-graph morphisms. ◇

Example 4.8. According to the type IC-graph of a department management component depicted in the left of Fig. 4.1, the instance graph already shown in the left of Fig. 3.2 is still valid. In this special case the type graphs have been changed slightly by a refactoring (see Ex. 4.2) without affecting the structure of their instance graphs. △

Example 4.9. Figure 4.2 shows two type IC-graphs in the top row with corresponding instance IC-graph samples in the bottom row. In detail, two different type graphs for a department management component are shown being mapped by a type IC-graph morphism denoted by dotted arrows. `Department` is straightforwardly mapped to `Department`. Furthermore, the abstract type node `Person` and the concrete type node `Employee` on the left-hand side are both mapped to `Employee` on the right-hand side which is a valid clan-compatible node mapping. The containment edge `persons` between `Department` and `Person` is mapped to the containment edge `employees` between `Department` and `Employee`. This is a valid IC-mapping altogether since the sources of both mapped edges are directly mapped and the origin target `Person` is mapped to the image target `Employee`. Since `Employee` on the left inherits the edge `persons` as well, related edge mappings have to be compatible with that inheritance. This is the case as `Employee` on the left is clan-compatibly mapped to the image of the target of `persons`, namely `Employee`.

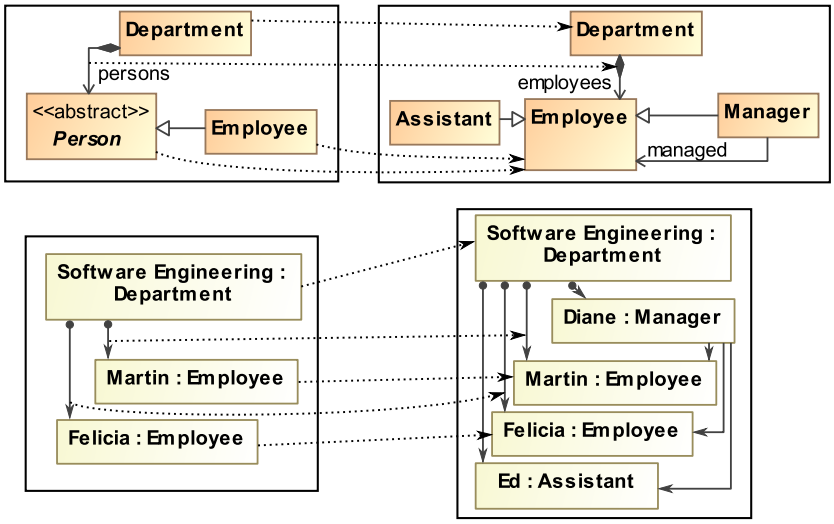


Figure 4.2: Type IC-graphs connected by a type IC-graph morphism (top) and corresponding typed IC-graphs connected by a typed IC-graph morphism (bottom).

For each type graph, an instance graph is shown in the lower part of Fig. 4.2. Again, to enable proper differentiation, each instance node is equipped with a name and its type. Moreover, small circles at edges denote edges typed over containment edge types.[2] Please note that the types of instance edges and corresponding edge typings are not explicitly shown but can be uniquely

[2]On instance level there is no UML-compliant and distinguishable visualization of instance edges being typed over compositions/aggregations or ordinary associations.

determined from their contexts. In the instance graph on the left, e.g., the edge between `Software Engineering :Department` and `Martin :Employee` is of type `persons` which runs between the types `Department` and `Person` and consequently between `Department` and `Employee` due to inheritance. Similarly on the lower right, non-containment edges are of type `managed`. This typing is compatible with the node typing since the target node types `Employee` and `Assistant` are in the clan of `Employee`. To summarize, the mappings between the instance graphs are compatible with the mappings between the type graphs.

Note that Fig. 4.2 shows instance graphs with nodes typed over concrete node types only. Consequently, the typing morphisms can be considered to be *concrete*. △

Now, the category of typed IC-graphs and typed IC-morphisms can be defined, called ICGRAPHS$_{TG}$. Moreover, please note Remark 4.11 below which also defines the categories IGRAPHS and IGRAPHS$_{TG}$ being sub-categories of ICGRAPHS and ICGRAPHS$_{TG}$, respectively. They essentially indicate an empty containment set and facilitate the handling of pushouts in the following.

PROPOSITION 4.10 (Category ICGRAPHS$_{TG}$)**.** Let TG be an IC-graph. IC-graphs and IC-graph morphisms typed over TG form a category, called ICGRAPHS$_{TG}$.

PROOF. The category of typed IC-graphs over TG and typed IC-morphisms is a slice category (cf. [23, Def. A.5]) of ICGRAPHS. □

REMARK 4.11 (Sub-categories)**.** I-graphs and IC-graph morphisms form a full sub-category of ICGRAPHS, called IGRAPHS. Let TG be an IC-graph and TG' its I-graph, I-graphs and IC-graph morphisms typed over TG' form a full sub-category of ICGRAPHS$_{TG}$, called IGRAPHS$_{TG}$. Similarly, category GRAPHS$_{TG}$ is the full sub-category of IGRAPHS$_{TG}$ where all graphs are typed over graph $T(TG)$, the underlying graph of TG. ▽

After having defined the category ICGRAPHS$_{TG}$ of typed IC-graphs and typed IC-morphisms, pushouts in that category are now considered. It is rather easy to see that ICGRAPHS$_{TG}$ does not have pushouts in general due to the additional constraints for containment structures. An example for an invalid existing pushout is given in Example 4.13 below. However, the existence of pushouts in the category ICGRAPHS$_{TG}$ is required for appropriate graph transformations based on the double-pushout (DPO) approach. For now, the existence of pushouts for IC-graphs with *empty containment* is shown only, i.e., for I-graphs, and invalid pushouts according to containments are tackled later.

In [38], Hermann et al. show that the category of graphs with inheritance and corresponding morphisms forms a weak adhesive HLR category (see

the definition of weak adhesive HLR categories in Def. 10.3 in the appendix) for a class $\mathcal{M}_{S-refl}$ of morphisms that reflect subtypes of original nodes in their images. It follows that such a category has pushouts over at least one morphism in $\mathcal{M}_{S-refl}$. This result is exploited here in terms of the category IGRAPHS$_{TG}$. Below, it is shown that the category IGRAPHS$_{TG}$ with the class $\mathcal{M}_{inj}$ of injective morphisms is also a weak adhesive HLR category.

PROPOSITION 4.12 ((IGRAPHS$_{TG}, \mathcal{M}_{inj}$) is a weak adhesive HLR category)**.** The category IGRAPHS$_{TG}$ with the class $\mathcal{M}_{inj}$ of injective typed I-graph morphisms forms a weak adhesive HLR category.

PROOF. Let $\mathcal{M}_{S-refl}$ be the class of S-reflecting morphisms as defined in 10.1. (IGRAPHS, $\mathcal{M}_{S-refl}$) is a weak adhesive HLR category (see [38, Thm. 1]). Since IGRAPHS$_{TG}$ is a slice category over IGRAPHS, (IGRAPHS$_{TG}, \mathcal{M}_{S-refl}$) is also a weak adhesive HLR category (see [23, Thm.4.15]).

Objects in IGRAPHS$_{TG}$ are graphs which have empty inheritance and containment structures (but which are typed over a graph TG with a possibly non-empty inheritance structure). Consequently, the constraints of class $\mathcal{M}_{S-refl}$ with respect to subtype reflection are trivially satisfied and therefore the class of all injective typed I-morphisms can be chosen as $\mathcal{M}_{inj}$. Thus, (IGRAPHS$_{TG}, \mathcal{M}_{inj}$) is also a weak adhesive HLR category.

□

EXAMPLE 4.13. In Fig. 4.3, instances of the department management component are considered again (cf. meta model in Fig. 4.1). Small circles at edges denote those edges typed over containment edge types and are located at the container's side equivalent to the diamonds on type level. The IC-graph morphisms are interpreted as normal typed graph morphisms and the pushout in the category GRAPHS$_{TG}$ is constructed. See the pushout graph in the bottom right corner of Fig. 4.3. This graph is an invalid IC-graph violating the constraint of forbidden cyclic containments. Thus, this example shows that the pushout construction in the category ICGRAPHS$_{TG}$ cannot be based on that of the category GRAPHS$_{TG}$

Moreover, there is no other IC-graph being the pushout graph due to the following argumentation: To avoid cyclic containment one can try a sub-graph of the pushout graph in GRAPHS$_{TG}$ by deleting at least one of the containment edges. However, pushout morphisms cannot be found then.

This shows that the categories ICGRAPHS and ICGRAPHS$_{TG}$ do not form weak adhesive HLR categories. △

The report in [39] (strongly related to [38]) shows furthermore that the category (IGRAPHS$_{TG}, \mathcal{M}_{S-refl}$) is finite if nodes and edges in graphs have finite cardinality. In correspondence with Remark 3.29, fundamental results for transformations such as the Local Church Rosser Theorem, the Parallelism Theorem, and the Concurrency Theorem are valid for (IGRAPHS$_{TG}, \mathcal{M}_{S-refl}$) and consequently for (IGRAPHS$_{TG}, \mathcal{M}_{inj}$) with finite graphs.

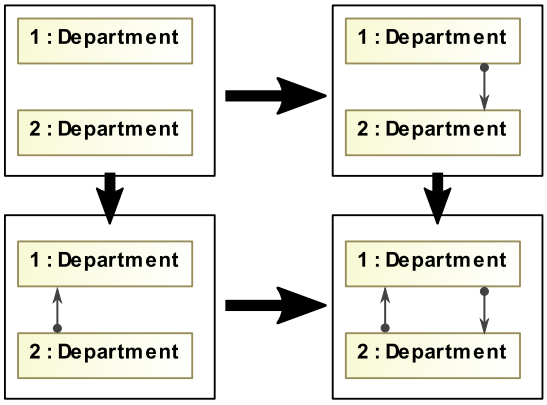


Figure 4.3: Counter example for co-completeness of the category ICGRAPHS$_{TG}$.

4.2 Consistent Transformation of Typed IC-Graphs

With typed IC-graphs and IC-morphisms at hand, this section introduces their transformation, called *IC-transformation*, which shall be obviously more sophisticated than the transformation of plain typed graphs. First of all, transformation rules and steps are defined as instances of the definitions 3.27 and 3.28 in the category ICGRAPHS$_{TG}$ with $\mathcal{M}$ being the class of typed IC-morphisms. As indicated previously, pushouts over (typed) IC-graphs and IC-morphisms may not exist in every case since prohibited containment cycles may occur. This issue is addressed by the definition of so-called *consistent IC-rules* which provide a consistency preserving structure and thus ensure the existence of pushouts. First elaborations on dedicated containment conditions ensuring valid pushouts have been made by Köhler et al. in [53]. Their results are then incorporate in [11] originally defining the notion of consistent rules which are recalled and slightly adapted here to fit the needs of the present approach.

DEFINITION 4.14 (Typed IC-graph rule)**.** An IC-graph rule is a typed rule $p_{TG} = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, type)$ in the category ICGRAPHS$_{TG}$ (see Prop. 4.10). Furthermore, the following must hold: $type_{R_N}(R_N^{Crt}) \cap A(TG) = \emptyset$. ◇

DEFINITION 4.15 (Typed IC-graph transformation step)**.** A direct typed IC-graph transformation $G \overset{p_{TG},m}{\Longrightarrow} H$ is a typed transformation step in the category ICGRAPHS$_{TG}$. ◇

REMARK 4.16. It is shown in [20] that the resulting graph H is well typed over TG. ▽

Remark 4.17. Analogously to Remark 3.21, typed IC-graph rules may be constructed by inclusion morphisms as well. Below, rules with inclusions are considered only to simplify constructions and proofs. ▽

In the following, the notion of consistent rules are recalled from [11] which provide sufficient criteria to ensure that the containment hierarchy remains consistent during their applications. Note that in [11], conditions according to both the containment and the rooted property are listed. In the present approach, the rooted property plays a minor role thus only those conditions are of importance here which prevent the creation of containment cycles.

Consistent transformation rules allow the following kinds of actions related to containment changes:

1. Create a containment relation together with the contained object node or change the container.

2. Create cycle-capable containment edges only if the old and the new container are both transitively contained in the same container. A cycle-capable containment is an edge whose source and target nodes may be typed over the same node type.

The purpose of condition 1 is pretty intuitive: If a containment edge is created together with its target, containment cycles cannot occur. In cases where the target node is preserved, the created containment edge is considered as a replacement, i.e., the old containment edge running to the target node needs to be deleted while the new containment edge is created at the same time. Note that this is only a sufficient condition. The host graph may carry nodes which are not yet the target of any containment edge and thus their preservation together with the creation of related containments would lead to valid IC-graphs. Condition 2 covers a special case of the containment replacement where source and target nodes of containments may be of the same type. Then, the replacement is only allowed along the current containment hierarchy of the source node. In Example 4.19 below, these conditions are illustrated by IC-rules.

Definition 4.18 (Consistent IC-graph rule)**.** Let $p_{TG} = (L \supseteq K \subseteq R, type)$ be a typed IC-graph rule and let $R_C^{Crt} = C(R^{Crt})$ and $L_C^{Del} = C(L^{Del})$. The IC-graph rule p_{TG} is *consistent* if the following constraints hold:

1. $\forall e \in R_C^{Crt}$ with $t_R(e) = n$: $n \in R_N^{Crt} \vee$ (containment edge creation)
   $(n \in K_N \wedge \exists e' \in L_C^{Del}$ with $t_L(e') = n)$

2. $\forall e \in R_C^{Crt}$ with $s(e) = n \wedge t(e) = m$: (cycle-capable containment edge creation)
   $\exists e' \in L_C^{Del}$ with $s(e') = o \wedge t(e') = m$:
   $((o, n) \in contains_L \wedge (m, n) \notin contains_L) \vee (n, o) \in contains_L$

◇

Example 4.19 (Cycle-capable containment)**.** Figures 4.4 and 4.5 show IC-rules typed over the *department management* type graph shown at the left of Fig 4.1. In the type graph, particularly note the `Department` type and the `subDepartment` containment edge type pointing to `Department` again. The IC-rules are shown by their left-hand side (LHS) and right-hand side (RHS). Note that the gluing graph is omitted and node identities are denoted by equal numbers. Edge identities are not shown at all in favor of readability but they can be uniquely deduced.

The rule given in Fig. 4.4 shows a linear containment hierarchy at which the lowermost department shall be contained by the topmost department instead. This is a containment creation/replacement as defined above, i.e., condition 1 holds. However, since `subDepartment`, the edge type of all edges here, is a cycle-capable containment edge, the condition 2 must hold as well. In the RHS of Fig. 4.4, the new containment edge's source still belongs to the same containment hierarchy as before and therefore satisfies condition 2. Since both conditions are satisfied the present rule is a consistent IC-rule.

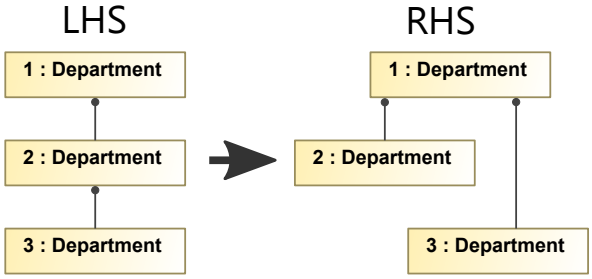


Figure 4.4: Consistent IC-rule which properly deletes and creates a cycle-capable containment edge.

Figure 4.5 shows another IC-rule which obviously violates condition 2. Its illegal behavior is illustrated by means of a corresponding host graph and the resulting graph, depicted at the lower left and right. This time, the replacement of the containment edge does not occur along the same containment hierarchy and in this case leads to a cyclic containment. △

Fact 4.20. Let $p_{TG} = (L \xleftarrow{l} K \xrightarrow{r} R, type)$ be a consistent rule, $m\colon L \to G$ be a match to an IC-graph G which is typed by $type_G\colon G \to TG$. Then, an IC-transformation step $G \overset{p_{TG},m}{\Longrightarrow} H$ yields an IC-graph H typed over TG.

Proof. See [11, A.1]. □

Please note that this proof only uses the conditions presented in Definition 4.18. However, additional consistency conditions presented in [11] are needed to show that if G is *rooted*, H is rooted as well. For example, this also requires that a new object node is immediately connected to its container.

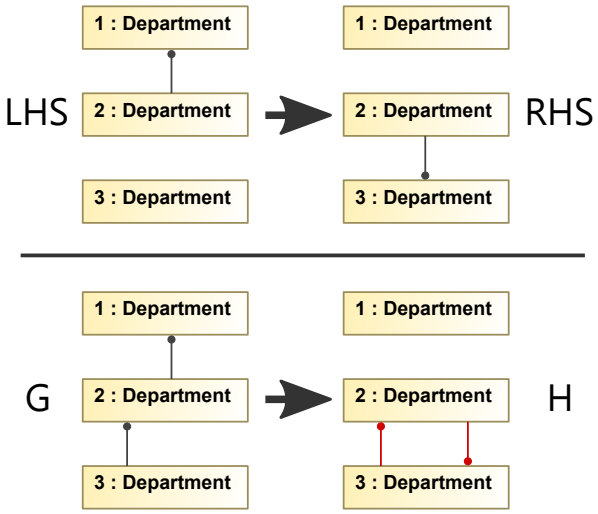


Figure 4.5: Inconsistent IC-rule leading to a cyclic containment.

Chapter 5

Transformation of Composite Graphs

This chapter presents the formalization of the composite modeling concept with explicit import and export interfaces which has been informally introduced in Sec. 2.2 and Sec. 2.3. With graphs representing the underlying structure of models (see Sec. 3.1 and Sec. 3.2), composite models and composite model transformation are formalized by composite graphs (see Sec. 5.2) and composite graph transformation (see Sec. 5.3), respectively. In order to focus on the composition, however, for now all concepts are given in its pure form, i.e., formalisms do not explicitly consider typing although typing may be incorporated in a straightforward way. Especially additional structures like inheritance and containment are not considered until the subsequent Chap. 6.

Composite graphs are initially proposed in [44]. They essentially rely on distributed graphs introduced by Taentzer in [77] which considers distributed graphs on the network layer describing the overall composition structure and on the object layer which specifies local graph structures and their interrelation. This twofold view is applied on composite graphs as well. A first version of the related composite graph transformation is introduced in [45] and is further elaborated in [47]. Both are also heavily influenced by the concepts of distributed graph transformation in [77].

A composite graph is constituted by a set of *component*s[1] each consisting of a *body graph* and a set of interface graphs, namely *export and import graphs*. While export interfaces contain graph elements that are provided to the environment, import interfaces specify required elements. Each interface graph maps into its body graph by a graph morphism. Furthermore, each import interface also maps to an export interface which constitutes interrelations between components. The present approach requires graph morphisms to be total which entails that all elements of import interface graphs are mapped to elements in export interface graphs, i.e., imports are always fully served

[1] The formal definition of *component* is omitted as its boundaries are each implicitly given by a body and its interfaces.

by connected exports. In general, however, the composition concept may also allow partial morphisms between interfaces yielding import interfaces being not fully served (see [44]). Such cases may then be interpreted as some sort of inconsistency.

The remaining chapter is organized as follows: Section 5.1 informally outlines the concepts of distributed graph transformation. Afterwards, Sec. 5.2 and Sec. 5.3 introduce composite graphs and composite graph transformation in detail, respectively. Finally, a special kind of composite graph transformation is provided in Sec. 5.4, called *weak*, which offers additional freedom in the use of composite models.

5.1 Distributed Graph Transformation

The concept of distributed graph transformation as exploited and extended in this thesis has been introduced by Taentzer in [77]. Taentzer essentially models distributed systems by means of distributed states and distributed actions where states and actions are represented in form of structured graphs and graph transformations, respectively.

Distributed graphs are considered on two levels: On *network level* a graph, called *network graph*, describes the distribution topology by network nodes and network edges between them. On local/*object level* each network node is refined by a *local graph* which represents the local state of the system, i.e., a local graph in turn consists of (local) nodes (objects) and edges (link) between them. Network edges are then refined by *graph morphisms* between local graphs which map corresponding elements between local graphs.

Distributed graphs as presented in [77] exhibit a special structural property: Some local graphs are interpreted as *interface graphs* which carry the elements being shared between two other local graphs (compare with component models with common interfaces in Sec. 2.1.1). There are consequently always two total morphisms from interface graphs to their two connected graphs. Connections between interface graphs are not considered, though.

Example 5.1. Figure 5.1 illustrates a distributed graph. The network graph is constituted by the big circles being network nodes with network edges as thick arrows between them. Refining local graph nodes and edges are directly embedded in the ellipses. Local graph morphism are not illustrated in favor of readability but node/edge mappings are encoded via layout position. The interface graph in the center connects the two exterior local graphs while carrying the correspondingly shared elements. △

One important result in [77] is that pushouts over a distributed graph can be constructed component-wise, i.e., the transformation of distributed graphs can be led back to a number of coordinated ordinary graph transformations. In detail, a distributed transformation includes the transformation of the network

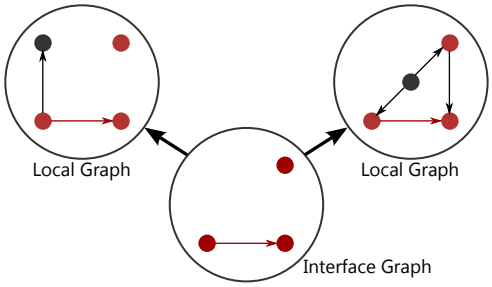


Figure 5.1: Distributed graph: Interface graph connecting two local graphs.

graph, on the one hand, which is an ordinary transformation since the network graph can be considered as plain graph. On the other hand and in correspondence with the network match, each local graph is transformed independently via an ordinary graph transformation again.

Section 3.2 describes the application of ordinary graph transformation in detail, exposes difficulties concerning the existence of pushouts and explains how the so-called gluing condition can help. It is obvious that similar problems may occur in the context of distributed graph transformation as well. For this reason, the *distributed gluing condition* [2] is defined requiring the ordinary gluing condition to hold in terms of the transformations of the network graph and of each local graph. It additionally comprises the so-called *connection conditions* which ensures that local graph elements are created/deleted such that all morphisms remain total. Moreover, the *network condition* ensures that network nodes/edges always have a correspondence on object level and vice versa, e.g., the deletion of a network node requires the deletion of the whole corresponding local graph including all of its elements.

In [26], Ehrig et al. consider distributed graphs transformation from a high-level categorical perspective where not only graphs may be distributed but arbitrary graph-like structures. That means that distributed graphs are generalized to distributed objects such that pushout diagrams consist of objects and morphisms in a certain category C. The *componentwise pushout* construction can be essentially characterized as gluing of distributed objects along their common sub-structure which may also require the extension of pushout objects with additional elements. This differs from the approach presented in [77] where additional properties in form of the gluing condition limit possible pushouts at which the extension of pushout objects is not necessary.

Example 5.2. A sample componentwise pushout as proposed in [26] is illustrated in Fig. 5.2. Since the graph in L occurs in R and G, it is preserved, i.e., it occurs in H as well. Due to G, the graph is also extended by a node

[2] Note that in contrast to the approach in this thesis distributed graph transformation as defined in [77] does not limit rule morphisms to be injective. Therefore their gluing condition comprises additional constraints not used here.

4 and an edge running from 1 to 4. In addition, R introduces a new graph and a relation from the preserved graph to the new one. In order to satisfy the totality of morphisms on object level, the new graph has to be extended by a node 4 and an edge although there are no correspondences in R for these elements. △

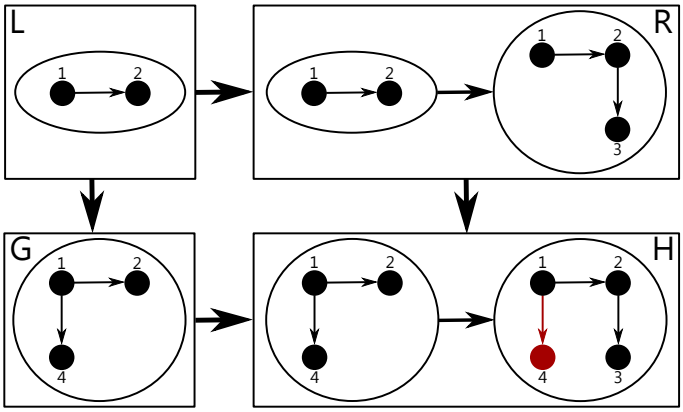


Figure 5.2: Componentwise pushout with pushout object extension.

Composite graphs and composite graph transformations as defined below adapt and extend parts of the concepts from Taentzer and use the work of Ehrig et al. for reasoning purposes. Particularly, the twofold layer concept of distributed graphs is heavily reused in the definition of composite models, i.e., the network layer as topology and the object layer as refinement. In contrast, composite models use a more subtle network structure which introduces two kinds of interfaces, export interfaces and import interfaces, in order to connect components. Composite model transformation can be considered as specialization of the general approach in [26]. Due to the specific structure of the network graph, the extension of pushout objects is not required, though.

5.2 Composite Graphs

This section introduces composite graphs whose concepts are based on distributed graphs [77] (cf. Sec. 5.1). Some formalizations are used in combination with results by Ehrig et al. in [26] in terms of category theoretical reasoning.

Composite graphs accomplish the modularization of simple (typed) graphs and inherit the twofold consideration of network layer and object layer from the concept of distributed graphs (see above). The network layer describes the topology of interconnected graphs while on object layer each graph with its elements can be found as well as the morphisms between the graphs. Below,

the network layer is specified at first followed by relating refinements by means of graphs and morphisms on object layer.

The network layer of composite graphs is specified by composite network graphs which are actually graphs typed over a dedicated type graph Net. Due to the type graph, nodes of a composite network graph are either body nodes, export nodes, or import nodes. Each export and import node is the source of exactly one edge running to a body. In addition, an import node has an outgoing edge running to an export node. Body nodes have no outgoing edges at all but can have an arbitrary number of incoming edges. Export nodes may have multiple incoming edges with import nodes as origin. Composite network graphs and morphisms between them, composite network graph morphisms, are shown to form a category, called CompoNetGraphs.

While such network structures are rather restrictive, so-called weak composite network graphs ease the restrictions such that exports may occur without a connected body. This is particularly used in transformation rules later on.

Definition 5.3 (Composite network graph)**.** Let Net be a graph consisting of three nodes, namely Body, Export, and Import and three edges running from Export to Body, from Import to Body, and from Import to Export (cf. Fig. 5.3). A *composite network graph* is a graph $G = (G_N, G_E, s_G, t_G)$ typed over Net. An element of G_N is called *network node* while an element of G_E is called *network edge*. For convenience the following sets are defined:

- $G_{Bod}, G_{Exp}, G_{Imp}$ are the disjoint sets of *bod*y nodes, *exp*ort nodes, and *imp*ort nodes with $G_N = G_{Bod} \uplus G_{Exp} \uplus G_{Imp}$.

- G_{EB}, G_{IB}, G_{IE} are the disjoint sets of edges running from *e*xport to *b*ody, *i*mport to *b*ody, and *i*mport to *e*xport with $G_E = G_{EB} \uplus G_{IB} \uplus G_{IE}$.

In addition, each composite network graphs has to fulfill the following constraints:

1. $\forall X \in \{EB, IB, IE\}$: (Unique edges)
   $\forall e1, e2 \in G_X : s_G(e1) = s_G(e2) \Rightarrow e_1 = e_2$,

2. a) $\forall n \in G_{Exp} : \exists e \in G_{EB} : n = s_G(e)$, and (Outgoing edges)
   b) $\forall n \in G_{Imp} : \exists e1 \in G_{IB} \wedge \exists e2 \in G_{IE} : n = s_G(e1) = s_G(e2)$.

If constraint 2a is violated, G is called *weak*. For convenience, $G_{\overline{Exp}}$ defines the set of export nodes without bodies, i.e., $G_{\overline{Exp}} = \{n \in G_{Exp} \mid \nexists e \in G_{EB} : s_G(e) = n\}$.

◇

Remark 5.4. As for the type graph Net, all constraints in Def. 5.3 above can be denoted by UML-like multiplicities as depicted in Fig. 5.3. Weak composite graphs would yield an edge running from Export to Body with multiplicity 0..1 instead of 1. ▽

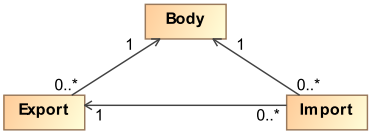


Figure 5.3: Type graph Net for composite network graphs in a UML-like fashion with multiplicities.

Example 5.5. An example composite network graph is shown in Fig. 5.4. Each node denotes its type after a colon. Typing of network edges can be uniquely deduced. For clarity, dashed edges indicate network edges between interfaces and their bodies and dotted edges occur between interfaces only. Furthermore, surrounding rounded rectangles group nodes of the same component. The component on the left-hand side consists of a body node and three export interface nodes while the component on the opposite side has one export node and two import nodes. As shown within the component on the right-hand side, relations between interfaces may not only occur between different components but also inside one and the same. In the following, it becomes apparent that this makes sense for type graphs only. △

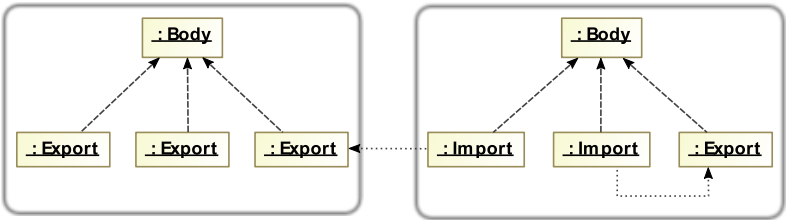


Figure 5.4: Example for a composite network graph.

Definition 5.6 (Composite network graph morphism)**.** Let G and H be two composite network graphs as defined above and let their typing over Net be implemented via $type_G \colon G \to Net$ and $type_H \colon H \to Net$. A graph morphism $f \colon G \to H$ is a *composite network graph morphism* if $type_H \circ f = type_G$. ◇

Proposition 5.7 (Category CompoNetGraphs)**.** Composite network graphs and network graph morphisms form the category CompoNetGraphs which has pushouts.

PROOF. Composite network graphs and network graph morphisms are graphs and graph morphisms typed over graph Net. Fact 3.9 states that typed graphs and graph morphisms form a category and Fact 3.11 states that such a category of graphs and graph morphisms has pushouts.

It is straightforward to show that compositions of network graph morphisms are again network graph morphisms since their definition does not comprise additional constraints compared to conventional typed graph morphisms. Therefore, it is obvious that COMPONETGRAPHS is a category.

Pushouts in this category are constructed as for typed graphs. It is left to show that the constraints for composite network graphs given in Def. 5.3 are satisfied for pushout graphs. Let $a\colon A \to B$ and $c\colon A \to C$ be composite network graph morphisms. The pushout morphisms are $b\colon B \to D$ and $d\colon C \to D$. The *unique edges* constraint for PO-graph D is shown at first: $\forall e, e' \in D_{EB}\colon s_D(e) = s_D(e') \wedge t_D(e), t_D(e') \in D_{Bod} \implies e = e'$. If $e \neq e'$, it can be assumed wlog. that $e \in b_E(B_E - A_E)$ and $e' \in d_E(C_E - A_E)$ since A, B, and C are composite network graphs. Then, either

1. $s_D(e) \in b_N(B_N - A_N)$ and $s_D(e') \in d_N(C_N - A_N)$ and due to PO-construction $s_D(e) \neq s_D(e')$ which contradicts the assumption or

2. $s_D(e) = b_N(a_N(x))$ and $s_D(e') = d_N(c_N(x))$ with $x \in A_N$. Then, there has to be an outgoing edge from x to some body node which has to be mapped to e and e'. Thus, $e = e'$ which contradicts the assumption.

This can be analogously shown for edges in D_{IB} and D_{IE}. Now, the *outgoing edge* constraint is shown to hold: $\forall n \in D_{Exp} : \exists e \in D_{EB}$ with $n = s_D(e)$. Assuming that there is no such e, then either

1. $n \in b_N(B_N - A_N)$ and since B is a composite network graph $\exists e \in b_E(B_E - A_E)$ which contradicts the assumption, or

2. $n = b_N(a_N(x))$ with $e'\colon x \to y \in A_E$ since A is a composite network graph. Due to the assumption and the PO-construction it follows $\nexists e''\colon a_N(x) \to a_N(y) \in B_E$ which contradicts the fact that B is a composite network graph.

Analogously, this can be shown for nodes in D_{Imp}.

Obviously, weak composite network graphs and network graph morphisms form a category as well, called COMPONETGRAPHS^{weak}. However, note that that category has no pushouts in general since resulting pushout graphs may violate the unique edge constraint (see Example 5.8).

□

Example 5.8. The diagram in Fig. 5.5 shows an invalid pushout over weak composite network graphs. Composite network graphs A, B, C, and D are illustrated by rounded rectangles containing nodes and edges. Mappings between nodes are denoted by numbers while edge mappings are omitted but can be uniquely deduced. Obviously, graph D is no (weak) composite network graph as the constraint *unique edges* (see Def. 5.3.1) is violated by the export node pointing to two different body nodes.

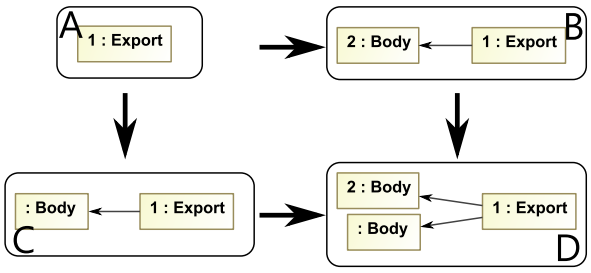


Figure 5.5: Invalid pushout in the category CompoNetGraphs^{weak} which does not lead to a valid (weak) composite network graph D.

Further examples can be constructed. Consider another pushout similar to the one in Fig. 5.5 where A contains an additional network node 2: *Body* not being connected by 1: *Export* and where the : *Body* node in C is actually 2: *Body*. Then, the pushout D contains the nodes 1 and 2 with two edges running from 1 to 2. This violates the *unique edges* constraint again. △

Now that the kind of distribution structure is specified, it follows the definition of their refinements by graphs and graph morphisms yielding composite graphs. Afterwards composite graph morphisms are defined to enable mappings between composite graphs. Note the last constraint in the definition below; it requires commutativity of import and export mappings in the following sense: A body element a being exported and in turn, imported by the same component has to be mapped to a again. Such a case is illustrated in the running example and discussed later on.

Definition 5.9 (Composite graph)**.** Given a composite network graph G, a *composite graph* $\hat{G}$ over G is defined as $\hat{G} = (G, \mathcal{G}(G), \mathcal{M}(G))$ with

- $\mathcal{G}(G)$ being a set of graphs, called *local graphs*, with each graph uniquely refining a network node in G_N: $\mathcal{G}(G) = \{\hat{G}(n) |\ \hat{G}(n)$ is a simple graph and $n \in G_N\}$,
- $\mathcal{M}(G)$ being a set of graph morphisms, called *local (graph) morphisms*, each refining a network edge in G_E: $\mathcal{M}(G) = \{\hat{G}(e) \colon \hat{G}(i) \to \hat{G}(j) | \hat{G}(e)$ is a graph morphism and $e \in G_E$ with $s(e) = i$ and $t(e) = j\}$.

- $\forall e_{IE}, e_{EB}, e_{IB} \in G_E\colon t_G(e_{EB}) = t_G(e_{IB}) \wedge s_G(e_{EB}) = t_G(e_{IE}) \wedge s_G(e_{IE}) = s_G(e_{IB})$ require that $\hat{G}(e_{IB}) = \hat{G}(e_{EB}) \circ \hat{G}(e_{IE})$. (commutative morphisms)

$\hat{G}$ is called *weak composite graph* if its network graph G is weak.

◇

Definition 5.10 (Composite graph morphism)**.** Given two composite graphs $\hat{G}$ and $\hat{H}$ with composite network graphs G and H, resp., a *composite (graph) morphism*, written $\hat{f}\colon \hat{G} \to \hat{H}$, is a pair $\hat{f} = (f, m)$ where

- $f\colon G \to H$ is a composite network graph morphism and
- m is a family of morphisms $\{\hat{f}(n) \mid n \in G_N\}$ such that
  - for all nodes $i \in G_N$, $\hat{f}(i)\colon \hat{G}(i) \to \hat{H}(f_N(i))$ is a graph morphism and
  - for all edges $e\colon i \to j \in G_E$, $\hat{H}(f_E(e)) \circ \hat{f}(i) = \hat{f}(j) \circ \hat{G}(e)$ (see the illustration in Fig. 5.6).

If morphism f and all morphisms in m are inclusions (injective), $\hat{f}$ is called inclusion (injective). ◇

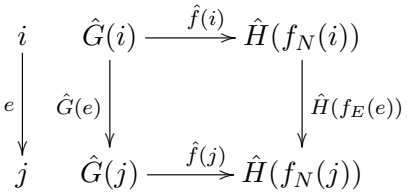


Figure 5.6: Illustration of a composite graph morphism.

Remark 5.11. Composite graphs and graph morphisms as defined in Defs. 5.9 and 5.10 form a category, called CompGraphs. It is a functor category (cf. [66]) where composite network graphs are considered as (small) categories such that nodes induce category objects and edges induce morphisms. In addition, identity morphisms and all compositions of defined morphisms belong to graph-induced categories. ▽

To show the co-completeness of the category CompGraphs, it is compared with the functor category DisC of distributed objects and morphisms as introduced in [26]. If the underlying category C is Graphs, CompGraphs is a full sub-category of DisC since composite network graphs are allowed only. The basic definitions of that approach are recalled in the Appendix. Distributed objects generalize composite graphs in two ways: (1) Any network graph is allowed and (2) local structures can be of any category C, i.e., they do not have

to be graphs. Ehrig et al. show that the category of distributed objects and morphisms is co-complete under certain conditions. This result is exploited for the category CompGraphs.

In the following, it is also shown that the construction of pushouts in the category CompGraphs can be performed in a component-wise manner, i.e., the network pushout and all pushouts on local graphs can be computed separately.

Corollary 5.12 (Category CompGraphs is co-complete)**.** The category CompGraphs is co-complete. Pushouts along injective composite network morphisms can be constructed component-wise.

Proof. The initial object is the empty composite graph, i.e., an empty network graph which consequently has no refining graphs on object layer.

In Prop. 10.6 it is shown that CompGraphs is a sub-category of DisC with C being Graphs. According to Fact 10.7, the pushout over arbitrary distributed morphisms in DisC exists, if the base category is (co)complete. Since Graphs is (co)complete (cf. Fact 3.11), so is the category DisGraphs.

Proposition 5.7 shows that pushouts in CompoNetGraphs can always be constructed. This serves as basis for the reasoning that pushout of two (typed) composite graph morphisms along injective composite network morphisms not only exists but can be constructed component-wise. In [26, Prop. 4], the authors show that for each two persistent network morphisms the pushout within their underlying distributed category can be constructed component-wise. Since injective composite network morphisms are always persistent as shown in Prop. 10.9, pushouts in the category CompGraphs along injective composite network morphisms can be constructed component-wise. □

Remark 5.13. Analogously, it can be shown that CompGraphs_{TG} is co-complete: The category CompGraphs_{TG} can be considered a sub-category of DisC with C being the category of typed graphs and typed graph morphisms, Graphs_{TG}. Since Graphs_{TG} is (co)complete, CompGraphs_{TG} has pushouts and as the network structure is equal to that in CompGraphs, pushouts can be constructed component-wise. ▽

Proposition 5.14 (CompGraphs and CompGraphs_{TG} are adhesive categories)**.** The categories CompGraphs and CompGraphs_{TG} are adhesive categories, where the monomorphisms in CompGraphs and CompGraphs_{TG} are the injective composite graph morphisms and typed composite graph morphisms, respectively.

Proof. According to [23, Thm. 4.15 item 3], given a (weak) adhesive category $(\mathrm{C}, \mathcal{M})$ one can construct a functor category $([\mathrm{X}, \mathrm{C}], \mathcal{M}')$ which is particularly a (weak) adhesive category for every category X if $\mathcal{M}'$ is a natural transformation $t\colon F \to G$ where all morphisms $t_X\colon F(X) \to G(X)$ are in $\mathcal{M}$.

This construction directly applies to CompGraphs with injective morphisms where CompGraphs is a functor category (cf. Remark 5.11) with $(\mathrm{C}, \mathcal{M})$ being the adhesive category Graphs with injective graph morphisms (cf. Fact 3.12) and with X corresponding to CompoNetGraphs. Then, all morphisms $t_X \colon F(X) \to G(X)$ are injective graph morphisms and therefore in $\mathcal{M}$.

This analogously holds for the category CompGraphs_{TG} at which $\mathcal{M}$ is the class of injective typed graph morphisms. □

5.3 Composite Graph Transformation

While previous sections laid the basis for the transformation of single graphs and the distribution of multiple interrelated graphs, this section elaborates on graph transformation in such a distributed environment. It particularly focuses on the transformation of composite graphs without additional inheritance and containment structures for now.

At first, appropriate rules and transformation steps are defined. Since composite graphs comprise (interrelated) graphs on network layer and object layer, the conventional graph transformation gluing condition (cf. Def. 3.17) is insufficient. Special gluing conditions are required and proposed to ensure the applicability of composite transformation steps. Then, pushout(s) regarding the transformation of composite graphs are constructed.

Definition 5.15 (Composite graph rule)**.** A *composite graph rule* $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{K} \xrightarrow{\hat{r}} \hat{R})$ is a rule in the category CompGraphs with $\mathcal{M}$ being the class of injective composite graph morphisms. ◇

Remark 5.16. Each composite graph rule induces graph rules on two layers:

- On network layer, one rule, called *network rule*, is induced in the category Graphs_{Net}.
- Each network node in K induces a corresponding rule on object layer, called *local rule*, in the category Graphs with injective graph morphisms, i.e., $\forall n \in K_N \colon \hat{p}(n) = (\hat{L}(n) \xleftarrow{\hat{l}(n)} \hat{K}(n) \xrightarrow{\hat{r}(n)} \hat{R}(n))$ is a graph rule.

▽

Definition 5.17 (Composite graph transformation step)**.** A *composite graph transformation step* (or direct composite graph transformation) $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ of a composite graph $\hat{G}$, called *host graph*, to $\hat{H}$ by a composite graph rule $\hat{p}$ and an injective composite morphism $\hat{m} \colon \hat{L} \to \hat{G}$, called *match*, is given in Fig. 5.7 below, where (1) and (2) are pushouts in the category CompGraphs with injective morphisms only. ◇

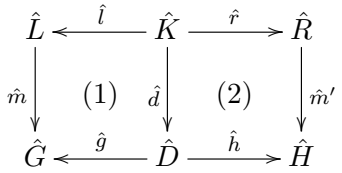


Figure 5.7: Illustration of a composite graph transformation step by means of two pushouts.

Remark 5.18. The double pushout in the category CompGraphs exists since CompGraphs is co-complete and pushouts are constructed componentwise (see Corollary 5.12). Moreover, the double pushout in the category CompGraphs$_{TG}$ exists as well. ▽

Remark 5.19. Again, analogously to Remark 3.21 and 4.17, composite graph rules may be constructed by means of inclusions. Below, only such composite rules are considered for the sake of simplicity. ▽

Similar to graph transformation steps in a non-distributed environment, composite graph transformation steps are not applicable in general. First of all, since composite graph transformations are performed componentwise, i.e., actually a number of ordinary (local) graph transformations are performed, the gluing condition for conventional graph transformation as defined in Def. 3.17 must hold. However, this is obviously not sufficient due to composite setting, i.e., the specific structure of network graphs and the (total) morphisms between graphs on object layer.

Definition 5.20 (Gluing condition for composite graphs)**.** Let there be a composite graph rule $\hat{p} = (\hat{L} \supseteq \hat{K} \subseteq \hat{R})$, a composite graph $\hat{G}$, and a composite morphism $\hat{m}\colon \hat{L} \to \hat{G}$. Furthermore, let L^{Del}, R^{Crt}, and G^{Ind} be defined on network level while corresponding definitions on object level are straightforward, e.g., $\hat{L}_N^{Del}(x)$ specifies the nodes to be deleted on object level with x being the corresponding network node which is not necessarily to be deleted itself; analogously for edges and network edges. $\hat{p}$ and $\hat{m}$ satisfy the *gluing condition for composite graphs* if the following hold:

1. $m\colon L \to G$ satisfies the gluing condition (see Def. 3.17) wrt. $p = (L \supseteq K \subseteq R)$ (network gluing condition)
2. $\forall x \in K_N : \hat{m}(x)$ satisfies the gluing condition (see Def. 3.17) wrt. $\hat{p}_x = (\hat{L}(x) \supseteq \hat{K}(x) \subseteq \hat{R}(x))$ (local gluing condition)
3. $\forall x \in L_N^{Del} : \hat{m}(x)$ is bijective (deletion of network node)
4. $\forall y \in K_N, \forall \hat{y} \in \hat{L}^{Del}(y)\colon \forall \hat{e}\colon \hat{x} \to \hat{m}(\hat{y}) \in \hat{G}(e)$ with $e \in G_E : \exists \hat{x}'$ with $\hat{m}(\hat{x}') = \hat{x}$ (deletion on target side)

Then, $\hat{m}$ is called a *match*. ◇

The composite gluing condition defined above ensures the valid transformation on different layers. The network and local gluing conditions (items 1 and 2) prevent dangling edges in network graphs and in local graphs on object layer. Item 3 ensures that the deletion of a network node yields the deletion of its entire refining local graph. Therefore, every single element of the local graph has to be captured by the match. The last item 4 avoids dangling mappings on object layer in the following sense. Assuming a mapping between two graph elements, the deletion of the target element requires the source element to be part of the rule as well. Since morphisms have to be total, the mapping between both elements is consequently part of the rule as well and therefore deleted together with the target element.

It remains to show that obeying the composite gluing condition ensures the existence and uniqueness of the context graph since this is the prerequisite for a successful transformation. At first, the pushout complement is constructed which comprises the context graph $\hat{D}$ and the morphisms $\hat{d}$ and $\hat{g}$ (cf. (1) in Fig. 5.7). Afterwards, it is shown that when taking the composite gluing condition into account the composite graph transformation step is always applicable and unique. Note that the construction and reasoning below orientate on those for ordinary graph transformation in [77, p. 127ff.].

Definition 5.21 (Composite context graph)**.** Given a composite rule $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R})$ and a match $\hat{m}\colon \hat{L} \to \hat{G}$ as shown in Fig. 5.7, the (composite) context graph $\hat{D}$ is constructed as follows:

- *Network layer:* Let $D = G \setminus m(L^{Del})$ be the context graph of p and m in the category CompoNetGraphs. Then, $d = m \circ l$ and $g = id_D$.

- *Object layer:*
  $$\hat{D}(x) = \begin{cases} \hat{G}(x) \setminus \hat{m}\left(\hat{L}^{Del}(y)\right) & \text{, if } \exists y \in K_N \text{ with } m(y) = x \\ \hat{G}(x) & \text{, if } x \in G_N^{Ind} \\ \hat{g}^{-1}(t(x)) \circ \hat{G}(x) \circ \hat{g}(s(x)) & \text{, if } x \in D_E \end{cases}.$$
  Furthermore, $\hat{d}$ is defined by $\hat{d}(y) = \hat{m}(y) \circ \hat{l}(y)$, $\forall y \in K_N$, and $\hat{g}(x) = id_{\hat{D}(x)}$, $\forall x \in D_N$.

◇

Remark 5.22 (Pushout complement)**.** Related to Def. 5.21 above, $\forall y \in K_N$ and $d(y) = x$ the pushout complement $\left(\hat{D}(x), \hat{d}(y), \hat{g}(x)\right)$ of $\hat{l}(y)$ and $\hat{m}(y)$ is constructed analogously in the category Graphs (see [23, Fact 3.11]) since composite transformations can be considered component-wise. ▽

Proposition 5.23 (Applicability of composite graph transformation steps)**.** Given a composite rule $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R})$, a composite match $\hat{m}\colon \hat{L} \to \hat{G}$ which satisfies the composite gluing conditions, and the context graph $\hat{D}$ as well as the composite graph morphisms $\hat{d}$ and $\hat{g}$ as constructed in Def. 5.21, then $(\hat{G}, \hat{m}, \hat{g})$ is the pushout of $\hat{d}$ and $\hat{l}$ in CompGraphs (see Fig. 5.7). Furthermore, the pushout of $\hat{d}$ and $\hat{r}$ exists.

Proof. At first it is shown that $\hat{D}$ is a composite graph (1) and that the morphisms $\hat{d}$ and $\hat{g}$ are composite graph morphisms (2). Then the uniqueness of $\hat{D}$ is shown (3). Afterwards, the pushout properties according to $(\hat{G}, \hat{m}, \hat{g})$ are shown, i.e., commutativity (4) and the universal property (5). Last but not least, the pushout of $\hat{d}$ and $\hat{r}$ yielding $(\hat{H}, \hat{m}', \hat{h})$ is shown to exist (6).

(1). Obviously, D is a graph due to the network gluing condition (5.20.1). In particular, D is a composite network graph (see Def. 5.3): Since $D = G \backslash m(L^{Del})$, i.e., D is a sub-graph of G, ambiguous edges cannot be introduced which satisfies constraint 5.3.1. It remains to check that all outgoing edges required by certain types of network nodes do exist. Assuming that $\exists n \in D_{Exp}\colon \nexists e\colon n \to b \in D_{EB}$ (cf. Def. 5.3.2), then $\exists e'\colon n' \to b' \in L_E^{Del}$ with $m(e') = e \wedge m(n') = n \wedge m(b') = b$. And since K is a network graph, from $e' \notin K_E$ it follows that $n' \notin K_N$, i.e., $n' \in L_N^{Del}$ and thus $n \notin D$ which contradicts the assumption. Analogously for outgoing edges of import nodes.

Next, the object layer is checked: The local gluing condition 5.20.2 ensures that each local transformation step yields a (local) graph $\hat{H}(x)$ for all $x \in H$. The constraint 5.20.3 ensures that the deletion of a network node also leads to the deletion of its entire refining graph on object level by requiring all corresponding elements to be part of the rule. In this case, (local) morphisms are properly deleted as well since outgoing and incoming network edges are properly deleted (see above), i.e., they are part of the rule's left-hand side which is a composite graph. It is left to show that graph morphisms between preserved graphs remain sound.

Let there be a network edge $f\colon a \to b \in K_E$ which matches $e\colon x \to y \in D_E$ via $m_E(f) = e$. Furthermore, let $\hat{e}\colon \hat{x} \to \hat{y} \in \hat{G}(e)$ be a mapping from a local node $\hat{x} \in \hat{G}(x)_N$ to $\hat{y} \in \hat{G}(y)_N$. The following cases can be distinguished:

- (Deletion on source side). If $\hat{x} \notin \hat{D}(x)_N$ then $\exists \hat{a} \in \hat{L}^{Del}(a)_N\colon \hat{x} = \hat{m}(\hat{a})$. Since $\hat{L}$ is a composite graph, there is a mapping $\hat{f}\colon \hat{a} \to \hat{b}$ with $\hat{m}(\hat{f}) = \hat{e} \wedge \hat{m}(\hat{b}) = \hat{y}$. Then since $\hat{a} \notin \hat{K}(a)$ and $\hat{K}$ is a composite graph it follows that $\hat{f} \notin \hat{K}(f)$, i.e., $\hat{f} \in \hat{L}^{Del}(f)$ and therefore the mapping $\hat{m}(\hat{f})$ is deleted as well.

- (Deletion on target side). If $\hat{y} \notin \hat{D}(y)_N$ then $\hat{y} = \hat{m}_N(\hat{b})$ with $\hat{b} \in \hat{L}^{Del}(b)_N$. Due to constraint 5.20.4, for each local node $\hat{x}$ mapped to $\hat{y}$ via an $\hat{e}$ there is a corresponding $\hat{a} \in \hat{L}(a)_N$ with $\hat{m}_N(\hat{a}) = \hat{x}$ and since $\hat{L}$ is a composite graph there is also a mapping $\hat{f}\colon \hat{a} \to \hat{b}$ which particularly corresponds to $\hat{e}$. Since $\hat{b} \notin \hat{K}(b)$ and $\hat{K}$ is a composite graph, $\hat{a} \notin \hat{K}(a)$ and $\hat{f} \notin \hat{K}(f)$, i.e., $\hat{a} \in \hat{L}^{Del}(a)$ and $\hat{f} \in \hat{L}^{Del}(f)$. This shows that local source nodes and mappings are deleted together with a local target node.

- (Removal of mappings only). Assume that $\hat{e} \notin \hat{D}(e)$ but $\hat{x} \in \hat{D}(x)$ and $\hat{y} \in \hat{D}(y)$. Then there exists an $\hat{m}(\hat{f}) = \hat{e}$ with $\hat{f}\colon \hat{a} \to \hat{b} \in \hat{L}^{Del}(f)$. It follows that $\hat{f} \notin \hat{K}(f)$ but $\hat{a} \in \hat{K}(a)$ and $\hat{b} \in \hat{K}(b)$ which contradicts the fact that $\hat{K}$ is a composite graph.

(2). Since $\hat{l}$ and $\hat{m}$ are composite graph morphisms, so is their composition $\hat{d}$ (see Remark 5.11). Morphism $\hat{g}$ is the identical morphism $id_{\hat{D}(x)}$ which is also a composite graph morphisms since D is shown to be a composite graph.

(3). According to the properties of (weak) adhesive HLR categories (cf. [23, Thm. 4.26]), the pushout complement is unique since CompGraphs is adhesive and $\hat{l}$ is in $\mathcal{M}$.

(4). Commutativity, i.e., $\hat{m} \circ \hat{l} = \hat{g} \circ \hat{d}$ is easy to show: Since $\hat{g}$ is defined as the identical morphism $id_{\hat{D}(x)}$, it follows that $\hat{m} \circ \hat{l} = \hat{d}$ which is exactly the definition of $\hat{d}$ (cf. Def. 5.21).

(5). In order to prove $(\hat{G}, \hat{m}\colon \hat{L} \to \hat{G}, \hat{g}\colon \hat{D} \to \hat{G})$ to be a pushout of $\hat{l}$ and $\hat{d}$, it has to be shown that for any $(\hat{X}, \hat{p}\colon \hat{L} \to \hat{X}, \hat{q}\colon \hat{D} \to \hat{X})$ with $\hat{p} \circ \hat{l} = \hat{q} \circ \hat{d}$ there is a unique morphism $\hat{x}\colon \hat{G} \to \hat{X}$ such that $\hat{x} \circ \hat{m} = \hat{p}$ and $\hat{x} \circ \hat{g} = \hat{q}$ (compare with Fig. 5.8). Let there be such an $\hat{X}$. $\hat{x}$ is defined as follows: For all $y \in G$ with $\exists y' \in L^{Del}\colon m(y') = y$, $\hat{x}(y) = \hat{p}(y')$. Furthermore, for all $y \in G^{Ind}$, $\hat{x}(y) = \hat{q}(y)$. It remains the case where $\forall y \in G$ with $y' \in L \setminus L^{Del}\colon m(y') = y$, i.e., $y' \in K$. Then, a unique $\hat{x}(y)$ exists due to Fact 3.20 (and [23, Fact 2.20]) which is induced by the pushout $(\hat{G}(y), \hat{m}(y'), \hat{g}(y))$ in Graphs.

The uniqueness of $\hat{x}$ follows from the fact that for each $y \in G$ only one of these three disjoint cases occurs. The well-definedness of $\hat{x}$ can be shown analogously to [77, Prop. 6.2.5] and is omitted here.

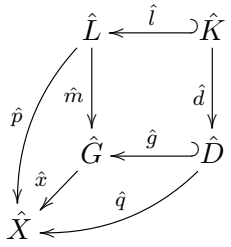


Figure 5.8: Illustration for the proof of Prop. 5.23.

(6). The pushout of $\hat{d}$ and $\hat{r}$ is constructed componentwise in Graphs. □

Proposition 5.24 (Uniqueness of composite transformation step)**.** Given a composite rule $\hat{p} = (\hat{L} \supseteq \hat{K} \subseteq \hat{R})$ and a composite match $\hat{m}\colon \hat{L} \to \hat{G}$ which satisfies the composite gluing conditions, the transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ in the category CompGraphs is unique up to isomorphism.

Proof. $\hat{H}$ is constructed componentwise analogously to Fact 3.20, i.e., set-theoretically as a disjoint union of $\hat{D}$ and $\hat{R}^{Crt}$. The uniqueness of $\hat{H}$ follows from the uniqueness of $\hat{D}$, shown in Prop. 5.23, and from the uniqueness of pushout objects. □

5.4 Transformation of Weak Composite Graphs

From a practical perspective, it may be desirable to specify composite rules in the category CompoNetGraphs^{weak}, i.e., with LHS and RHS being *weak* composite graphs. For example, consider the case where exported objects of a foreign component shall be imported in an owned component. On the one hand, the foreign body structure might not be of interest to the modeler, and on the other hand, it is actually not necessary to be known in order to establish the connection properly. Beyond that, the foreign body might even be intentionally hidden from the modeler for some reasons. Note that the body exists though.

Transformation steps in CompoNetGraphs^{weak} face a number of challenges. One has already been demonstrated in Example 5.8 where pushouts may not necessarily yield valid (weak) composite network graphs on the network layer. Moreover, while LHS and RHS may be proper weak composite graphs as motivated above, rule applications shall be understood as taking place on weak composite graphs which represent selected elements of non-weak composite graphs. In order to maintain a valid composite structure in the resulting graph (even if it is just a selection of a composite graph by means of a weak composite graph), invalid operations on network layer as well as on object layer need to be prohibited. For example, on network layer a rule with weak composite LHS and RHS graphs can be easily defined which deletes the edge between an export node and its body node. The related pushout in CompoNetGraphs^{weak} is obviously a valid weak composite graph, again. However, understanding the weak composite host graphs as selection of a composite graph, the resulting graph is no composite graph anymore. Even if the network layer does not contain any deletion/creation of export-body network edges, on object layer objects in uncoupled export graphs may be created or deleted which also prevent the pushout to be a valid composite graph.

Consequently, weak composite graph rules have to obey certain constraints in order to ensure the existence of pushouts in CompoNetGraphs^{weak} on the one hand, and on the other hand to ensure that the resulting graph corresponds to a composite graph with the host graph being interpreted as composite graph.

Such constraints would essentially maintain the integrity of uncoupled exports. That means, on network level export nodes not being connected to a body node must not be deleted or created. This must hold as well on object level for all elements in corresponding refining export graphs. Furthermore, edges running from exports to bodies must not be deleted or created. Roughly spoken, such export interfaces may not be created, modified, deleted, and coupled to or decoupled from bodies. However, relations from import interfaces to such uncoupled exports are allowed to be created, modified, and deleted as this has no effect on the export itself nor on its represented elements in the body.

Definition 5.25 (Weak composite graph rule)**.** A *weak composite graph rule* $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R})$ is a composite rule as defined above where $\hat{L}$ and $\hat{R}$ (and thus $\hat{K}$) are weak composite graphs. In addition, the following constraints hold:

1. $\forall n \in K_{\overline{Exp}}$ it holds (export integrity)
   a) $n \in L_{\overline{Exp}}$,
   b) $n \in R_{\overline{Exp}}$, and
   c) $\hat{L}^{Del}(n)$ and $\hat{R}^{Crt}(n)$ are empty graphs,
2. $L^{Del}_{\overline{Exp}} = \emptyset$, and (export deletion)
3. $R^{Crt}_{\overline{Exp}} = \emptyset$. (export creation)

◇

The following gluing condition prevents the case where an uncoupled export and a body occurs in the LHS which are then matched towards an export and a body being interconnected. While this scenario does not make sense from a practical point of view, it may also lead to problems when the corresponding body in the LHS is smaller than the export.

Definition 5.26 (Gluing condition for weak composite graphs)**.** Let there be a weak composite graph rule $\hat{p} = (\hat{L} \supseteq \hat{K} \subseteq \hat{R})$, a (weak) composite graph $\hat{G}$, and a composite morphism $\hat{m}\colon \hat{L} \to \hat{G}$. $\hat{p}$ and $\hat{m}$ satisfy the *gluing condition for weak composite graphs* if

1. the gluing condition for composite graphs (cf. Def. 5.20) is satisfied and
2. $\forall n \in m(L_{\overline{Exp}})$ with $\exists e\colon n \to n' \in G_{EB} \Rightarrow n' \notin m(L_{Bod})$.

◇

Definition 5.27 (Weak composite graph transformation step)**.** A weak composite transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ is a transformation step in the category CompoNetGraphs^{weak} with a weak composite rule $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R})$, a weak composite graph $\hat{G}$ as host and a composite match $\hat{m}\colon \hat{L} \to \hat{G}$ which satisfies the gluing conditions for weak composite graphs. ◇

In the following, the applicability of weak composite transformation steps is led back to the applicability of composite transformation steps. The construction below prepares the related proposition by defining a conversion from weak composite graph transformation steps to non-weak composite graph transformation steps. To this end, participating graphs with uncoupled export interfaces are equipped with additional bodies being *surrogates* for the missing ones. Since weak composite rules require that uncoupled exports are fully preserved (compare Def. 5.25), their surrogate bodies must be preserved as well. This is an intuitive behavior for the introduced bodies as they are not the actual bodies but only represent them.

The conversion is performed is in the following steps: *Step 1* completes the host graph $\hat{G}$ to $\hat{G}'$, i.e., on network level for each export interface a new body is introduced and assigned while on object level the refining body graphs are constructed identical to their export graph. *Step 2*, on the one hand, completes the left-hand side $\hat{L}$ of the rule with respect to the host graph $\hat{G}'$ and the mappings provided by the match $\hat{m}$. On the other hand, the match $\hat{m}$ is correspondingly extended itself by mappings to new elements in $\hat{L}'$. First of all, both L and m are extended on network level. Therefore, L is extended by all the body nodes of G' whose export nodes are matched by an uncoupled export node in L. While these body nodes are actually the same in L' and G', the new body graphs in $\hat{L}'$ differ as they are the union of their export graphs glued together along those graph elements which map to the same body graph element in $\hat{G}'$. This is especially relevant, if two exports in the LHS are mapped to two exports in the host graphs which in turn point to a common (given) body. In *Step 3*, the gluing graph K and the right-hand side R are completed analogously to L as the additional surrogate bodies are fully preserved. This step also includes an adequate adaption of the morphisms between L', K', and R', namely $\hat{l}'$ and $\hat{r}'$.

Definition 5.28 (Weak composite graph transformation step conversion)**.** Given a weak composite transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ with the weak composite rule $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R})$, the weak composite graph $\hat{G}$ and the composite match $\hat{m}\colon \hat{L} \to \hat{G}$ which satisfies the gluing conditions for weak composite graphs. Then, the weak composite transformation step is converted to a composite transformation step $\hat{G}' \overset{\hat{p}',\hat{m}'}{\Longrightarrow} \hat{H}'$ as follows:

Step 1. Complete the weak composite graph $\hat{G}$ to composite graph $\hat{G}'$.

a) $G'_N = G_N \wedge G'_E = G_E$,

b) $G'_{Bod} = G_{Bod} \biguplus \{\overline{b}_n \mid n \in G_{\overline{Exp}}\}$,

c) $G'_{EB} = G_{EB} \biguplus \{\overline{e}_n \colon n \to \overline{b}_n \mid n \in G_{\overline{Exp}}\}$,

d) $\forall x \in G_N \bigcup G_E \colon \hat{G}'(x) = \hat{G}(x)$,

e) $\forall \overline{b}_n \in G'_{Bod} \setminus G_{Bod} \colon \hat{G}'(\overline{b}_n) = \hat{G}(n)$, and

f) $\forall \overline{e}_n \in G'_{EB} \setminus G_{EB} \colon \hat{G}'(\overline{e}_n) = id_{\hat{G}(n)}$.

Step 2. Complete the weak composite graph $\hat{L}$ to composite graph $\hat{L}'$ and the mapping $\hat{m} \colon \hat{L} \to \hat{G}$ to $\hat{m}' \colon \hat{L}' \to \hat{G}'$ (compare Figs. 5.9). This steps starts at the network level and proceeds at the object layer afterwards.

Let $\overline{B} = \{b \in G'_{Bod} \mid \exists e \colon a \to b \in G'_{EB} \wedge m^{-1}(a) \in L_{\overline{Exp}}\}$.

a) $L'_N = L_N \wedge L'_E = L_E \wedge \forall x \in L_N \bigcup L_E \colon m'(x) = m(x)$,

b) $L'_{Bod} = L_{Bod} \biguplus \overline{B}$,

c) $\forall b \in \overline{B} \colon m'(b) = b$,

d) $L'_{EB} = L_{EB} \biguplus \{\overline{e}_n \colon n \to b \mid n \in L_{\overline{Exp}} \wedge b \in \overline{B} \wedge \exists e \colon m(n) \to m(b)\}$,

e) $\forall e \colon n \to b \in L'_{EB} \setminus L_{EB} \colon m'(e) = e'$ with $\exists e' \colon m(n) \to b \in G'_{EB}$,

f) $\forall x \in L_N \bigcup L_E \colon \hat{L}'(x) = \hat{L}(x) \wedge \hat{m}'(x) = \hat{m}(x)$,

g) $\forall b \in \overline{B} \colon \hat{L}'(b) = \left(\biguplus_n \hat{L}(n)\right)_{/\sim} \colon \forall n \in \hat{L}_{\overline{Exp}}$ with $\overline{e}_n \colon n \to b \in L'_{EB}$ and with the equivalence relation $\sim = \{(x, y) \in \hat{L}'(n) \times \hat{L}'(n') \mid \forall n, n' \in L_{\overline{Exp}} \colon \hat{G}'(m'(\overline{e}_n)) \circ \hat{m}(n)(x) = \hat{G}'(m'(\overline{e}_{n'})) \circ \hat{m}(n')(y)\}$ (cf. Fig. 5.9a and Fig. 5.9b),

h) $\forall b \in \overline{B} \colon \forall x \in \hat{L}'(n)$ with $e \colon n \to b \in L'_{EB} \colon \hat{m}'(b)(x) = \hat{G}'(m'(e)) \circ \hat{m}'(n)(x)$ with $x \in [x]_\sim$ (cf. Fig. 5.9b),

i) $\forall \overline{e}_n \in L'_{EB} \setminus L_{EB} \colon \forall x \in \hat{L}'(n) \colon \hat{L}'(\overline{e}_n)(x) = [x]$

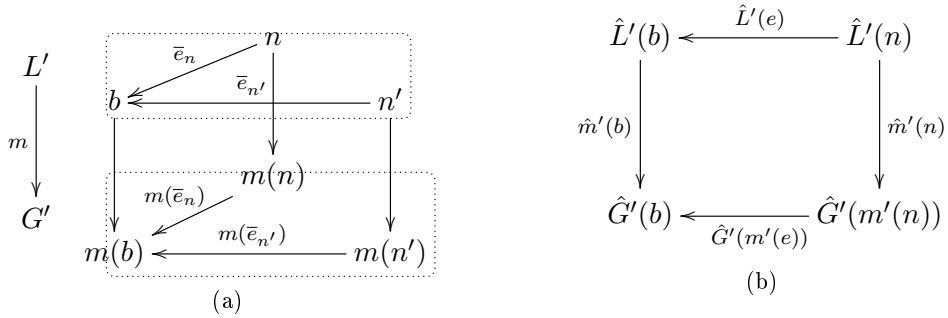


Figure 5.9: Illustrations for Step 2 of Def. 5.28.

Step 3. Based on $\hat{L}'$, complete the weak composite graphs $\hat{K}$ and $\hat{R}$ to $\hat{K}'$ and $\hat{R}'$, respectively (compare Fig. 5.10).

Let $\overline{B}$ be defined as above or at this point shorter $\overline{B} = L'_{Bod} \setminus L_{Bod}$ and let there be $\overline{E} = L'_{EB} \setminus L_{EB}$.

a) $K'_N = K_N \wedge K'_E = K_E \wedge R'_N = R_N \wedge R'_E = R_E$,

b) $K'_{Bod} = K_{Bod} \biguplus \overline{B}$ and $K'_{EB} = K_{EB} \biguplus \overline{E}$,

c) $R'_{Bod} = R_{Bod} \biguplus \overline{B}$ and $R'_{EB} = R_{EB} \biguplus \overline{E}$,

d) $\forall x \in R_N \bigcup R_E \colon \hat{R}'(x) = \hat{R}(x)$,

e) $\forall x \in K_N \bigcup K_E \colon \hat{K}'(x) = \hat{K}(x) \wedge l'(x) = l(x) \wedge \hat{l}'(x) = \hat{l}(x) \wedge r'(x) = r(x) \wedge \hat{r}'(x) = \hat{r}(x)$,

f) $\forall x \in \overline{B} \bigcup \overline{E} \colon l'(x) = x \wedge r'(x) = x \wedge \hat{K}'(x) = \hat{L}'(x) \wedge \hat{R}'(x) = \hat{L}'(x)$,

g) $\forall x \in \overline{B} \colon \hat{l}'(x) = id_{\hat{L}'(x)} \wedge \hat{r}'(x) = id_{\hat{L}'(x)}$,

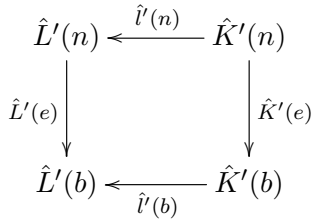


Figure 5.10: Illustration for Step 3 of Def. 5.28.

◇

It remains to show that the conversion is well-defined.

Lemma 5.29. In Def. 5.28, $\hat{G}'$, $\hat{L}'$, $\hat{K}'$, and $\hat{R}'$ are well-defined and also $\hat{l}'$, $\hat{r}'$, and $\hat{m}'$.

Proof. The well-definedness is shown with regard to each step in Def. 5.28.

Step 1: $\hat{G}'$ is well-defined. $\hat{G}'$ is obviously a composite graph. On network level, for each uncoupled export node in G a body node is created and assigned (cf. items b) and c)). Consequently, the property 2a of Def. 5.3 is satisfied, i.e., $\forall n \in G_{Exp} : \exists e \in G_{EB} : n = s_G(e)$. On object layer, the new body graphs are minimal since item e) defines them as being identical to their corresponding export graph. This is also reflected by item f) which appropriately adapts the related refinement of the export-body edge as the identical morphism.

The construction is unique since each new body directly corresponds to one export which is the one assigned via an export-body edge. Obviously, the construction can always be performed.

Note that the constructed $\hat{G}'$ is not necessarily isomorphic to the possibly underlying true composite graph of $\hat{G}$. For instance, consider the weak composite graph $\hat{G}$ with two uncoupled export nodes only. Then, the underlying composite graphs may contain these two exports pointing to a single body while the construction in Def. 5.28 yields two separate bodies. However, uncoupled exports and new bodies are fully preserved and just help to reuse the transformation step defined for composite graphs.

Step 2: $\hat{L}'$ and $\hat{m}'$ are well-defined. The first group of extensions are performed on network level . Item b) adds missing body nodes to L which are actually body nodes of the host composite graph $\hat{G}'$. Only those body nodes are added whose exports are matched by uncoupled export in L. Note that such a body node might have been created previously in Step 1. Accordingly, the match mappings in m' are immediately defined in item c) to match to the same body node. The new export-body edges in d) are inferred according to their edges between the matched export nodes and body nodes. This deduction is unique since there is only one edge running from an export node to a body nodes (see Def. 5.3) in composite graphs. It follows that the extensions of the match mappings for export-body edges in e) are unique and well-defined as well. At this point, the network graph L' is non-weak and m' maps all nodes and edges in a compatible way.

On object level, the refining graphs of new body nodes in L'_{Bod} are constructed by a union of their export graphs in $L'_{\overline{Exp}}$, performed in item g). Cases where targeted body elements of two (or more) exports intersect (cf. Fig. 5.9a) are handled by an equivalence relation which glues such elements to an equivalence class together. This equality is identified by the target body graph of those exports in $\hat{G}'_{Exp}$ being the matched images of the exports in

$\hat{L}'_{\overline{Exp}}$. The morphisms between the export graphs in $\hat{L}'_{\overline{Exp}}$ and the new body graphs are also well-defined since each morphism is total by construction and injective since the equivalence relation $\sim$ maps (if necessary) two elements of different export graphs to one element in a body graph (see item i)).

In the following, the well-definedness of $\hat{m}'$ is shown more formally. It is sufficient to show that the diagram depicted in Fig. 5.9b commutes and that $\hat{m}'$ is injective.

To show: $\forall e\colon n \to b \in L'_{EB} \setminus L_{EB}\colon \hat{m}'(b) \circ \hat{L}'(e) = \hat{G}'(m'(e)) \circ \hat{m}'(n)$.
$\forall x \in \hat{L}'(n)\colon \hat{m}'(b) \circ \hat{L}'(e)(x) \overset{i)}{=} \hat{m}'(b)([x]) \overset{h)}{=} \hat{G}'(m'(e)) \circ \hat{m}'(n)(x)$.

For injectivity to show: $\hat{m}'(b)\colon \hat{L}'(b) \to \hat{G}'(b)$ is injective $\forall b \in \overline{B}$, i.e., $\forall x \in \hat{L}'(n)$ with $e\colon n \to b \in L'_{EB} \setminus L_{EB}$ and $\forall y \in \hat{L}'(n')$ with $e'\colon n' \to b \in L'_{EB} \setminus L_{EB}$ the following holds: $[x] \neq [y] \in L'(b) \Rightarrow \hat{m}'(b)([x]) \neq \hat{m}'(b)([y])$.
$\hat{m}'(b)([x]) \overset{h)}{=} \hat{G}'(m'(e)) \circ \hat{m}'(n)(x) \overset{g)}{\neq} \hat{G}'(m'(e')) \circ \hat{m}'(n')(y) \overset{h)}{=} \hat{m}'(b)([y])$ with $x \in [x]$ and $y \in [y]$ since this is exactly how the equivalence relation is defined.

It can be followed that $\hat{m}'\colon \hat{L}' \to \hat{G}'$ is an injective composite graph morphism.

The constructions of $\hat{L}'$ and $\hat{m}'$ heavily relies on the structure of $\hat{G}'$. They can obviously always be performed since missing structures are solely added. The uniqueness of $\hat{L}'$ follows from the uniqueness of $\hat{G}'$, since body nodes and export-body edges in L' are directly inferred by G' and body graphs in $\hat{L}'$ are constructed such that they uniquely match the body graphs in $\hat{G}'$

Step 3: $\hat{K}'$ and $\hat{l}'$, $\hat{R}'$ and $\hat{r}'$ are well-defined. The construction in this step is rather simple. Consider Fig. 5.10 in addition. Since uncoupled exports are required to be fully preserved, i.e., no changes happen on network and object layer (cf. Def. 5.25), they are identical in $\hat{L}$, $\hat{K}$, and $\hat{R}$. Thus, $\hat{K}'$ is obviously a composite graph as it comprises $\hat{K}$ and all new bodies and export-body edges of $\hat{L}'$ (see items a),b),e), and f)).

It remains to show that the diagram in Fig. 5.10 commutes and thus $\hat{l}'$ is well-defined, i.e., $\forall x \in \hat{L}'(n)$ with $e\colon n \to b \in \overline{E}\colon \hat{L}'(e) \circ \hat{l}'(n)(x) = \hat{l}'(b) \circ \hat{K}'(e)(x)$.
$\hat{L}'(e) \circ \hat{l}'(n)(x) \overset{3e)}{=} \hat{L}'(e) \circ \hat{l}(n)(x) \overset{\hat{l}(n) \text{ is embedding}}{=} \hat{L}'(e)(x) = id_{\hat{L}'(b)} \circ \hat{L}'(e)(x) \overset{3g),3f)}{=} \hat{l}'(b) \circ \hat{K}'(e)(x)$. All other commutativities are given by $\hat{l}$. Since $\hat{l}'$, roughly spoken, comprises $\hat{l}$ (being an embedding) and the identity morphisms of the new bodies and the new export-body edges, $\hat{l}'$ is an embedding as well.

Analogously $\forall x \in \hat{L}'(n)$ with $e\colon n \to b \in \overline{E}\colon \hat{R}'(e) \circ \hat{r}'(n)(x) = \hat{r}'(b) \circ \hat{K}'(e)(x)$.

Since all elements of $\hat{p}'$ are well-defined, i.e., they particularly are composite graphs and embedding composite graph morphisms, $\hat{p}'$ is a composite graph rule.

The existence and uniqueness of $\hat{K}'$ and $\hat{R}'$ follows from the existence and uniqueness of $\hat{L}'$ since $\hat{K}'$ and $\hat{R}'$ are only extended by "missing" parts to be found in $\hat{L}'$. The extensions of the embeddings $\hat{l}'$ and $\hat{r}'$ are trivial.

□

Example 5.30. Figure 5.11 illustrates the weak conversion partially, i.e., it is limited to the construction of the host graph and the left-hand side of the rule. Accordingly, only Step 1 and Step 2 are shown being divided into four stages, (a) - (d). Each stage is represented by a box which shows the LHS composite graph in the upper area and the host composite graph below. Both are shown by means of their network graph (left) and their refining local graphs. Relationships between network levels are denoted by their location. Match mappings are indicated by vertical dashed arrows.

Stage (a) shows as initial state a left-hand side comprising three uncoupled export nodes. The object level exhibits that each of them contains a single object. The host graph consists of two body nodes at which one is equipped with two export interfaces. Furthermore, a single uncoupled export node is given. All elements on network layer and object layer in the LHS are already appropriately matched into the host composite graph.

The conversion in Step 1 is performed on the host composite graph only and yields stage (b). On network layer, the single uncoupled export node is now connected to a new body node. On object layer, the new body related graph consists of a single object similar to its export graph. In fact, according to the construction the body graph actually consists of the same elements as its export graph. This is difficult to visualize though and does not matter for the illustration.

Steps 2.a) - e) complete the LHS and create match related mappings on network layer yet. Figure 5.11(c) reflects this: Each export network node in the LHS is now connected to a body node. This is achieved by selecting all body nodes in the host graph whose export nodes are matched by uncoupled exports. These selected bodies are then used in the LHS. Match mappings are added appropriately.

Finishing Step 2 leads to the composite graphs depicted at stage (d). Changes essentially concern the completion of the LHS on object layer. The empty graphs refining the new bodies are filled up with related objects. The two nodes enclosed by the dashed rectangle indicate a grouping into an equivalence class; the equivalence relation defined in conversion Step 2.g) glues both nodes together as they point to a single node in the host graph. △

Proposition 5.31 (Applicability of weak composite graph transformation steps)**.** Given a weak composite rule $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R})$, a weak composite graph $\hat{G}$ as host and a composite match $\hat{m}\colon \hat{L} \to \hat{G}$ which satisfies the gluing conditions for weak composite graphs, then a weak composite graph transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ exists and is unique.

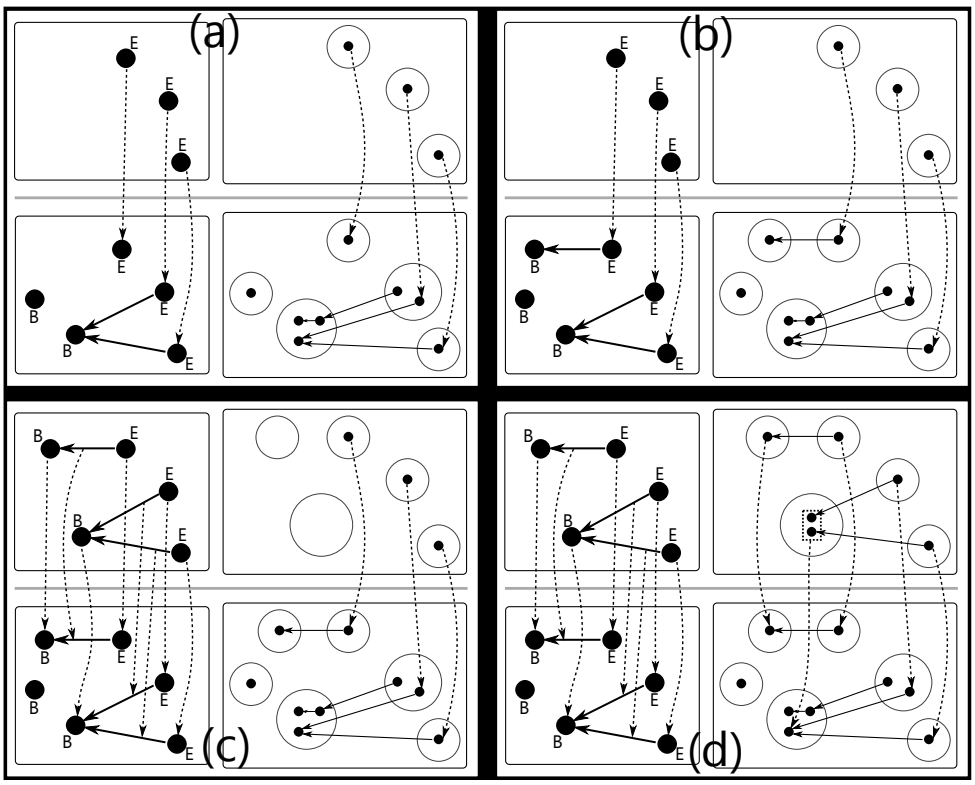


Figure 5.11: Example completion of the left-hand side and the host composite graph in the context of a weak composite transformation conversion (see Def. 5.28). The conversion is shown in four steps (a) to (d) at which each time the LHS is arranged above the host and both are shown on network (left) and object layer (right). The matching from LHS to host is denoted by dashed edges.

Proof. A weak composite graph transformation step is performed by converting $\hat{p}$, $\hat{G}$, and $\hat{m}$ to a composite rule $\hat{p}'$, a composite graph $\hat{G}'$, and a composite morphism $\hat{m}'$ as defined in Def. 5.28 and applying them in the context of composite graph transformation step.

$\hat{m}'$ is a composite match as it subsumes $\hat{m}$ which already obeys the gluing condition for composite graphs and additional total and injective morphisms between bodies in $\hat{L}'$ and $\hat{G}'$ being all preserved. Thus, the gluing condition is also satisfied for all $b \in \overline{B}$ (with $\overline{B}$ being defined as in Def. 5.28).

Then, a transformation step in CompGraphs is performed yielding the composite graphs $\hat{D}'$ and $\hat{H}'$.

From the construction of $\hat{D}'$ (cf. Def. 5.21) it follows that $\forall b \in G'_{Bod} \backslash G_{Bod}$, $b \in D'_{Bod}$ since each b is preserved. This holds analogously for all $e \in G'_{EB} \setminus$

G_{EB} since each e and their export source nodes are preserved. Consequently, each b and e are also part of H'_{Bod} and H'_{EB}, respectively. Thus, the weak composite graphs $\hat{D}$ and $\hat{H}$ can be deduced by removing those additional bodies and export-body edges as follows: Let $\overline{B} = G'_{Bod} \setminus G_{Bod}$ and $\overline{E} = G'_{EB} \setminus G_{EB}$. Then, $D_N = D'_N \setminus \overline{B}$ and $D_E = D'_E \setminus \overline{E}$; analogously for $\hat{H}$.

The resulting graphs $\hat{D}$ and $\hat{H}$ are weak composite graphs being defined in a straightforward way by removing all graphs of nodes in $\overline{B}$ and all morphisms of edges in $\overline{E}$.

The existence and uniqueness of the transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ follows from the existence and uniqueness of $\hat{G}' \overset{\hat{p}',\hat{m}'}{\Longrightarrow} \hat{H}'$. The existence and uniqueness of $\hat{G}' \overset{\hat{p}',\hat{m}'}{\Longrightarrow} \hat{H}'$ follows from the conversion construction from weak composite graphs and morphisms to strong ones (cf. Def. 5.28) and back again. Above, the construction of $\hat{G}'$, $\hat{L}'$, $\hat{K}'$, $\hat{R}'$ and all remaining morphisms have been argued to always exist and to be unique. The resulting graph $\hat{H}'$ can uniquely be reduced as follows $\hat{H} = \hat{H}' \setminus (\hat{G}' \setminus \hat{G})$ in a componentwise manner. This is always possible since the content of $\hat{G}' \setminus \hat{G}$ only comprises bodies and export-body edges which are particularly fully preserved on network and object layer due to the restrictions of weak composite rules in Def. 5.25. □

Chapter 6

Transformation of Composite Graphs with Inheritance and Containment

In the present chapter, the concepts of composite graph transformation (see Chap. 5) and the transformation of graphs with inheritance and containment (see Chap. 4) are combined. It is structured as follows: Section 6.1 introduces typed composite graphs with inheritance and containment structures to provide a formal basis for composite EMF models. They are based on the definitions of composite graphs where the graphs on object level are equipped with inheritance and containment structures. This has been done in [46] for the first time and is recalled and slightly improved here. Related transformations are presented in Sec. 6.2 and it is shown under which preconditions pushouts can be constructed and consequently when transformations can be performed. This chapter finishes with an elaboration on the transformation of weak typed composite graphs in Sec. 6.3.

6.1 Typed Composite Graphs with Inheritance and Containment

Now that composite graphs and (typed) graphs with inheritance and containment structures are defined, both are combined yielding composite IC-graphs. At first, (typed) composite IC-graphs and IC-morphisms are defined and it is shown that together they form a category, called CompICGraphs$_{TG}$. Furthermore, this section investigates under which conditions pushouts exist in that category. This allows in the subsequent section to introduce the transformation of composite IC-graphs based on the DPO approach.

Definition 6.1 (Composite IC-graph and composite IC-morphism)**.** Given a composite network graph G, $\hat{G} = (G, \mathcal{G}(G), \mathcal{M}(G))$ is a *composite IC-graph* if $\mathcal{G}(G)$ is a set of IC-graphs and $\mathcal{M}(G)$ is a set of IC-morphisms refining network nodes and edges of G as in Definition 5.9. Moreover, the commutativity condition of Definition 5.9 holds.

Given two composite IC-graphs $\hat{G}$ and $\hat{H}$, a pair $\hat{f} = (f, m) = \hat{f} \colon \hat{G} \to \hat{H}$ is a *composite IC-morphism* if f is a composite network morphism and m is a family of IC-morphisms as defined in Definition 5.10.

◇

Remark 6.2. A *straight extension with inheritance and containments* of a (simple) composite graph leading to a composite IC-graph is defined componentwise, i.e., each simple graph refining a network node is straightly extended as defined in Def. 4.1. Related (simple) graph morphisms can be easily lifted to equivalent IC-morphisms. ▽

Example 6.3. According to the network graph in Fig. 5.4, a composite IC-graph is shown in Fig. 6.1. Each network node is illustrated by a rounded rectangle which is equipped with meaningful names in the upper left corner and a refining IC-graph in the center. Network edges are omitted in favor of readability while their refining mappings are shown in form of patterned arrows. Dashed arrows illustrate mappings between interfaces and their bodies while dotted arrows are mappings between interfaces. The department management component (Dep) is shown on the left while the project management component (Prj) is shown on the right. The body graphs *DepBody* and *PrjBody* are depicted in the upper area of Fig. 6.1 and their interfaces are arranged below.

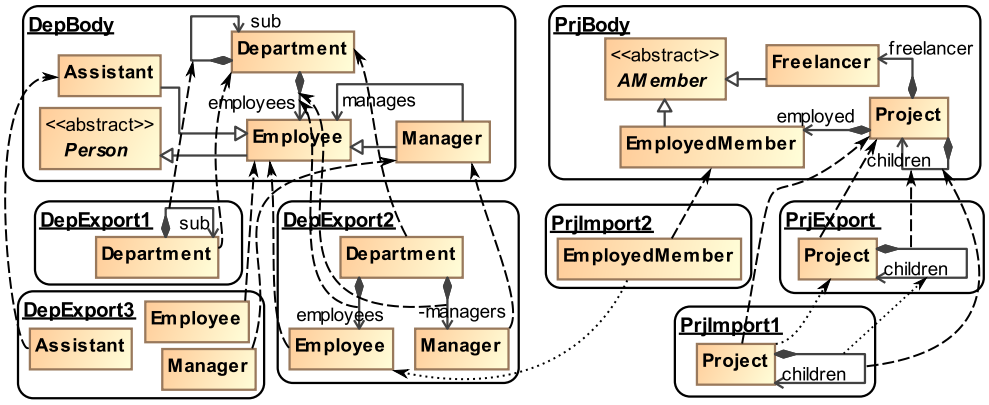


Figure 6.1: Composite IC-graph over the network graph in Fig. 5.4.

This scenario describes how two software components may be connected. On the one hand, there is a department software component providing inter-

faces to gather information about published department structures, employed persons and assignments of persons to certain departments. On the other hand, a project software component may reveal its project structure and may require other projects to be associated at the same time. Furthermore, it may depend on (exported) employees of a department to work within a project.

Interfaces do not need to be mapped into their corresponding body graph in a one-to-one manner, e.g., export graph *DepExport2* provides a structure which is different to the corresponding part in its body. This export graph is valid, though, since the containment edge `managers` can be easily mapped to the body's containment edge `employees`. This technique allows to hide structural dependencies and complexity by providing simple and convenient interfaces at the same time.

Both components are connected by the import *PrjImport2* and the export *DepExport2*. Moreover, the project management component is connected with itself by *PrjExport* and *PrjImport1*. The impact of both connections is examined in detail in Example 6.8.

With regard to EMF models, it would be meaningful to require each body graph to be *rooted*. However, this requirement is not made explicit in the present approach. Interface graphs do not have to be rooted anyway as they provide representatives of elements from their corresponding rooted body graphs, e.g., the export *DepExport3* does not provide a root. △

In the following definition, typed composite IC-graphs are defined by special composite IC-morphisms expressing typing.

Definition 6.4 (Typed composite IC-graph and typing composite IC-morphism)**.** A composite graph $\hat{G}(G)$, called *typed composite IC-graph*, with composite network graph G is typed over a composite IC-graph $\hat{TG}(TG)$ with composite network graph TG if there is a composite IC-morphism $\hat{type}_{\hat{G}}\colon \hat{G} \to \hat{TG}$, called *typing composite IC-morphism*, which is a pair $\hat{type}_{\hat{G}} = (type_G, m)$ where:

- $type_G\colon G \to TG$ is a composite network graph morphism, and
- m is a family of typing IC-morphisms $\{\hat{type}_{\hat{G}}(n) \mid n \in G_N\}$ (cf. Def. 4.7).

◇

Remark 6.5 (Global acyclic containment)**.** Note that each component part of a typed composite IC-graph is constituted by local IC-graphs which have acyclic containments by definition. However, corresponding nodes in different local graphs can be in opposite containment relations which would lead to cyclic containment if all local graphs would be merged into one big graph. This is easy to see: Consider a case where a node contains another one and which are both exported without their containment relation. The importing component may locally add a containment relation in the opposite direction.

Nevertheless, the property of global acyclic containments can be achieve in two ways:

1. Consider a given typed composite IC-graph as diagram in the category ICGraphs$_{TG}$, whose colimes construction is required to yield a typed IC-graph without cyclic containments.

2. Import and export interfaces are restricted by always exporting/importing all containment edges between exported/imported nodes. The corresponding proof is straightforward.

▽

Definition 6.6 (Typed Composite IC-Morphism)**.** Let $\hat{G}$ and $\hat{H}$ be two composite IC-graphs typed over a composite IC-graph $\hat{TG}$ by $\hat{type}_{\hat{G}}$ and $\hat{type}_{\hat{H}}$, respectively. A *typed composite IC-morphism* is a composite morphism $\hat{f}\colon \hat{G} \to \hat{H}$ such that $\hat{type}_{\hat{H}} \circ \hat{f} = \hat{type}_{\hat{G}}$. ◇

Remark 6.7 (Categories and Sub-categories)**.** Note that the category of typed composite IC-graphs and morphisms, CompICGraphs$_{TG}$, is defined as a slice category of CompICGraphs, the category of composite IC-graphs and composite IC-morphisms. Composite I-graphs and composite IC-graph morphisms form a full sub-category of CompICGraphs, called CompIGraphs.

Let TG be a composite IC-graph and TG' its composite I-graph, i.e., $\forall n \in TG' = TG$, $\hat{TG}'(n)$ is the I-graph of $\hat{TG}(n)$. Then, composite I-graphs and composite IC-graph morphisms typed over TG' form a full sub-category of CompICGraphs$_{TG}$, called CompIGraphs$_{TG}$.

Similarly, the category CompGraphs$_{TG}$ is the full sub-category of CompIGraphs$_{TG}$ where all composite graphs are typed over the composite graph $\hat{TG}'$ with $\forall n \in TG' = TG$, $\hat{TG}'(n) = T(\hat{TG}(n))$. ▽

Example 6.8. Considering the composite type IC-graph in Fig. 6.1, Fig. 6.2 shows an example composite IC-graph typed over that composite IC-graph. Each graph is equipped with a meaningful name according to its typing body or interface graph. Again, network edges are not explicitly shown while their refining mappings denote them. On the left, a department management component instance is given with one body and two export interfaces. These interfaces are typed over two different export type graphs. On the right, two project management component instances are given. In order to distinguish both, the suffixes ".1" and ".2" are added to related component parts. Mappings between typed edges are neglected as in previous examples.

The export interface *DepExport2* exports managers, employees, and departments. It illustrates that not all body objects have to be exported since some of them are missing in the export. Moreover, although `Martin` is contained in a department (see *DepBody*), this information does not have to be

exported either. In export *DepExport1*, a subset of the department structure is exposed.

On the right-hand side, the import interface *PrjImport2.1* is depicted with a morphism running to the export interface *DepExport2*. This morphism maps an `EmployedMember` instance to an `Employee` instance. Note that both instances being mapped are typed over different types (compare Fig. 6.1) but are meant to semantically correspond to each other. However, this mapping is permitted due to the corresponding mapping between `EmployedMember` and `Employee` on type level. Furthermore, consider the import and export graphs *PrjImport1.1* and *PrjExport.2*. They illustrate how component instances which are typed over the same component type can relate to each other. Please note the related mapping on type level in the bottom right of Fig. 6.1 between interfaces of the same component. △

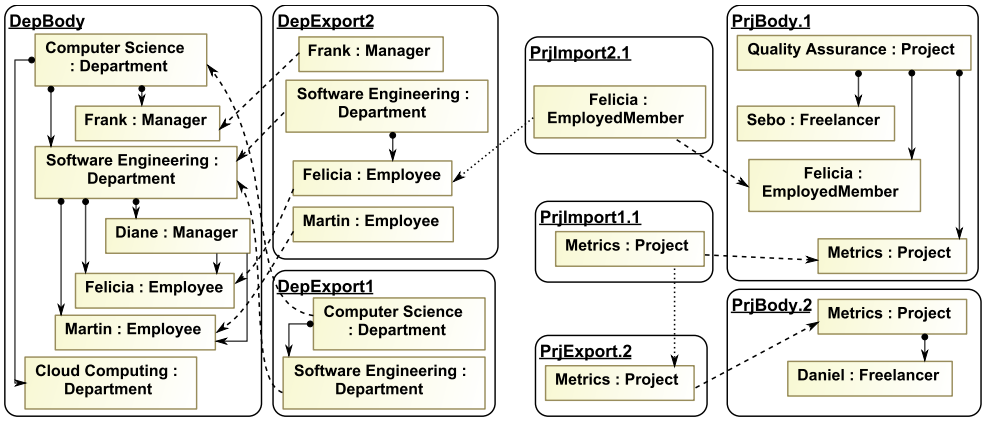


Figure 6.2: Example of a typed composite IC-graph.

Below, pushouts in the category CompICGraphs$_{TG}$ are examined and double pushouts thereafter in order to prepare the definition of transformations in CompICGraphs$_{TG}$. As already shown, containments are problematic here, since there may be cases where, e.g., a pushout includes an object with two containers which is forbidden by definition. Therefore, for now composite IC-graphs with empty containments are consider only by means of the category CompIGraphs$_{TG}$. The problems related to containments are tackled later by the definition of appropriate *consistent* rules which never cause containment problems.

In the remainder of this section, it is shown that pushouts in the category CompIGraphs$_{TG}$ exist. The argumentation for pushouts relies on the fact that IGraphs$_{TG}$ is co-complete.

Proposition 6.9 ((CompIGraphs$_{TG}, \mathcal{M}_{inj}$) is a weak adhesive HLR category)**.** The category CompIGraphs$_{TG}$ with the class $\mathcal{M}_{inj}$ of injective typed

composite I-graph morphisms forms a weak adhesive HLR category. Pushouts along injective composite network morphisms can be constructed component-wise.

PROOF. The first statement follows from the reasoning in Prop. 5.14 and Prop. 4.12 and can be argued analogously.

Additionally, the component-wise construction follows from Corollary 5.12. □

PROPOSITION 6.10 (DPO in COMPIGRAPHS$_{TG}$)**.** Let $(PO1)$ and $(PO2)$ be two pushouts in the category COMPIGRAPHS$_{TG}$ as illustrated in Fig. 6.3 with $\hat{l}$, $\hat{r}$, and $\hat{m}$ being injective typed composite I-graph morphisms and with the typing morphisms $\hat{type}_{\hat{L}}$, $\hat{type}_{\hat{K}}$, $\hat{type}_{\hat{R}}$, and $\hat{type}_{\hat{G}}$. The typing morphisms $\hat{type}_{\hat{D}}$ and $\hat{type}_{\hat{H}}$ exist and (7) and (8) commute.

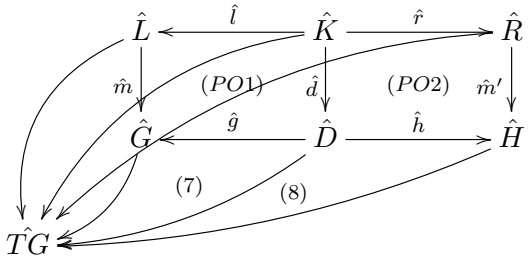


Figure 6.3: Double pushout in COMPIGRAPHS$_{TG}$.

PROOF. Let $(PO1)$ and $(PO2)$ be double pushouts in COMPGRAPHS$_{TG}$ and let $\hat{type}_X$ be typing morphisms with $X \in \{\hat{L}, \hat{K}, \hat{R}, \hat{G}, \hat{D}, \hat{H}\}$ from a composite graphs X to the composite I-graph $T\hat{G}$ such that $\hat{type}_{\hat{L}} \circ \hat{l} = \hat{type}_{\hat{K}} = \hat{type}_{\hat{R}} \circ \hat{r}$ (1) and $\hat{type}_{\hat{G}} \circ \hat{m} = \hat{type}_{\hat{L}}$ (2). The typing morphisms to be defined are $\hat{type}_{\hat{D}} \colon \hat{D} \to T\hat{G}$ with $\hat{type}_{\hat{G}} \circ \hat{g} = \hat{type}_{\hat{D}}$ and $\hat{type}_{\hat{H}} \colon \hat{H} \to T\hat{G}$ with $\hat{type}_{\hat{H}} \circ \hat{h} = \hat{type}_{\hat{D}}$ and $\hat{type}_{\hat{H}} \circ \hat{m}' = type_{\hat{R}}$.

Let $\hat{type}_{\hat{D}} = \hat{type}_{\hat{G}} \circ \hat{g}$ (3). This implies $\hat{type}_{\hat{D}} \circ \hat{d} = \hat{type}_{\hat{R}} \circ \hat{r}$ (4) using equations (1), (2), (3) and pushout $(PO1)$. Now, Prop. 6.9 can be exploited considering the pushout $(PO2)$ and the equation (4). Then, there is a unique composite I-graph morphism $\hat{type}_{\hat{H}} \colon \hat{H} \to T\hat{G}$ with $\hat{type}_{\hat{H}} \circ \hat{h} = \hat{type}_{\hat{D}}$ and $\hat{type}_{\hat{H}} \circ \hat{m}' = \hat{type}_{\hat{R}}$. □

6.2 Consistent Transformation of Typed Composite IC-Graphs

Analogously to non-composite IC-graph transformation, this section introduces the transformation of composite IC-graphs and incorporates the notion of consistency in order to tackle the containment related issues in pushouts.

Definition 6.11 (Composite IC-graph transformation)**.** A composite transformation system $TS = (\hat{S}, \hat{P}, \hat{TG})$ over the category CompICGraphs$_{TG}$ consists of the composite IC-graphs $\hat{TG}$ and $\hat{S}$, called *composite type graph* and *composite start graph*, with $\hat{S}$ being typed over $\hat{TG}$, and a set of *consistent* composite IC-graph rules $\hat{P}$.

- A *composite IC-graph rule* $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R}, \hat{type})$ consists of composite IC-graphs $\hat{L}$, $\hat{K}$, and $\hat{R}$ typed over $\hat{TG}$ by the triple $\hat{type} = (\hat{type}_{\hat{L}}\colon \hat{L} \to \hat{TG}, \hat{type}_{\hat{K}}\colon \hat{K} \to \hat{TG}, \hat{type}_{\hat{R}}\colon \hat{R} \to \hat{TG})$ being composite IC-morphisms and typed composite IC-morphisms $\hat{l}\colon \hat{K} \hookrightarrow \hat{L}$ and $\hat{r}\colon \hat{K} \hookrightarrow \hat{R}$ being *inclusions* such that $\forall n \in K_N\colon \hat{p}(n) = (\hat{L}(n) \overset{\hat{l}(n)}{\hookleftarrow} \hat{K}(n) \overset{\hat{r}(n)}{\hookrightarrow} \hat{R}(n), \hat{type}(n))$ is an IC-graph rule (cp. Definition 4.14). Moreover, $\forall n \in R_N - K_N\colon \hat{type}_{\hat{R}}(n)$ is concrete.

  A composite IC-rule $\hat{p}$ is called *consistent*, if $\forall n \in K_N\colon \hat{p}(n)$ is consistent (cf. Def. 4.18).

- A *composite IC-graph transformation step* (or direct composite IC-graph transformation) $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ of a typed composite IC-graph $\hat{G}$ to $\hat{H}$ by a consistent composite IC-graph rule $\hat{p}$ and a typed injective composite IC-morphism $\hat{m}\colon \hat{L} \to \hat{G}$ is given in Fig. 6.4 below, where (1) and (2) are pushouts in the category CompICGraphs$_{TG}$.

- A *composite IC-graph transformation* is a sequence $\hat{G}_0 \Rightarrow \hat{G}_1 \Rightarrow ... \Rightarrow \hat{G}_n$ of direct composite IC-graph transformations, written $\hat{G}_0 \overset{*}{\Rightarrow} \hat{G}_n$.

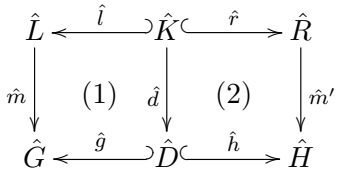


Figure 6.4: Illustration of a composite (IC-)graph transformation step by means of two pushouts.

◇

As for the transformation of local typed IC-graphs, in Chap. 4 arising difficulties regarding the preservation of proper containment relations have been discussed and how to face them by consistent IC-rules. Composite IC-graph transformations are essentially constituted by a number of local IC-graph transformations such that an obvious idea might be to exclusively use consistent IC-rules on object layer (cf. Example 6.12). And indeed, Theorem 6.13 shows that such a consistent transformation yields a composite IC-graph again.

Example 6.12. Figure 6.5 shows a composite IC-graph rule which conforms with the composite type IC-graph in Fig. 6.1. The presentation is similar to previous figures, i.e., network nodes (or component parts) are illustrated by rounded rectangles equipped with meaningful names according to their typing. Refining graphs are located within the rectangles where object identities are denoted by equal numbers while mappings between typed edges are neglected. Network edges are not explicitly shown but are constituted by their refining mappings illustrated by dashed edges.

The LHS contains a department management component with a body and an export interfaces. The body shows three departments in a hierarchical containment relation where two of them are exported together with their relation. As indicated by the RHS, an application of this rule shall not modify the network structure, i.e., body and export are preserved whereas changes on object level do occur. In detail, the hierarchical containment structure of the three departments changes which also involves the deletion of the containment relation in the export interface. Furthermore, an employee appears in the body.

Each two corresponding component parts in the LHS and RHS, slightly highlighted by the rectangles with dotted borderlines, can be considered as the LHS and RHS of a local rule. For instance, the local rule of the component part `DepBody` resembles the sample *consistent IC-rule* shown in Fig. 4.4 which especially illustrates how cycle capable containment has to be treated. In addition, the new containment edge in the RHS has been added together with a new employee object. This satisfies another requirement for consistent rules (cf. Sec. 4.2), i.e., a containment edge has been created together with a contained object node. The deletion of containment edges as shown in the local rule of the interface `DepExport2` does not conflict with any consistent IC-rule constraint. To sum up, since both local rules are consistent IC-rules, the composite IC-rule is considered consistent as well and so is the corresponding transformation with arbitrary host composite IC-graphs. That this holds true is shown immediately in the proof of Theorem 6.13 below. △

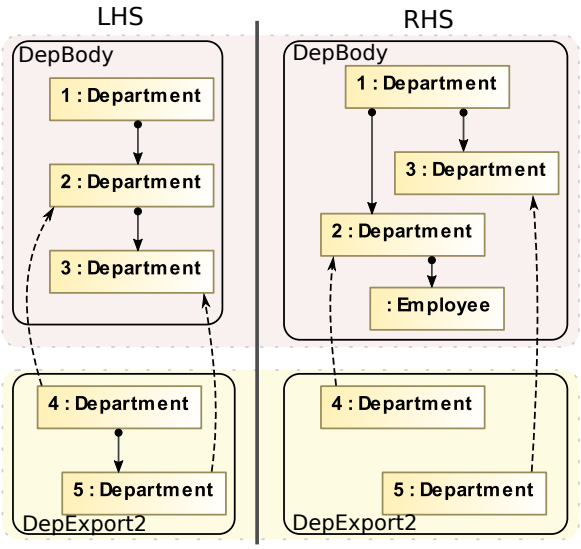


Figure 6.5: Sample consistent composite IC-graph rule.

Theorem 6.13 (Consistent composite IC-graph transformation step)**.** A composite IC-graph transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ as defined in 6.11 exists and the resulting graph $\hat{H}$ is a composite IC-graph typed over $\hat{TG}$.

Proof. First, each given IC-morphism is restricted to an I-morphism with empty containments and the DPO in category CompIGraphs$_{TG}$ is constructed. Pushouts exist since CompIGraphs$_{TG}$ is shown to be co-complete (see Proposition 6.9). Therefore, a composite I-graph $\hat{H}$ results which is typed over $\hat{TG}'$ being $\hat{TG}$ with empty containment (cf. Prop. 6.10). It is still to be shown that $\hat{H}$ can be typed over $\hat{TG}$:

1. Each local graph $\hat{H}(n)$ for $n \in H_N$ is an IC-graph concretely typed over $\hat{TG}(type_H(n))$.
2. Each local graph morphism $\hat{H}(e)$ for $e \in H_E$ is an IC-morphism typed over $\hat{TG}(type_H(e))$.

For the proof of item 1, consider Fig. 6.4 showing a composite graph transformation step as double pushout. For proof of item 2, consider Fig. 6.6 in particular.

Proof of Item 1. Pushouts in CompIGraphs$_{TG}$ are constructed componentwise. According to that, $\forall x \in H_N$: $\hat{H}(x)$ is either constructed by a pushout in the category IGraphs$_{TG}$ or is equal to $\hat{D}(x)$ or $\hat{R}(y)$ with $m'(y) = x$. In case of $\hat{H}(x) = \hat{R}(y)$, $\hat{H}(x)$ is an IC-graph, since $\hat{R}(y)$ is one. Moreover, $\hat{H}(x)$ is concretely typed since $\hat{R}(y)$ is so, due to assumptions. In case of

$\hat{H}(x) = \hat{D}(x)$, it follows that $\hat{D}(x) = \hat{G}(x)$. Since $\hat{G}(x)$ is an IC-graph, so is $\hat{H}(x)$. If $\hat{H}(x)$ is constructed by a pushout, there is a local graph transformation $\hat{G}(x) \overset{\hat{p}(y),\hat{m}(y)}{\Longrightarrow} \hat{H}(x)$ with $m(y) = x$. Since $\hat{p}(y)$ is consistent by assumption, $\hat{H}(x)$ is an IC-graph concretely typed according to Fact 4.20.

Proof of Item 2. The following is known:

(i) From the pushout construction in CompIGraphs$_{TG}$, it follows that $\hat{TG}(type_H(e)) \circ \hat{type}(i) = \hat{type}(j) \circ \hat{H}(e)$ for I-graphs (cf. Prop. 6.9).

(ii) For all $e \in H_E$: $\hat{TG}(type_H(e))(C(\hat{TG}(type_H(s(e))))) \subseteq C(\hat{TG}(type_H(t(e))))$, since $\hat{TG}$ is a composite IC-graph.

(iii) $\hat{type}(i)_E(C(\hat{H}(i)) \subseteq C(\hat{TG}(type_H(i)))$, since $\hat{type}(i)$ is an IC-morphism (Fact 4.20).

(iv) $\hat{type}(j)_E(C(\hat{H}(j)) \subseteq C(\hat{TG}(type_H(j)))$, since $\hat{type}(j)$ is an IC-morphism (Fact 4.20).

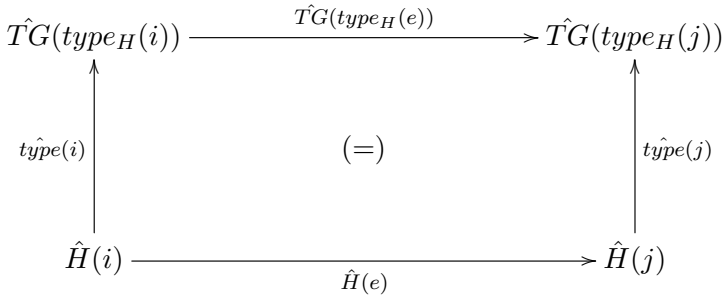


Figure 6.6: A typed local graph morphism.

To show: $\hat{H}(e)(C(\hat{H}(i))) \subseteq C(\hat{H}(j))$, $\forall e \in H_E$.
$A = \hat{TG}(type_H(e))(\hat{type}(i)(C(\hat{H}(i)))) \subseteq \hat{TG}(type_H(e))(C(\hat{type}(i))) \subseteq C(\hat{TG}(type_H(j)))$, since $\hat{type}(i)$ and $\hat{TG}(type_H(e))$ are IC-morphisms.
$B = \hat{type}(j)_E(C(\hat{H}(j))) \subseteq C(\hat{TG}(type_H(j)))$, since $\hat{type}(j)$ is an IC-morphism.
$A = \hat{type}(j)_E(\hat{H}(e)(C(\hat{H}(i))))$, due to equation (i).
$A \subseteq B$, since $\hat{type}(j)$ is a typing IC-morphism.
Hence: $\hat{H}(e)(C(\hat{H}(i))) \subseteq C(\hat{H}(j))$, $\forall e \in H_E$. □

Remark 6.14. In order to ensure local consistency in terms of acyclic containments, consistent composite IC-graph transformation steps apply consistent local IC-rules only. This approach is not sufficient to ensure global acyclic containments though. As in Remark 6.5, consider again the case where a node x *contains* a node y and both nodes are exported without their containment

relation. Furthermore, an importing component may already refer to the contained node y. A consistent composite IC-graph transformation step may locally add node x and a containment relation running from y to x, which is, however, opposite to the original containment relation.

Following the approach proposed in item 2 of Remark 6.5, consistent composite IC-graph transformations where all containments between two exported and imported objects have to be declared would always maintain this consistency. The proof is straightforward and omitted here. ▽

6.3 Transformation of Weak Typed Composite IC-Graphs

The liberation of composite graphs to weak composite graphs, where exports may appear without their bodies, as well as the transformation of weak composite graphs are introduced in Sec. 5.4. Such transformations are led back to the transformation of ordinary composite graphs by extending rule graphs and host graphs by surrogate bodies if needed. The crucial part is the artificial introduction of new bodies which are surrogates for the ones "currently" not available and which must particularly be suitable. For this reason, the body graphs are constructed based on the structure of their relating (previously uncoupled) export graphs, on the one hand, and the already existing matching from these export graphs into the host graph, on the other hand.

The concept of *weak* composite graph transformation can be applied to typed composite graphs with inheritance and containment structures since typed composite IC-graphs are actually composite graphs with typing IC-morphisms to distinguished composite IC-graphs.

Example 6.15. Figure 6.7 shows a composite IC-graph rule being obviously *weak*. The LHS contains an incomplete composite graph where the department management export interface of type `DepExport2` (see type composite IC-graph in Fig. 6.1) is shown but its belonging body is missing. As defined in Sec. 5.4, such exports must be fully preserved which is the case here. The application of this rule shall create a new project management component consisting of a body and an import interface where the interface is connected to the export of the department management. Nevertheless, the operation does not depend on the information of the department management's body .

Now consider the case, where the host composite graph of the transformation is equal to the LHS, i.e., the body component part is missing. Then, the export interfaces of LHS, RHS and host composite graph are each completed by a body which is isomorphic to the exports. The corresponding mappings and matching are obviously trivial. If the body is already given in the host composite graph, the LHS (and RHS) are completed with a body which carries

only a minimal graph, i.e., a graph which carries only those elements needs to be mapped by the exported elements.

Both scenarios can be found in the schema in Fig. 5.11. △

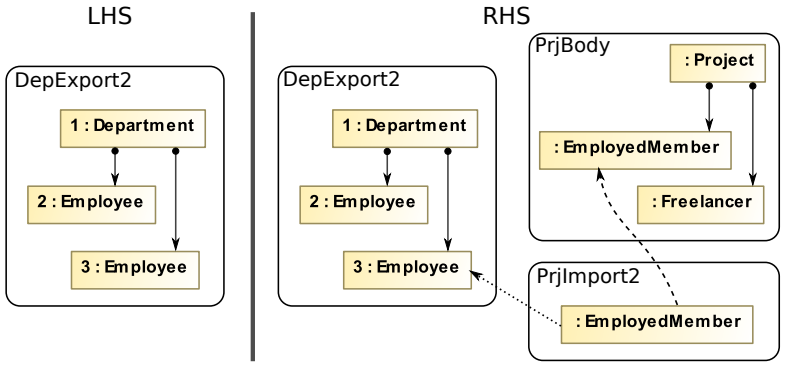


Figure 6.7: Sample weak composite IC-graph rule.

In the following, weak composite IC-graph rules and the corresponding gluing condition are not re-considered as they are similar to those in Sec. 5.4. The interesting part is rather the adaptation of the body construction in terms of a *weak composite IC-graph transformation step conversion* analogously to Def. 5.28. It also remains to show that such a construction always yields well-typed composite IC-graphs. Additionally, the properties of typed IC-graphs, *at most one container* and *no containment cycles*, must be satisfied.

The body-construction on network level is always straightforward: Since each (type) export interface uniquely belongs to one (type) body, the typing of a new body node can be easily deduced by its export node. The body-construction on object level is almost that simple. For the host composite IC-graph, each new body graph structure exactly corresponds to one export graph structure and since each element in a (type) export graph points to exactly one element in the (type) body graph, the typing of an element in a typed body graph can be deduced. The construction of new body graphs in the composite rule appears more subtle at the first glance due to the equivalence relation which potentially maps elements from different export graphs to one element in the body graph. Nevertheless, the equivalence relation refers to relations between export graphs and their body graph in the host graph which are well-typed from which is follows that the new body graphs in the rule are well-defined, too.

In the following, the composite graph transformation step conversion is shown with regard to the typing over composite graphs with inheritance and containment structures. Then, the well-definedness of the body construction and the satisfaction of typed IC-graph properties are shown in a formal manner and are illustrated by an example afterwards.

Definition 6.16 (Weak typed composite IC-graph transformation step conversion)**.** Let there be a weak typed composite IC-rule $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R}, \hat{type})$ with $\hat{L}$, $\hat{K}$, and $\hat{R}$ being weak composite IC-graphs typed over the non-weak composite IC-graph $\hat{TG}$ by the triple $\hat{type} = (\hat{type}_{\hat{L}}\colon \hat{L} \to \hat{TG}, \hat{type}_{\hat{K}}\colon \hat{K} \to \hat{TG}, \hat{type}_{\hat{R}}\colon \hat{R} \to \hat{TG})$ being composite IC-morphisms.

Given a weak typed IC-composite transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ with the weak typed IC-composite graph $\hat{G}$ as host typed via $\hat{type}_{\hat{G}}\colon \hat{G} \to \hat{TG}$, and the composite match $\hat{m}\colon \hat{L} \to \hat{G}$ which satisfies the gluing conditions for weak composite graphs, then the weak typed IC-composite transformation step is converted to a typed IC-composite transformation step $\hat{G}' \overset{\hat{p}',\hat{m}'}{\Longrightarrow} \hat{H}'$ as follows:

At first, the conversion according to Def. 5.28 is applied whose result, $\hat{G}' \overset{\hat{p}',\hat{m}'}{\Longrightarrow} \hat{H}'$, serves as starting point for the conversion steps below.

For convenience the following is defined: $G'_{\overline{B}} = G'_{Bod} \setminus G_{Bod}$ and $G'_{\overline{E}} = G'_{EB} \setminus G_{EB}$, analogously for L'.

Step 1. Complete the typing of $\hat{G}'$ (cf. Fig. 6.8).

a) $\forall x \in G_N \bigcup G_E\colon type_{G'}(x) = type_G(x)$,

b) $\forall n \in G_N\colon \hat{type}_{\hat{G}'}(n) = \hat{type}_{\hat{G}}(n)$,

c) $\forall b \in G'_{\overline{B}}\colon \exists e\colon n \to b \in G'_{\overline{E}}\colon type_{G'}(b) = b_{TG}$ with $\exists e_{TG}\colon type_{G'}(n) \to b_{TG} \in TG$,

d) $\forall e\colon n \to b \in G'_{\overline{E}}\colon type_{G'}(e) = e_{TG}$ with $e_{TG}\colon type_{G'}(n) \to type_{G'}(b)$,

e) $\forall b \in G'_{\overline{B}}$ with $e\colon n \to b \in G'_{\overline{E}}\colon \hat{type}_{\hat{G}'}(b) = \hat{TG}(e_{TG}) \circ \hat{type}_{\hat{G}'}(n)$ with $e_{TG}\colon type_{G'}(n) \to type_{G'}(b)$.

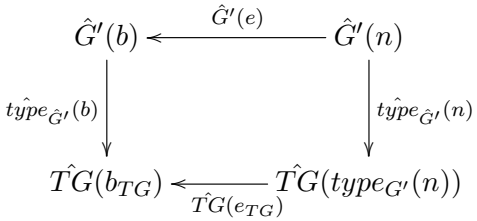


Figure 6.8: Illustration for Step 1 of Def. 6.16.

Step 2. Complete the typing of $\hat{L}'$.

a) $\forall x \in L_N \bigcup L_E$: $type_{L'}(x) = type_L(x)$,

b) $\forall n \in L_N$: $\hat{type}_{\hat{L}'}(n) = \hat{type}_{\hat{L}}(n)$,

c) $\forall b \in L'_{\overline{B}}$: $type_{L'}(b) = type_{G'}(b)$,

d) $\forall e\colon n \to b \in L'_{\overline{E}}$: $type_{L'}(e) = e_{TG}$ with e_{TG}: $type_{L'}(n) \to type_{L'}(b)$,

e) $\forall b \in L'_{\overline{B}}$: $\forall x \in \hat{L}'(n)$ with $e\colon n \to b \in L'_{\overline{E}}$: $\hat{type}_{\hat{L}'}(b)([x]) = \hat{TG}(e_{TG}) \circ \hat{type}_{\hat{L}'}(n)(x)$ with e_{TG}: $\hat{type}_{\hat{L}'}(n) \to \hat{type}_{\hat{L}'}(b)$ and $x \in [x]$,

Step 3. Complete the typing of $\hat{K}'$ and $\hat{R}'$.

a) $\forall x \in K_N \bigcup K_E$: $type_{K'}(x) = type_K(x)$,

b) $\forall n \in K_N$: $\forall x \in \hat{K}(n)$: $\hat{type}_{\hat{K}'}(n)(x) = \hat{type}_{\hat{K}}(n)(x)$,

c) $\forall b \in L'_{\overline{B}}$: $type_{K'}(b) = type_{L'}(b) \wedge \hat{type}_{K'}(b) = \hat{type}_{L'}(b)$,

d) $\forall e \in L'_{\overline{E}}$: $type_{K'}(e) = type_{L'}(e) \wedge \hat{type}_{K'}(e) = \hat{type}_{L'}(e)$,

e) analogously for $\hat{R}$.

◇

Lemma 6.17. In Def. 6.16, the typing morphisms of $\hat{G}'$, $\hat{L}'$, $\hat{K}'$, and $\hat{R}'$ are well-defined.

Proof. Let there be a weak typed composite IC-rule $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{K} \xhookrightarrow{\hat{r}} \hat{R}, \hat{typ}e)$ with $\hat{L}$, $\hat{K}$, and $\hat{R}$ being weak composite IC-graphs typed over the composite IC-graph $\hat{TG}$ by the triple $\hat{typ}e = (\hat{type}_{\hat{L}}\colon \hat{L} \to \hat{TG}, \hat{type}_{\hat{K}}\colon \hat{K} \to \hat{TG}, \hat{type}_{\hat{R}}\colon \hat{R} \to \hat{TG})$ being typing composite IC-morphisms (see Def. 6.6). Additionally, let there be a weak typed composite IC-graph $\hat{G}$ as host typed via $\hat{type}_{\hat{G}}\colon \hat{G} \to \hat{TG}$ and a consistent IC-composite transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$.

The conversion starts with the conversion defined in Def. 5.28 which ignores typing. Lemma 5.29 shows that the resulting $\hat{G}'$, $\hat{L}'$, $\hat{K}'$, and $\hat{R}'$ are (typeless) composite graphs and that $\hat{l}'$, $\hat{r}'$, and $\hat{m}'$ are composite morphisms. Since subsequent conversion steps concern typing morphism only, i.e., main structures are left untouched, Lemma 5.29 still holds.

It remains to show that $type_{\hat{G}'}$,$type_{\hat{L}'}$,$type_{\hat{K}'}$, and $type_{\hat{R}'}$ are typing composite IC-morphisms yielding typed IC-graphs $\hat{G}'$, $\hat{L}'$, $\hat{K}'$, and $\hat{R}'$ and that $\hat{l}'$, $\hat{r}'$, and $\hat{m}'$ are well-defined typed IC-morphisms. In the following, the typing morphisms are examined corresponding to the step they have been defined in Def. 6.16. Existing typing in $\hat{G}$, $\hat{L}$, $\hat{K}$, and $\hat{R}$ are taken over in item 1 and 2 of each step. Therefore, the subsequent reasoning focuses on the structural extensions and their typing only.

Step 1: $\hat{type}_{\hat{G}'}$ is a well-defined typing composite IC-morphisms and $\hat{G}'$ is a typed composite IC-graphs. On network level, according to the definition of composite network graphs, each export node points to exactly one body in a unique way. This is exploited by items c) and d) which find the corresponding types of the new body node and the new export-body edge with the help of TG and the typing of the given export node type.

This is similarly exploited on object level where the types of body graph elements are directly deduced from the types of related export graph elements and their related body graph elements in $\hat{TG}$. Figure 6.8 illustrates this in a diagram. The commutativity of that diagram has to be shown in addition, i.e., $\forall b \in G'_{\overline{B}}$: $\forall x \in \hat{G}'(n)$ with $e\colon n \to b \in G'_{\overline{E}}$ the following must hold: $\hat{type}_{\hat{G}'}(b) \circ \hat{G}'(e)(x) = \hat{TG}(e_{TG}) \circ \hat{type}_{\hat{G}'}(n)(x)$.

$\hat{type}_{\hat{G}'}(b) \circ \hat{G}'(e)(x) \overset{Def.\ 5.28\ \text{Step}\ 1f)}{=} \hat{type}_{\hat{G}'}(b) \circ id_{\hat{G}'(n)}(x) \overset{e)}{=}$
$\hat{TG}(e_{TG}) \circ \hat{type}_{\hat{G}'}(n) \circ id_{\hat{G}'(n)}(x) = \hat{TG}(e_{TG}) \circ \hat{type}_{\hat{G}'}(n)(x)$.

Furthermore, one can argue that for all $b \in G'_{\overline{B}}$: $\hat{G}'(b)$ is a typed IC-graphs, i.e., $\hat{G}'(b)$ does not contain containment cycles and there is at most one container (see Def. 4.7) since its structure is similar to its export and the export is a typed IC-graph.

Step 2: $\hat{type}_{\hat{L}'}$ and $\hat{m}'$ are a well-defined typing composite IC-morphism and a typed composite IC-morphism, resp., and $\hat{L}'$ is a typed composite IC-graph. For the proof of this step, consider the diagram in Fig. 6.9. So far, with regard to the figure the following is known: Step 1 in Lemma 5.29 states the commutativity of the front face (A). Step 2 in this lemma shows that the top face (B) commutes and that $\hat{type}_{\hat{G}'}(b)$ is well-defined. The right face (C) commutes since exports are part of the original weak typed composite IC-graphs $\hat{L}$ and $\hat{G}$ being matched by $\hat{m}$. Furthermore, $\hat{type}_{\hat{G}'}(m(n))$ and $\hat{type}_{\hat{L}'}(n)$ are well-defined for the same reasons. Consequently, the concatenation of (A) and (B) commutes as well as the concatenation of (A), (B), and (C) where related typing IC-morphisms are also well-defined.

It remains to show that (D) commutes and that $\hat{type}_{\hat{L}'}(b)$ is well-defined. In accordance with the definition of epimorphisms in [23, Def. A.12], for an epimorphism $e\colon A \to B$ and two morphisms $f, g\colon B \to C$ it holds that $f \circ e = g \circ e \Rightarrow f = g$. This definition is exploited in the following. $\hat{L}'(e)\colon \hat{L}'(n) \to \hat{L}'(b)$ can be considered as an epimorphism in the category GRAPHS as it maps from elements in the graph $\hat{L}'(n)$ to classes of elements in $\hat{L}'(b)$, i.e., $\hat{L}'(e)$ is surjective. To show: $\hat{type}_{\hat{L}'}(b) = \hat{type}_{\hat{G}'}(b) \circ \hat{m}'(b)$. $\forall x \in \hat{L}'(n)$, $\hat{type}_{\hat{L}'}(b) \circ \hat{L}'(e)(x) \overset{Def.\ 5.28\ Step2.i)}{=} \hat{type}_{\hat{L}'}(b)([x]) \overset{e)}{=} \hat{TG}(e_{TG}) \circ \hat{type}_{\hat{L}'}(n)(x) \overset{commut.(A)+(B)+(C)}{=} \hat{type}_{\hat{G}'}(b) \circ \hat{m}'(b) \circ \hat{L}'(e)(x)$. Therefore, (D) commutes and $\hat{type}_{\hat{L}'}(b)$ is well-defined since $\hat{type}_{\hat{G}'}(b) \circ \hat{m}'(b)$ is already shown to be well-defined. This yields the commuting concatenation of (A), (B), and (C) as

described above. Moreover, it follows that the outer square (the concatenation of (A)+(B)+(C)+(D)) does also commute, i.e., the typing of exports and new bodies are consistent.

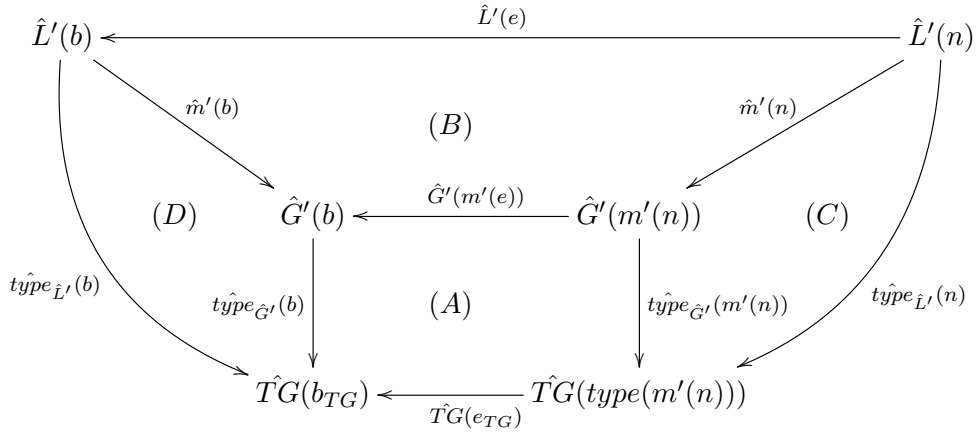


Figure 6.9: Illustration for the proof of Step 2 in Lemma 6.17.

It is rather easy to see that all $\forall b \in L'_{\overline{B}}$: $\hat{L}'(b)$ are IC-graphs. Due to the construction of $\hat{L}'(b)$, containment cycles cannot occur since each element $[x]$ corresponds to an element $x' \in \hat{G}'(b)$ in a structure preserving way. That means, containment cycles in $\hat{L}'(b)$ may only occur if there are also containment cycles in $\hat{G}'(b)$ which are shown to be absent. For the same reason, the *at most one container* constraints holds as well for $\hat{L}'(b)$.

Step 3: $t\hat{y}pe_{\hat{K}'}$ and $t\hat{y}pe_{\hat{R}'}$ are well-defined typing composite IC-morphisms and $\hat{K}'$ and $\hat{R}$ are typed composite IC-graphs. The construction of $\hat{K}'$ and $\hat{R}'$ as well as their typing of new bodies and body-export edges directly correspond to $\hat{L}'$. $\hat{K}'$ and $\hat{R}'$ are therefore well-defined IC-graphs. □

Part II

Implementation and Tooling

Now that the concepts of composite graphs and composite graph transformation are theoretically elaborated and formalized, the present part reports on the efforts and results which bring these concepts to practical life.

This is achieved by developing dedicated tools based on the programming language Java and especially based on technologies in the context of Eclipse [22]. Eclipse is a huge open source community whose projects aim for an open and extensible development platform able to support the developer throughout the entire software life-cycle. Moreover, the Eclipse platform with its sophisticated plugin marchitecture has also proven very useful as application platform (cf. `http://www.eclipse.org/community/rcpos.php`) in a number of scenarios ranging from 3D graphical editors over homepage builder to network tools. The community is supported by the non-profit Eclipse Foundation which is driven by many famous companies as, e.g., IBM.

These are a number of reasons to choose Eclipse as a very first platform for the implementation of composite modeling. In fact, composite models are implemented by means of the Eclipse Modeling Framework (EMF) [27] which can thoroughly be called the standard modeling framework in the Eclipse environment. Furthermore, composite transformations of EMF models are implemented based on Henshin [37] a tool for in-place EMF model transformation. Both composite EMF models and composite EMF transformations are presented in Chapters 7 and 8 by means of the running example.

Chapter 7

Composite EMF Models

This chapter presents the implementation of composite models based on the Eclipse Modeling Framework (EMF) [27]. It is structured as follows: At first, EMF itself is introduced and its features and properties are highlighted. Afterwards, CompoEMF [73], the composite EMF model implementation is presented including its dedicated editor support. CompoEMF has been developed in collaboration with Tim Schäfer [72].

7.1 Meta Modeling with EMF

The Eclipse Modeling Framework allows defining models and modeling languages by means of so-called structured data models. They can be created and modified using dedicated wizards, editors and an API. Furthermore, EMF provides the capability of generating Java code out of an EMF model which can then be customized to fit ones needs. In the following key concepts of EMF are outlined.

7.1.1 Ecore

The most important EMF model is Ecore which serves as general (meta) model. Moreover, Ecore is an EMF model itself, i.e., it is typed over itself and therefore sort of a bootstrap for EMF models. Figure 7.1 gives an impression of the Ecore meta model by means of a subset of Ecore with its most important classes and relations shown. Abstract classes are indicated by italic letters and are colored by a slightly darker background. The concrete classes located in the lower part of Fig. 7.1 essentially correspond to common entities in UML class diagrams, i.e., EPackage, EClass, and EAttribute correspond to packages, classes, attributes. The class EDataType represents certain primitive and non-primitive types like int (EInt) and Integer (EIntegerObject). EReference corresponds to associations whereas references are always directed.

Additional attributes further support the use of these Ecore elements in different ways, e.g., the attribute nsURI of EPackage is very important as it assigns a globally unique namespace to a package to allow for its unambiguous identification. Classes may be declared as abstract classes or interfaces by corresponding attributes of EClass. Furthermore, references may be explicitly equipped with lower and upper bound and be declared as derived references which shall rather be calculated in a certain way.

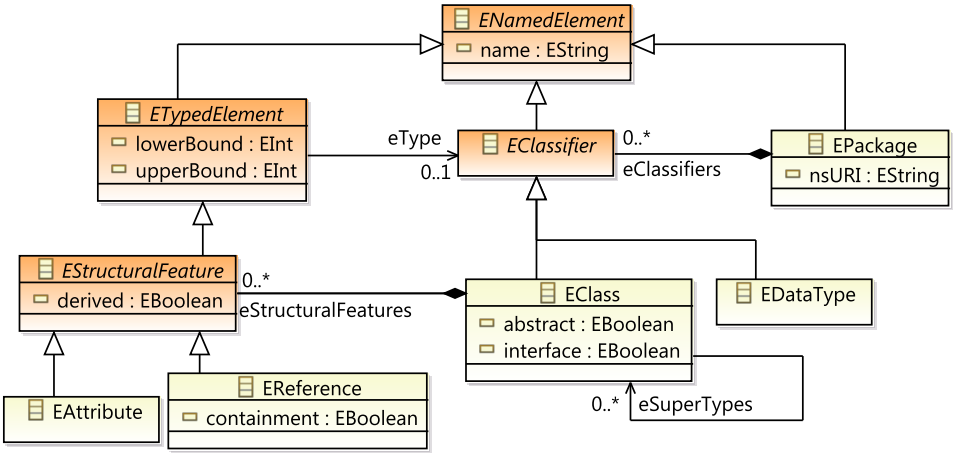


Figure 7.1: Subset of the Ecore meta model exposing the most important parts.

Since Ecore is typed over itself, it can be quit tricky to find the right term when speaking of a certain model layer. Therefore, to avoid confusion, in the following the terms Ecore (meta model), Ecore model and EMF instance (model) are respectively used referring to Ecore itself, a model directly typed over Ecore, and in turn an instance of the latter. If either of these models is to be addressed, the term EMF model is used.

Example 7.1. Figure 7.2 shows the running example by means of two Ecore models, i.e., the department management is depicted at the left while the project management is given at the right. They are already shown in a similar manner in Fig. 4.1. However, this time both models are truly Ecore models with their concrete syntax visualized by means of the EcoreTools editor shipped with EMF.

Their abstract syntax can be easily deduced with help of Fig. 7.1. In Fig. 7.2, each named class is an instance of `EClass` at which the value of the `EClass.name` property is set correspondingly. Classes with a name in italic letters, e.g., `Person`, are declared as being abstract with the help of the `EClass.abstract` property set to `true`. Attributes such as `pno` are de-

fined by instances of `EAttribute` again with a corresponding value set to `EAttribute.name`. Note that the attribute types `EString`, `EInt`, and `EFloat` are Ecore-specific types which are assigned via the reference `EAttribute.eType` and which shall provide a platform independent type system.

Not visualized in Fig. 7.2, both class diagrams are wrapped each by an instance of `EPackage` whose attribute values are given below the diagrams in the lower area of Fig. 7.2. As explained already, especially the name space URI (Ns URI) is an important value and shall be unique.

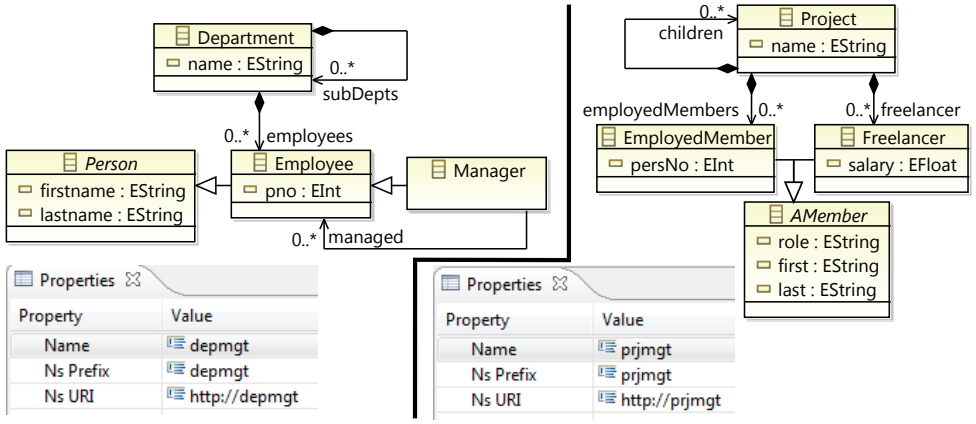


Figure 7.2: The running example of a department management (left) and project management (right) model by means of Ecore models.

△

7.1.2 Containment

The concept of containments, in UML known as composition, plays an important role in EMF as it describes an ownership relation aiming at hierarchical containment structures on instance level. Consequently, objects must belong to at most one container and (transitive) cyclic containment is forbidden. The attribute that declares relations to be containments is shown in Fig. 7.1 as part of class EReference. Another desired property of EMF models is a single root element, i.e., an object which transitively contains all other model objects. While this third property plays a minor role in the formalization above, in terms of the EMF tooling it is important as all three properties together allow EMF to persist arbitrary models along their spanning containment tree.

The abstract syntax of the Ecore models in Fig. 7.2 provides such valid containment structures. The instances of `EPackage` each serve as the root element which contains related `EClass` instances which in turn contain their `EAttribute` instances as well as the `EReference` instances.

7.1.3 Remote References

EMF models are primarily persisted as a so-called resource (usually a file) in a format called XML Metadata Interchange (XMI) [64] an OMG standard for serializing metadata using XML. The hierarchical structure of such an XMI document closely follows the containment hierarchy of its persisted model and also provides a way to uniquely identify elements in that model. Exploiting this, EMF is capable to fragment models across different resources by letting one model's object relate to a remote model's object via a reference, then called cross-document reference or remote reference. Beyond that, the remote object is initially represented by a proxy object which is resolved to the original element not before it is actually needed.

7.1.4 Code Generation

Instances over an Ecore model can be created in two ways. The first one yields so-called *dynamic instances* which can be considered as on-the-fly instances, i.e., objects are actually instances of an EMF internal generic Java class and just point to the model class whose instance the object pretends to be.

However, the second and rather common way is to exploit generated model code. To this end, EMF provides a generator facility based on the template language JET [57]. For each Ecore model class a corresponding Java class is generated including member variables that represent provided attributes and references. In fact, EMF generates Java interfaces and implementation classes where the former only expose accessors to properties according to the Ecore model while the latter includes additional functionality as notification handling, list management, proxy resolution and so on. Then, instances of the original Ecore model are created by instantiating the generated Java classes.

The EMF generator offers additional facilities such as the generation of a basic tree-based editor dedicated to edit related instance models and a facility to generate skeletons of unit test classes. Note that the generated code is enclosed by plugin projects which seamlessly integrate into Eclipse.

7.1.5 Editor support

The basic editors in the Eclipse Modeling Framework are the tree-based editors providing basic CRUD operations. Since EMF offers generator capabilities leading to such editors it is not surprising that the standard editor for Ecore models is the one generated out of the Ecore meta model itself. However, EMF also ships a generic tree-based editor, called Sample Reflective Editor, that allows to edit arbitrary EMF models, i.e., Ecore models and instance models.

Since models are rather graph-like structures, a visual representation is often more adequate. Therefore, EMF comes with a GMF [34] based diagram editor for Ecore models as well which yields diagrams as depicted in Fig. 7.2.

EMF instances are by default edited using a generated tree-based editor or the Sample Reflective Editor. A generic visual editor for EMF instances is not available. However, dedicated diagram editors can be generated with some effort using frameworks like GMF and EuGENia [54].

Example 7.2. Figure 7.3 shows the standard EMF generated tree-based editor resulting from a department management Ecore model as depicted in Fig. 7.2. It essentially consists of a tree-based editor view (in the upper area) showing the overall structure along the containment hierarchy and a synchronized properties view (in the lower area) providing access to the attribute values of the object selected in the editor view. New objects can be added by means of context menu entries (not shown). △

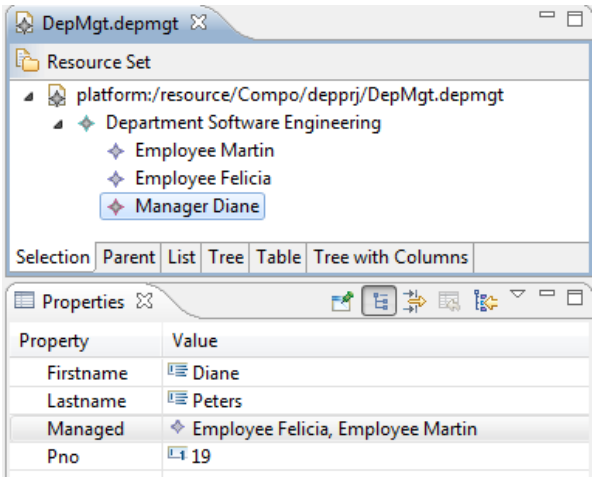


Figure 7.3: Sample EMF generated editor generated from an Ecore model.

7.2 CompoEMF: A Framework for Composite EMF Models

This section presents CompoEMF, a first implementation of composite graphs with inheritance and containment having EMF as underlying technology. It starts with a discussion of the main requirements towards the implementation of CompoEMF and is followed by an introduction of its underlying meta model. Some technical points are laid open thereafter concerning loose containment in interfaces and the delegation of attribute values along interface chains. Finally, the tool support of CompoEMF is highlighted.

7.2.1 Requirements

Different ways are conceivable of applying the concepts of composite models to EMF. However, the current implementation of CompoEMF pursues a number of promising goals. One of the main goals is to achieve high reuse of existing EMF models, i.e., ordinary EMF models shall easily and especially non-invasively become body component parts, i.e., without the need to modify them. As a result, interface-related information needs to be stored elsewhere.The implementation approach shall furthermore, analogously to the formalism, support the pretty useful capability of having interface object structures being simpler than their body models. Moreover, interfaces shall be especially able to carry a number of single objects without an explicit containment hierarchy. Note that this requirement conflicts with the desired EMF model property of having a single root element.

Although the concept of composite IC-graphs does not explicitly consider attributes, in practice attribute values are of particular interest when exporting and importing objects. Therefore, a further goal is to treat attributes analogously to objects in composite EMF models, i.e., attributes need to be explicitly exported and imported in order to let components share their values.

7.2.2 CompoEMF Meta Model

Complying with the requirements above, in CompoEMF each component model is essentially constituted by a body being an ordinary EMF model. In order to define export and import interfaces it appear obvious to exploit EMF models again. This further increases the reuse of EMF and actually reminds of the underlying formalism where bodies, exports, and imports are graphs. However, while Ecore serves as meta model for body models, it lacks of structures required by interface models namely the references to corresponding bodies and imports. CompoEMF introduces dedicated interface meta models, called *EExport* and *EImport*, which extend the kernel classes of Ecore and enrich them with remote references to body and export models. Interface models can thus be treated similarly to ordinary EMF models and existing EMF features/tools can be reused (to a certain extent) such as code generators, generic editors, etc.

Figure 7.4 illustrates this constellation in form of a meta level hierarchy. Level M3 shows Ecore as topmost meta model which is typed over itself, i.e., Ecore on M4 and so on. One level below on M2, Ecore, EExport, and EImport are all typed over Ecore on M3 and serve as meta models for body and interface models on M1. Note that on level M2, the classes of EExport and EEImport inherit elements from Ecore and enrich them with additional references. This fact is denoted by the (not UML compliant) inheritance relation between the interface models and Ecore. Obviously, level M0 specifies the instance level with EMF instance models and instances of interfaces defined on M1.

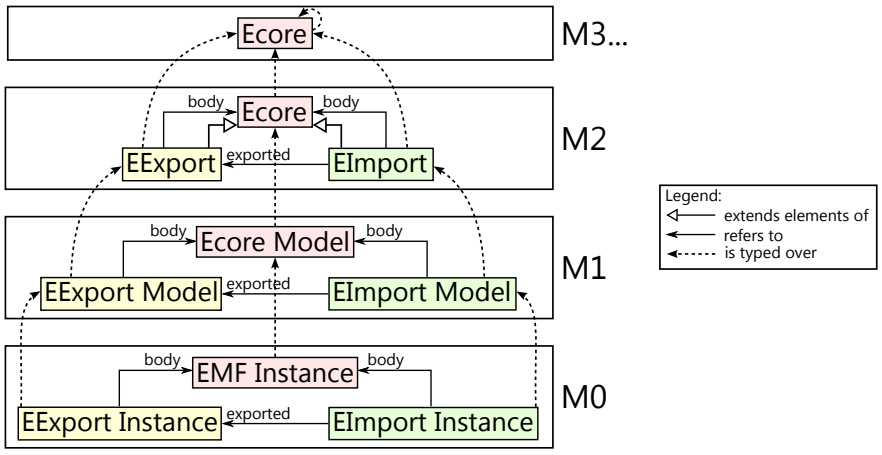


Figure 7.4: Meta layers corresponding to bodies and interfaces.

In the formalism of composite graphs, network graphs define the kind of graphs on object level in terms of component parts, i.e., whether they are bodies, exports, or imports. Since *EExport*, *EImport*, and Ecore already declare the component part type of a model, the question arises whether one needs a similar second layer in practice too. The difficulty here, however, comes with the design decision that body models are original EMF models, i.e., they are not aware of their interfaces nor of interconnected components. Therefore, CompoEMF provides an additional Ecore model, called *Composite*, which maintain such information.

In the following, EExport, EImport, and Composite are presented in more detail.

EExport and EImport. Figure 7.5 depicts details of the meta models EExport and EImport together with parts of Ecore. The original Ecore meta classes are shown in the center (compare Fig. 7.1) while elements belonging to EExport and EImport are arranged at the left and right. The *from* annotation in brackets additionally indicates the origin of each meta model element. Especially notice the outgoing references introduced by each element of EExport and EImport. For example, the Ecore class `EClass` is inherited by `ExportedClass` and `ImportedClass` both referring to their corresponding body element via reference `bodyClass` while `ImportedClass` also has a reference to `ExportedClass`. In fact, each of the three (meta) models shown in Fig. 7.5 is modularized in a separate resource[1] and references like `bodyClass` are actually remote references.

[1]The meta models are de facto files each located in a separate Eclipse plugin together with their generated (meta) model code.

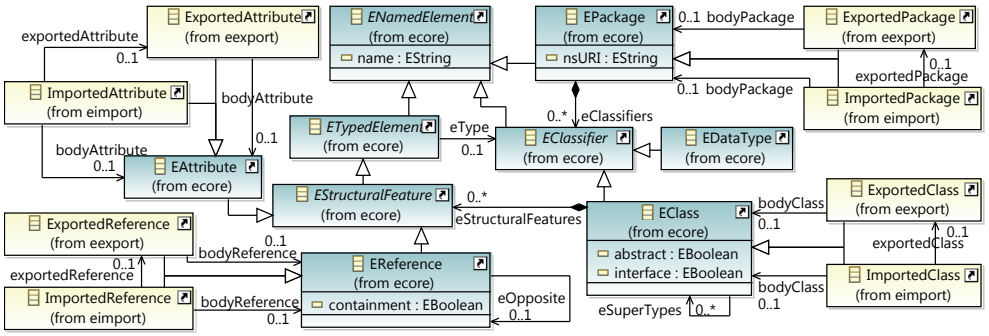


Figure 7.5: Extended Ecore model for composite EMF models. Ecore metaclasses are arranged in the center and annotated with their origin.

In order to further support the special needs of composite EMF models, the EMF generator is utilized to generate model code and editor code for both EExport and EImport. The resulting model code is then essentially extended by a mechanism which allows multiple elements being not contained explicitly and by a delegation mechanism to pass attribute values along interfaces. The editor code is also adapted and tailored, e.g., by unique icons, by a wizard to easily choose elements from a body to be added to the export, and by a capability to create dynamic interface instance models.

Example 7.3. Figure 7.6 shows the models of the running department/project management example by means of a composite EMF model on level M1 (cf. Fig. 7.4). Unsurprisingly, in general it looks pretty similar to what is shown throughout Chap. 6. The body models are located in the upper area with the department management Ecore model at the left and the project management Ecore model at the right. Their export and import models are located below. Remote references between the model elements are not shown but are denoted by arrows between the models. The packages wrapping the two body models, the export model and the import model are not shown either[2]. Close to each model the corresponding file name is shown. Since the body models are instances of Ecore, their file extension is *.ecore*. Analogously, since the department export and the project import models are instances of EExport and EImport their file extensions are *.eexport* and *.eimport*, respectively. △

Composite Ecore model. CompoEMF also includes a model to represent the network structure of a composite EMF model, called *Composite* Ecore model. In Fig. 7.7, the `Composite` Ecore model is given together with two elements of the Ecore meta model. The class `Composite` is the root ele-

[2]The diagrams are shown by means of the original EMF EcoreTools diagram editor which does not support the illustration of surrounding packages.

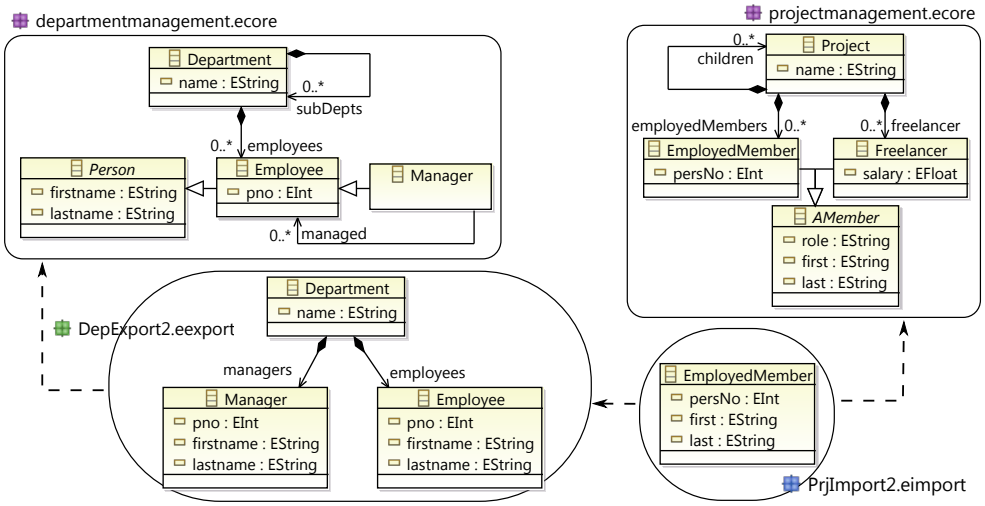


Figure 7.6: Example composite EMF model similar to the running example in Chap. 6 (e.g. compare with Fig. 5.4).

ment and may contain arbitrary `ComponentParts` such as `Component` and `Interface`. `ComponentParts` remotely refer to the root element of their represented component part model (`rootEObject`) and to the meta model of that part (`ePackage`). It also contains a derived attribute `EPackageUri` whose value is directly derived from the `nsUri` attribute value of the associated `:EPackage` object. This attribute has been introduced to allow for a convenient matching only as explained later. A class `Component` represents a component by, on the one hand, referring to its body model and, on the other hand, also containing its interfaces. Note that the reference `interfaces` is directed opposite to what has been formalized so far. Nevertheless, this is a bidirectional reference and the correct direction is modeled as well by `body`. Interfaces are represented by the abstract class `Interface` and their children `ImportInterface` and `ExportInterface`. Please also note that in the current state of development, *Composite* does not support the representation of *weak* composite graphs which would allow uncoupled export interfaces.

Analogously to the interface meta models, model and editor code of the Composite Ecore model are adapted and extended. Moreover, a visual editor is generated by GMF [34] which offers more convenient capability of editing composite EMF models.

Example 7.4. Considering Fig. 7.6 in the context of a *Composite* model, the two body models in the upper area are represented by two `:Component` objects contained in a single `:Composite` object. The `rootEObject` references of both components point to the root packages of the body Ecore model while the `ePackage` references point to the root package of the body Ecore meta models

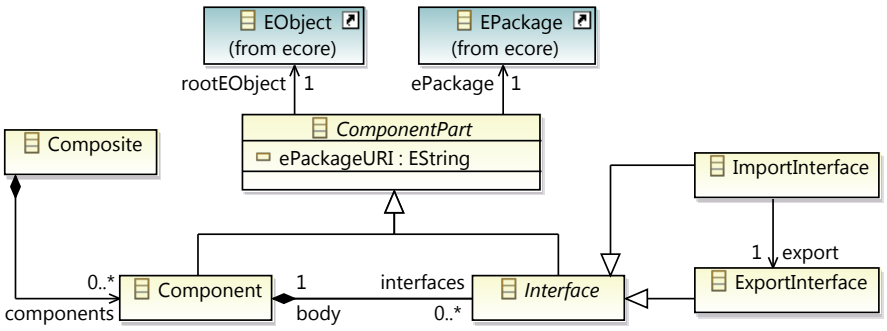


Figure 7.7: Composite Ecore model serving as network meta model for composite EMF models.

as this is the meta model of Ecore models. The department management related `:Component` contains the object `:ExportInterface` which points to the packages of the *DepExport2* model and the EExport meta model. The project management related `:Component` contains the object `:ImportInterface` which points to the packages of the *PrjImport2* model, the EImport meta model, and in addition to the `:ExportInterface` object of the department management component. △

7.2.3 Transparent Interface Container

A requirement based on the formalism is to allow interfaces models to carry multiple classes/objects not necessarily being contained, i.e., having more than one root element. In general, EMF supports multiple root elements in a single resource. The handling of such resources is comparatively inconvenient, though, e.g., having a single root each element can be treated similar for instance using the convenient traversal method over all contained elements while multiple roots require a special handling here.

The design decision for CompoEMF is to introduce an artificial container which is automatically created and transparent to the user. Therefore, the model code of *EExport* and *EImport* is appropriately extended to introduce a topmost container class, called `Container`, each time an interface model is created. Adding a new class to an interface model then also automatically results in a new containment reference running from `Container` to that class. Consequently, in interface instances a `:Container` object is the root element which is *able* to contain all objects if necessary. Note that the editor code is also adapted such that the `Container` class is hidden to the user.

Example 7.5. Given the composite EMF model in Fig. 7.6, Fig. 7.8 shows an overview of the real structure of *DepExport2*. The class `Container` is the root element and carries containment references to all classes in the model. △

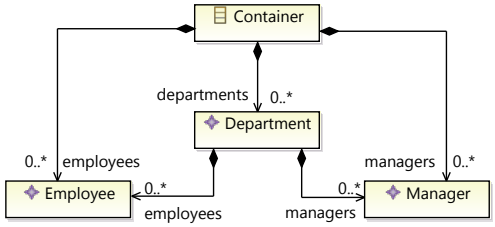


Figure 7.8: Real structure of interface model *DepExport2* in Fig. 7.6.

7.2.4 Delegation of Attribute Values

With the structures described above it is now possible to establish EMF component models which may share selected information with each other by exporting and importing objects. However, attributes values (often also called properties) of objects are of particular interest as they provide the actual data. For example, an object of type `Employee` is rather meaningless without the knowledge of the properties `name` and personnel number (`pno`) being "Felicia" and 512, respectively.

In CompoEMF, attributes are treated similar to classes and references: they must be explicitly exported and imported in order to be shared. Then, if the value of an imported attribute of an imported object is accessed, the request is delegated along the import-export interface to the original body element. This is illustrated in Fig. 7.9 which shows a small instance of the composite EMF model given in Fig. 7.6. In addition to the (remote) references between the models, dashed arrows denote the direction of the delegation. For example, an access to the value of the attribute `first` of the `:EmployedMember` object triggers a request to the import model (PrjImport2) which asks the export model (DepExport2) which in turn retrieves the actual value of the related `:Employee` object in the department management body. Finally, the resulting value is returned along the query chain.

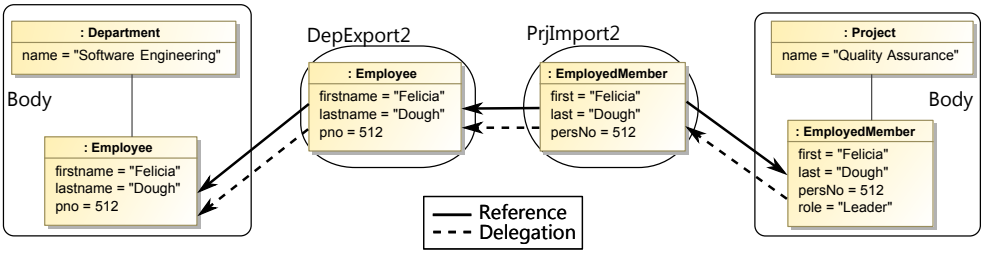


Figure 7.9: Composite EMF model illustrating the direction of references and delegation.

The delegation along interface models is technically achieved by exploiting an EMF feature called *setting delegates*. CompoEMF automatically attaches them to attributes defined in interface models which leads to a dynamic evaluation of requests [3] . Body models are, however, ordinary EMF models and imported elements in body models are themselves not aware of being imported. Consequently, they are not able to trigger the delegation. Moreover, having a closer look at Fig. 7.9, the direction of the required delegation is opposite to the reference import-body reference such that the body even does not know where to delegate to. To enforce the delegation anyway, an aspect-oriented approach by AspectJ [4] is applied, i.e., delegation code is planted into body objects at runtime. This technique works for both instance created with generated model code and for dynamic instances. Please find more details in the master thesis of Tim Schäfer [72].

7.2.5 Editor support

Since body models are ordinary Ecore models, the EMF editors described in the previous section are applicable to the full extent. The same holds true for EExport and EImport model which are Ecore models as well.

The interface meta models EExport and EImport provide EMF-generated tree-based editors which are slightly modified, e.g., by hiding the artificial container class (see Sect. 7.2.4). It is recommended, however, not to used them but the Composite model editor (see below) which transparently integrates editor parts of EExport and EImport and serves as common editor for all parts of a composite EMF model. Note that the Ecore diagram editor can be reused only to a certain extent as its tool palette merely provides the creation of elements from Ecore. Consequently, that editor is better to be used for visualization, editing, and deletion than for creation. It is conceivable, however, to build upon the Ecore diagram editor or even generate dedicated diagram editors for interface models in the future.

The *Composite* Ecore model comes with generated model code and a generated tree-based editor, too, as depicted in Fig. 7.10. It shows not only the network layer but allows the navigation into each refining model. Moreover, the highlighting of correspondences among bodies, exports, and imports and a convenient drag'n'drop functionality to easily assign body elements to exports and imports facilitate the handling of composite EMF models.

The modeler is further supported by a *Composite* diagram editor which is recommended as the main editor of composite EMF models. In Fig. 7.11, this editor show a composite EMF model scenario equal to the one in Fig. 7.6. A tool palette at the right offers the creation of component parts and to connect

[3] The recent implementation of setting delegates in CompoEMF caches attribute values to handle cases where the target is unreachable, e.g., due to network problems. However, the present state of the approach generally assumes connected models to be accessible.

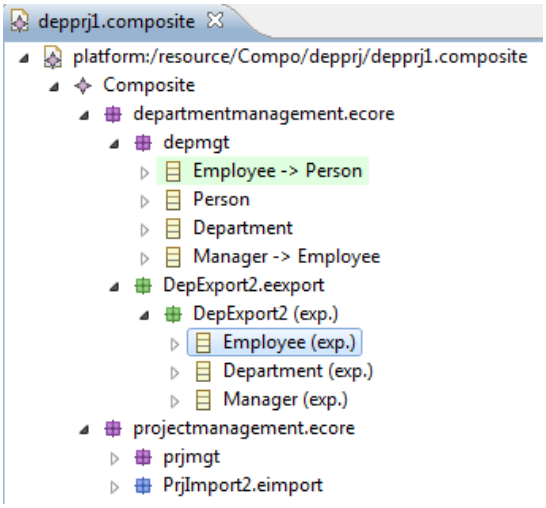


Figure 7.10: Composite tree editor shipped with CompoEMF.

them. Notice the "Export Interface (foreign Component)" which is yet inoperable but will serve as tool to create connect to (uncoupled) exports without the knowledge of their bodies. The left part of the editor shows the topology of the composite EMF model. Each node is equipped with small links that trigger certain actions such as the creation or assignment of models. Especially these kinds of action include mechanisms to relieve work from the modeler, e.g., the creation of a model takes the predefined meta model into account or a model assignment automatically sets the meta model.

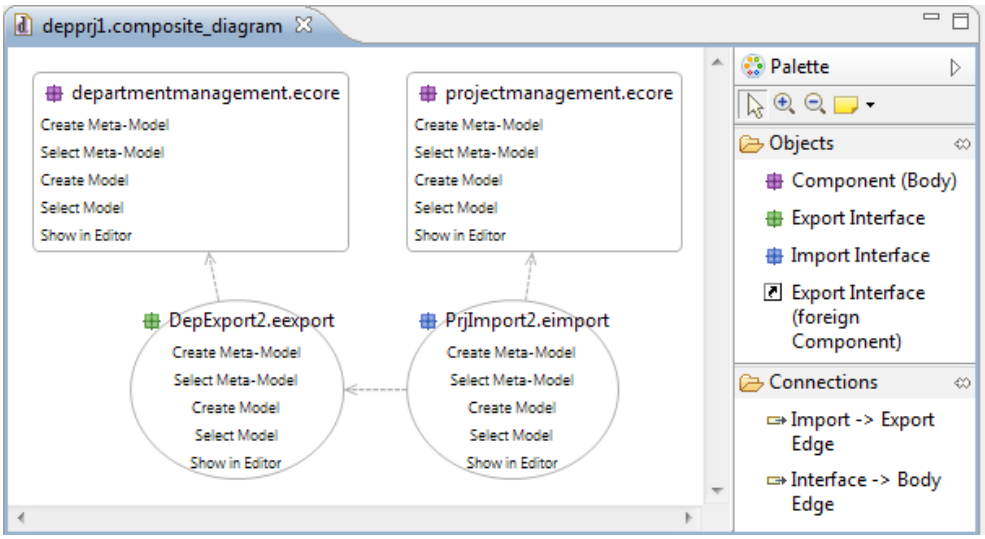


Figure 7.11: Composite diagram editor shipped with CompoEMF.

Moreover, when creating interface models a wizard opens up and allows, e.g., to specify the model elements to share. This is illustrated by the wizard in Fig. 7.12 which appears if the modeler is to create a new export model with a given body model as depicted in Fig. 7.6. Obviously, this wizard page does not allow the specification of all possible combinations. For example, an export interface like *DepExport2* in Fig. 7.6 with one reference (`employees`) in the body being represented by two reference (`employees` and `managers`) in the interface cannot be defined here. Nevertheless, this wizard page suits well to quickly specify simple interfaces. More subtle interfaces, however, should be established using the tree-based editor yet which opens using the *Show in Editor* link with the selected component part focused.

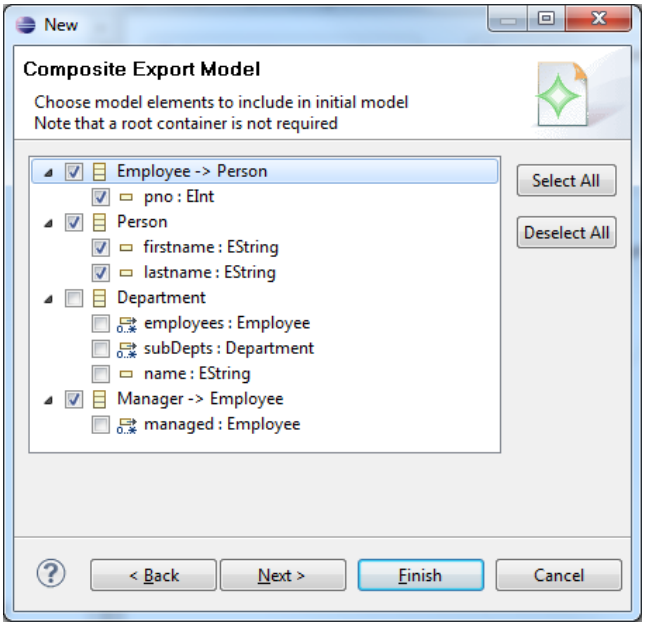


Figure 7.12: "New Export" wizard.

Considering the instance level, Ecore instance models do not have a default visual editor but only a tree-based editor (the generated editor or the Sample Reflective Editor). This holds consequently true for body instance models and interface instance models. Since *Composite* models are used on model and instance level, their tooling offers a tree-based editor and a diagram editor again.

Chapter 8
Composite EMF Model Transformation

This chapter presents an implementation of model transformation for Compo-EMF models which is joint work with Daniel Strüber within the scope of his diploma thesis [75] co-supervised by the author.

Since composite EMF models are formalized by means of graphs and graph morphisms, it is straight forward to use graph transformation concepts. Composite EMF transformation is conceptually based on composite IC-graph transformation whose theoretical findings open up a way for a convenient implementation. Particularly, in Chap. 5 reasoning and proofs have shown that a composite transformation step can be performed at object level by a number of coordinated ordinary graph transformation steps on each local graph. This technically results in the implementation of a composite EMF transformation engine called CompoHenshin [76] which exploits the capabilities of Henshin [37], a model transformation tool for EMF models.

The remainder of the present chapter is structured as follows: At first, Sec. 8.1 introduces Henshin which serves as background information for the subsequent presentation of CompoHenshin in Sec. 8.2.

8.1 Henshin: In-Place EMF Model Transformation

Henshin is a transformation language as well as an engine and comes with a rich tooling environment (compare Fig. 8.1) for EMF model transformation. Model transformations can be performed on arbitrary EMF models in an in-place fashion, i.e., changes are applied on models directly without the need of a copy. Thus, the engine is obviously specialized on transformation within the range of the same underlying meta model, called endogenous transformation. However, exogenous transformations, i.e., transformations where underlying source and target meta models differ, are supported as well.

As a successor of EMF Tiger [9] it is also based on graph transformation concepts but extends the transformation language of EMF Tiger considerably.

Henshin offers a powerful declarative model transformation language that is represented by an EMF model itself. It conceives transformation rules as the main artifacts which can furthermore be enriched by powerful application conditions and flexible attribute computations based on Java or JavaScript. Rules may even be nested which offers a forall-operator on patterns. The transformation language also provides control structures for rule applications in a modular way, i.e., so-called transformation units with predefined control-flow semantics may be nested arbitrarily in order to define even more subtle control-flows. A set of rules and transformation units is subsumed as transformation system and stored the EMF way in a so-called Henshin file in the XMI format (see Sec. 7.1).

The tool environment primarily consists of a transformation engine, a tree-based and a visual editor, as well as a graphical state space exploration tool that allows formal reasoning.

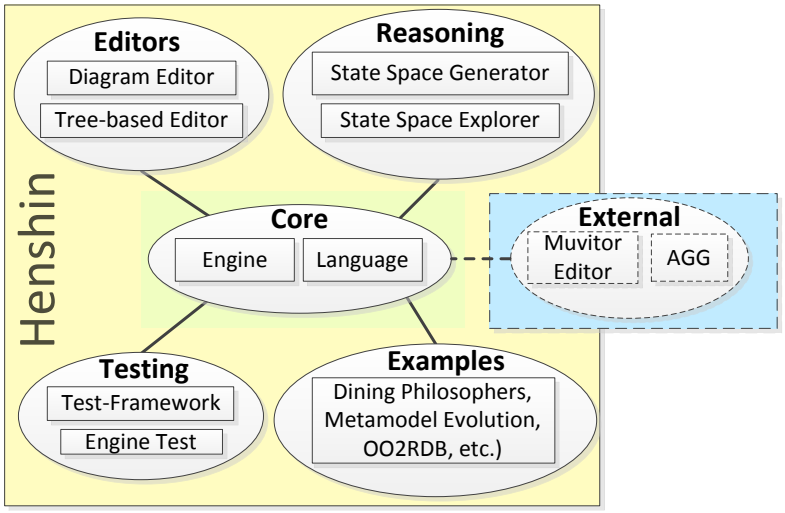


Figure 8.1: Outline of Henshin showing its kernel and tool environment.

In the following, the meta model of the transformation language is informally described also serving as foundation of CompoHenshin as explained in the Sec. 8.2. A more detailed view on Henshin's tool set is given afterwards.

8.1.1 Transformation Meta Model

The Henshin transformation language originates from graph transformation concepts what is apparently reflected by Henshin's meta model shown in Fig. 8.2. Note that most properties of model elements and multiplicities are neglected in order to focus on the structure of the transformation meta model. Similarly to the definitions in Chap. 3, a `Rule` basically consists of a left-hand side and a right-hand side `Graph`, also referred to as LHS and RHS. Graphs

may have attributed `Node`s being connected by `Edge`s where in addition each `Attribute` may carry a value. The identity of nodes in LHS and RHS is defined over `Mapping`s contained in the rule. The overall root element which contains all other elements in the meta model is `TransformationSystem`.

The structure defined over this meta model is used to match a corresponding structure of EMF classes and references while the `Attribute.value` furthermore restrict valid matches. To capture the meta modeling of EMF models, i.e., the typing of model elements, each node, attribute, and edge is equipped with type information by the reference `type` pointing to an `EClass`, `EAttribute`, and `EReference`, respectively. Using these generic Ecore meta model elements (cf. Sec. 7.1.1) enables Henshin rules to match arbitrary EMF-based models. For a proper matching, these types are required to fit in with the types of the target structure.

Note that the typing references once again resemble the formalism in Chap. 3 where mappings of typing morphisms run from elements of one graph to the elements of another graph being interpreted as types in a typed graph. Since EMF does not allow parallel edges of one type between the same nodes, the identity of edge mappings can be easily deduced and is therefore not modeled explicitly.

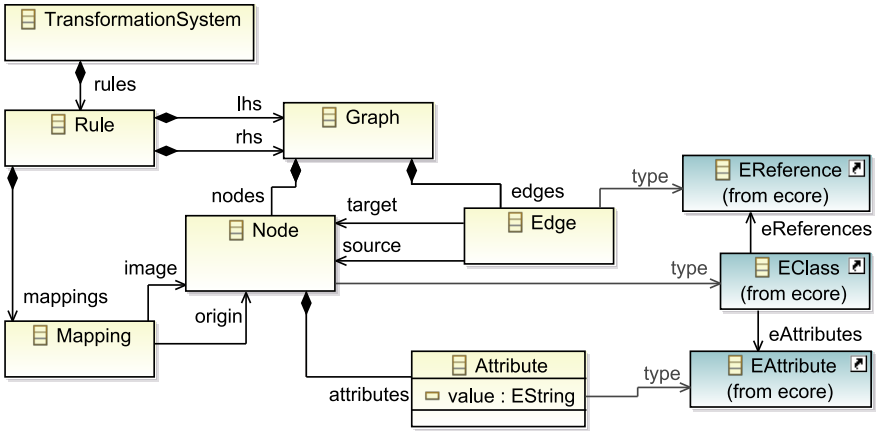


Figure 8.2: Simplified Henshin meta model concerning transformation rules.

Henshin's transformation meta model covers additional concepts not yet incorporated into the approach of composite modeling. For example, rules may also be equipped with conditions over attributes which further restrict the number of valid matches. Additionally, the LHS (and RHS) may contain so-called application conditions which are model patterns that can appear in a positive or a negative form and that can even be nested arbitrarily and combined in logical formulas. Their successful matching, possibly (partially) predefined by the matching of the LHS or another application condition pattern, affects the overall matching. There are other very interesting features of Henshin as pa-

rameters to define partial pre-matches or to preset possible attribute values, nesting of rules to specify for-all semantics and transformation units expressing control flows. As part of Henshin's transformation language, they all have their corresponding elements and structures in the meta model. Please refer to Henshin's website [37] and to various publications [3, 48, 49, 43, 10] for more details and example application scenarios.

8.1.2 Rule Application

Rules can be applied via Henshin's API or via a wizard dialog shipped with Henshin. The overall transformation procedure starts with the matching. To this end, the explicit type information of nodes is used, i.e., a node whose `type` reference points to a certain `EClass` will match to objects of this type only. This analogously applies to edges which implicitly point to an `EReference` and only match such references typed over a corresponding `EReference`. If a match has been found that also passes all checks concerning the gluing condition, an object of type `Match` is instantiated which stores the mappings between elements of the rule and element of the host model. The match can either be dismissed in order to determine other possible matches, or be used as a base to apply the given rule on. The rule application is performed by calculating and accumulating all model changes and by executing them together afterwards.

8.1.3 Tool Environment

Fig. 8.1 presents an outline of the Henshin tool environment arranged around the core. It consists of a visual diagram editor and a tree-based editor for transformation systems where both offer a different representation of the same content. In particular, the diagram editor exploits an integrated view of rules where LHS and RHS are merged to a single view with created and deleted model elements marked by dedicated stereotypes. This allows the concise and fast definition of rules. A state space generator and the sophisticated visual state space explorer support the reasoning of transformations by model checking. In Fig. 8.3, screenshots of the diagram editor, the tree-based editor, and the state space explorer are shown to gain a closer impression of how these tools look like. The use of rules and transformation units are illustrated by a number of given examples covering different settings. They also include the use of Henshin's transformation engine API. In order to help developers and modelers to test the behavior of their transformation system in an automatic way, a transformation test-framework is shipped with Henshin which is also used to test Henshin's transformation engine.

Since the transformation concepts of Henshin are close to graph transformation concepts, it is possible to export the rules to AGG [1], a tool environment for algebraic graph transformation where they might be analyzed concerning conflicts and dependencies of rule applications as well as their termination.

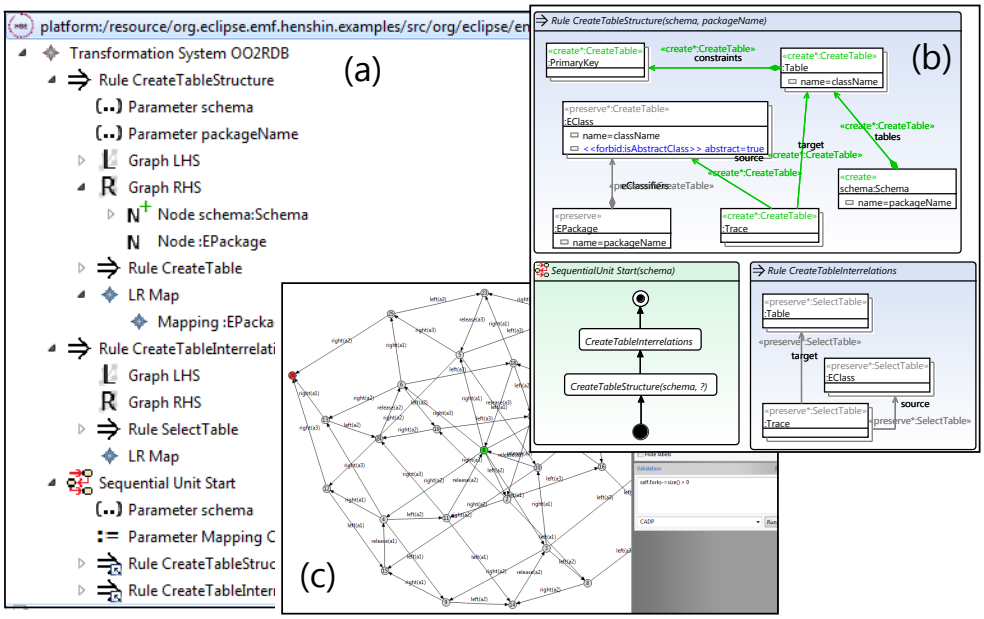


Figure 8.3: Samples of Henshin's tree-based editor (a), the diagram editor (b), and the state space exploration tool (c).

Furthermore, the TFS group [79] at the Technical University of Berlin offers yet another visual multi-view editor [80] for Henshin transformation systems based on the Graphical Editing Framework (GEF) [32] and a GEF-based framework designed for the development of multi-view editors, called MuVitor [60] . Figure 8.4 depicts the multi-view Henshin editor whose presentation of rules in the upper center reminds of their original formal definition, i.e., the LHS and RHS are separate models (or graphs from the formal point of view). Mappings, i.e., the declaration of identities, are encoded by equal colors and numbers. This is considerably different to the diagram editor shown in Fig. 8.3 where identic elements of LHS and RHS are visualized as truly one element.

Straight left to the LHS and RHS, a bird's eye view of an application condition (AC) is given. Although looking like a model pattern, it shows the nesting of ACs. Nevertheless, ACs itself are model patterns analogously to the one given in the LHS and RHS. In the present case, the AC `isAbstract` is nested by a `NOT` operator which means that the pattern defined in `isAbstract` must not be found in order to apply the rule. Such condition is usually called negative application condition (NAC).

Henshin transformation units are means to apply rules in a structured manner and are supported by the multi-view editor as well. The lower area shows the visualization of a sequential unit which applies contained rules and units one after the other. It also illustrates the parameter `schema` to be passed

by the first rule. Henshin offers a number of nestable transformation units for different purposes, e.g., the so-called *Independent Unit* implements a non-deterministic rule/unit choice, *Conditional Units* requires an initial rule application as precondition for further applications, the *Priority Unit* applies rules/units according to their priority, and *Loop Units* allow to define how often rules/units shall be applied. Each unit has its own graphical representation in the editor.

In the leftmost position of Fig. 8.4, a tree-based view additionally supports the user with a concise outline of the transformation system. Note that this editor provides the basis of the implementation of the CompoHenshin editor explained later.

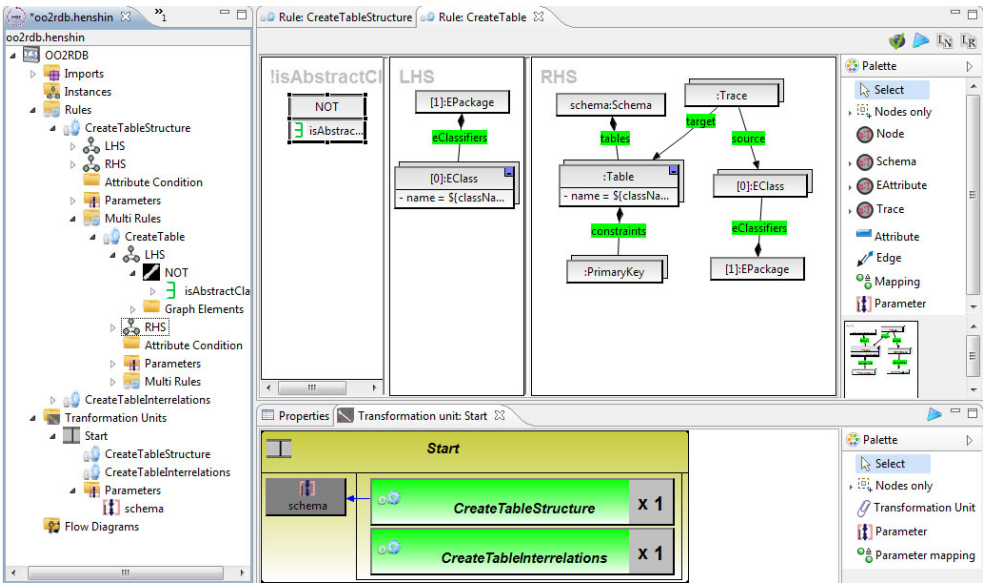


Figure 8.4: Externally developed editor for Henshin transformation systems.

8.2 CompoHenshin: A Composite EMF Model Transformation Tool

This section presents CompoHenshin [76], a Henshin [37] based transformation engine for CompoEMF models [73]. In the following, the requirements towards the implementation of CompoHenshin are discussed. Afterwards, the transformation language is introduced by defining its underlying meta model. The application of a CompoHenshin rule is described and illustrated thereafter. Finally, the tool support of CompoHenshin is highlighted.

8.2.1 Requirements

The main requirement is obviously the ability to transform composite EMF models based on CompoEMF. In the same way as CompoEMF conforms to the formalism of composite graphs, CompoHenshin shall conform to the formalism of typed composite IC-graph transformation presented in Chap. 6.2. This includes considerations about composite rule consistency which shall be reflected in CompoHenshin by means of a rule validation facility.

Choosing Henshin as underlying transformation engine, it is advisable to orientate at Henshin's tooling and go for high reuse as much as possible. This shall range from Henshin's transformation meta model to the reuse of actions and icons in Henshin editors. While Henshin provides a number of interesting advanced features such as control flow structures, the first version of CompoHenshin shall be limited to the basic transformation concepts defined in Part I.

As basic editor, a tree-based editor shall be available merely providing CRUD functionalities. Nevertheless, composite model structures tend to become complex very fast and therefore deserve a visual editor supporting a network view and the capability to conveniently navigate into related graphs on object layer.

8.2.2 CompoHenshin Meta Model

Figure 8.5 shows the CompoHenshin transformation meta model and in addition related Henshin kernel elements arranged in the center (cf. Sec. 7.1). The very fact that each CompoHenshin class inherits from a Henshin class already indicates the remarkable high reuse on Henshin. This is also apparent since CompoHenshin classes are not directly but only transitively associated with each other due to the inheritance from parent Henshin classes.

Analogously to Henshin's meta model, a composite model transformation system is constituted by a dedicated root element, `CMTransformationSystem`, containing composite model rules (`CMRule`). In accordance to the formalism, a `CMRule` represents a network rule and a number of local rules on object layer. A `CMRule` considered as a `Rule` is actually the network rule with an LHS and a RHS network graph (`NetworkGraph`) each carrying network nodes (`NetworkNode`) and network edges (`NetworkEdge`). Identical network nodes are specified over `Mapping`s analogously to original Henshin rules. The type reference of a `:NetworkNode` object is supposed to refer to the classes `Component`, `ExportInterface`, and `ImportInterface` of the *Composite* Ecore model only (see Fig. 7.7); analogously for `NetworkEdge`s. Thereby, the kind of each network node and network edge is explicitly defined and typing information can be directly used for a matching on Composite instance model (see below).

By means of the reference `objectRules`, `CMRule` contains other `Rule`s which are ordinary Henshin rules and specify the transformation on object level. Each

network node in a LHS or RHS is refined by a corresponding graph in the LHS or RHS of a local rule (see the Example 8.1 below). The relationship between a network node and its refining graph is defined by the reference `objectGraph`. The supposed meta model of that graph is predefined by the network node via its reference `domainModel` pointing to the EPackage of the targeted meta model. In compliance with the formalism, mappings between local models are maintained by related network edges, expressed by the containment reference `objectMappings`.

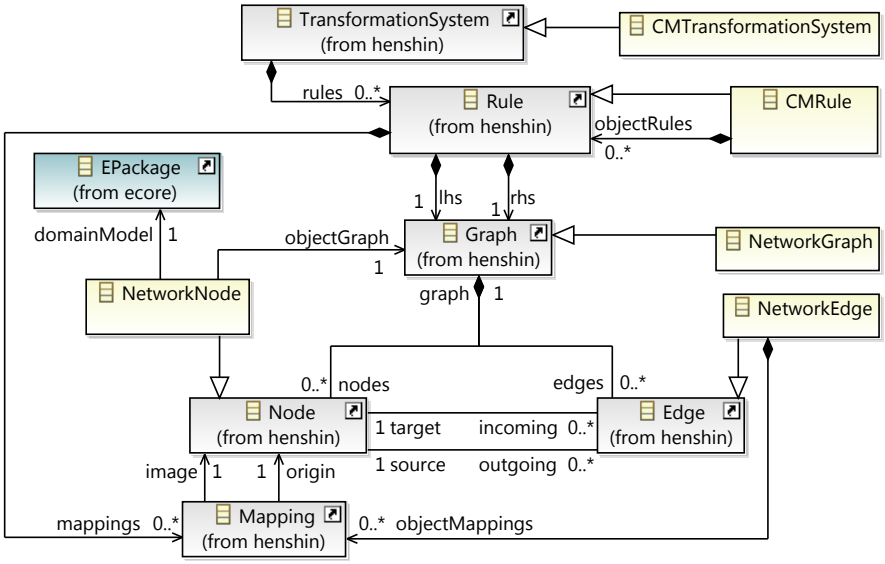


Figure 8.5: The CompoHenshin transformation meta model together with parts of Henshin's transformation meta model.

While the heavy reuse of Henshin's meta model simplifies the rule application as explained later, a disadvantage is that various meaningless instances of CompoHenshin can be created, e.g., local rules containing network nodes and network edges. To restrict possible instances, a number of OCL constraints are embedded into the CompoHenshin meta model. In fact, instances are not actively restricted by EMF (except by customized editors) but related checks automatically incorporated into the validation feature of EMF enable the user to conveniently ensure the correctness of specified composite rules. These constraints are not show here but are described in [75] and can be looked up in the model itself [73].

Example 8.1. In order to clarify the use of CompoHenshin's meta model, Fig. 8.6 shows the abstract syntax of a sample CompoHenshin rule. Attributes and typings as well as two `objectRules` links running from `:CMRule` to both `:Rule` objects are omitted in favor of clarity.

The root object `:CMTransformationSystem` at the top of Fig. 8.6 comprises a single composite rule `:CMRule`, arranged below, whose network LHS contains two associated network nodes. It is assumed here that the left node represents an export node and the right one represents a body node while accordingly the network edge is an export-body edge. Comparing the RHS, only one network node is contained which is then identified as the body node by the `Mapping` contained in `CMRule`. An application of that network rule would consequently lead to a removal of an export interface.

Taking identities into account, two network nodes exist. Each of them is refined by the LHS and/or RHS graphs of their local rules. Considering the mapped body network nodes in the LHS and RHS, their refinements on object level are the LHS and RHS graphs of the local rule located in the right center of Fig. 8.6. Their structure looks incidentally similar to the network rule and would lead to the deletion of a local node and an edge. The export network node in the LHS is refined by an LHS graph of another rule arranged at the bottom of Fig. 8.6. This local export graph contains only one export node whose corresponding body node is identified by the given `:Mapping` object. Since the export network node appears in the LHS only, i.e., the node is to be deleted, the local RHS graph is empty and not associated by a network node.

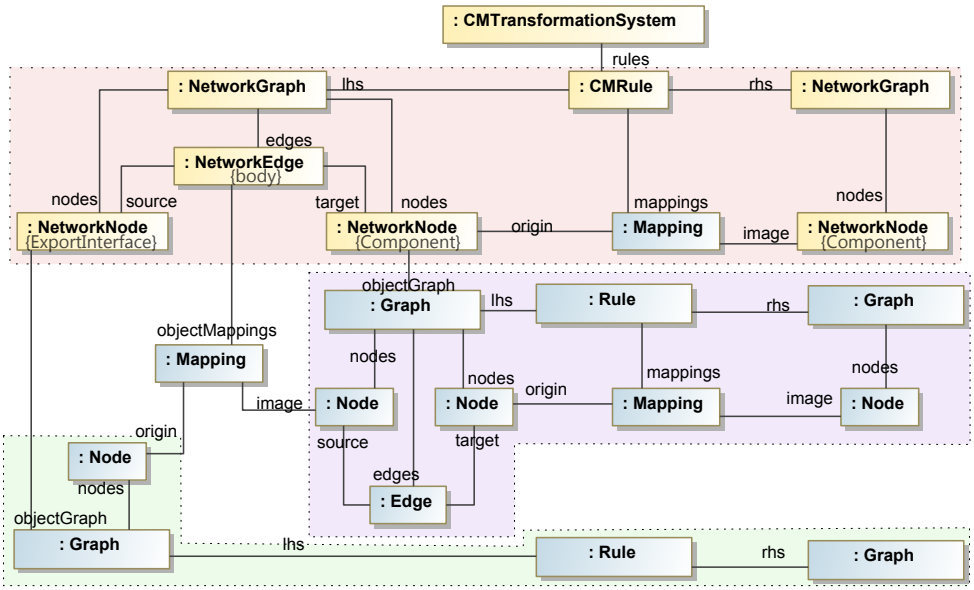


Figure 8.6: Sample instance of CompoHenshin's transformation meta model in abstract syntax.

Figure 8.7 shows the given example in concrete syntax. The network rule is given in the upper area and reveals that the export node (E) and its edge to

the body (B) shall be deleted. Note that the body node in the LHS and RHS is identic. Below, the two corresponding rules for the refinments of body and export are given. Equal numbers mean equal nodes. Note here in particular that numbers declare identities beyond rule borders and that the RHS of the export rule is empty which is due to the deletion of the export.

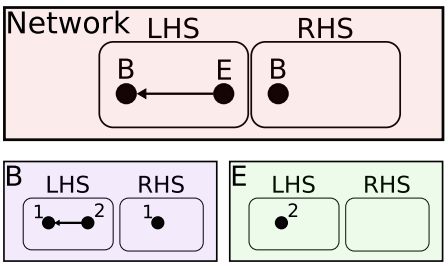


Figure 8.7: Sample concrete syntax of Figure 8.6.

△

8.2.3 Composite Rule Application

Aiming at lifting Henshin's functionality to composite modeling, the composite rule application API (not shown here) closely resembles the one provided by Henshin. However, obviously the transformation procedures of Henshin and CompoHenshin differ considerably since composite rules in contrast to ordinary rules consist of a number of rules to be matched and to be transformed. In the following, the transformation procedure is outlined. See [76] for a more detailed description.

Matching. Figure 8.8 illustrates the matching algorithm of a CompoHenshin rule in a simplified form. First of all, the algorithm determines a match on network level. This includes related composite gluing condition checks. If a network match can be found, the algorithm determines a set of matches on object layer which is consistent with the network match. That means, the match finding on object layer takes the network match and the refinements of all network nodes and edges into account. If such a set can be found, the composite gluing conditions for the object layer are checked to ensure the match to be valid. If no such set can be found or if the potential match turned out to violate the composite gluing condition, a different network match is tried to be determined and so on. If no network match is left, the algorithm finishes.

The network matching is actually performed by means of an ordinary matching from the CompoHenshin's network rules into a *Composite* instance model without considering refinements on object layer yet. The direct application of the network rule is possible since the network rule is a real Henshin

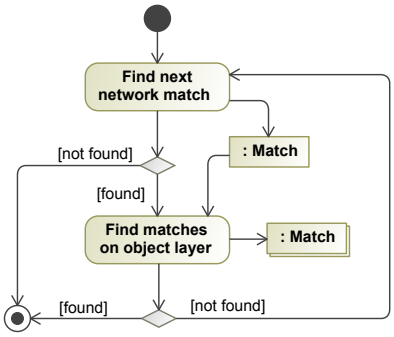


Figure 8.8: Simplified process of composite matching in CompoHenshin.

rule and all nodes in the LHS by definition refer to elements of the `Composite` Ecore model via the `type` reference (see Sec. 7.2.2). So far, this allows to select those nodes in the *Composite* instance which are of the same kind (body, export, import). What remains is to filter them according to their domain model referred to by the `:NetworkNodes` reference `domainModel` (cf. Fig. 8.5). Since there are no direct correspondences to match, this is achieved by transparently equipping `:NetworkNode` objects with an additional `:Attribute` which is typed over the attribute `Component.ePackageUri` (cf. Fig. 7.7). The value of this attribute is then set to the `nsUri` of the assigned domain model.

Example 8.2 (Network matching)**.** Figure 8.9 illustrates a simple setting of network matching by means of models in abstract syntax. In the upper center, the network LHS of a CompoHenshin rule is given which is matched to the network of a host model located below. The rule as well as the host model consist of only one component part for clarity. In detail, the host model consists of the node `:Component` being the network representative of the department management body constituted by the object `:Department`. At the left, a subset of the *Composite* Ecore model is given and at the right, the domain model of the host model's body is shown which shall be the department management model again.

The `:NetworkNode` in the LHS refers to the class `Component` and therefore declares to match a body component part. The targeted body is defined more closely by the reference `domainModel` which points to the `:EPackage` of the department management Ecore model. An `:Attribute` representing `ePackageUri` attributes is added automatically during runtime and preset by the value of the associated `:EPackage`, "http://depmgt". `:NetworkNode` and `:Attribute` together are now sufficient to match the desired structure. △

Rule application. The activity diagram at the left of Fig. 8.10 outlines the application of the object rule with respect to the matches found beforehand. Analogously to composite matching explained above, the application initially

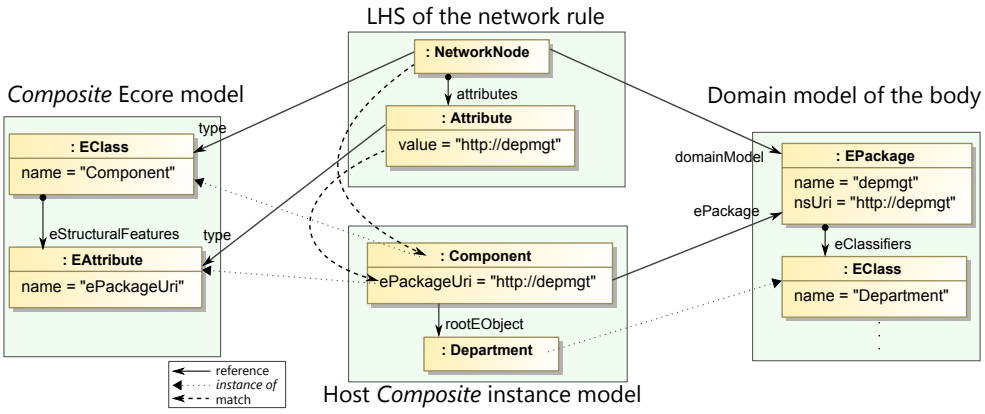


Figure 8.9: Matching from a network rule's LHS into the host *Composite* instance model.

takes place on the network layer using the network match and is performed by Henshin. In the background, for each newly created network node an empty graph is created immediately which serves as host for the corresponding object rule later.

The transformation of the object layer requires a more sophisticated approach due to the references between objects of different graphs, e.g., objects in interfaces refer to objects in bodies. Obviously, a certain application order is meaningful to prevent cases where new objects shall point to others which have not been created yet. The basic procedure is to transform those object graphs first whose network nodes are preserved, afterwards those that have been created and finally the deleted ones. In particular, each time the network kind order *body* $\rightarrow$ *export* $\rightarrow$ *import* is adhered, i.e., at first all preserved body graphs are transformed and then all preserved export graphs and preserved import graphs.

Figure 8.10 especially illustrates the steps concerning preserved network nodes. Note the two complex activities in the diagram on the left with the small tree structures in their lower right corner; both complex activities are refined by activity diagrams themselves depicted in the right of Fig. 8.10. When the network rule has been applied on a match, the graphs of all preserved body graphs are successively selected and their corresponding rules are applied with respect to their match. Afterwards, all graphs of preserved export nodes are selected. The application of their rules comprises each an ordinary graph transformation and in addition the update of references towards the body. Similarly, the subsequent application of graphs of preserved import nodes is performed while the additional task contains the updating of references towards the body graph and towards the related export graph.

When all preserved network nodes are processed, a similar procedure starts

for all newly create nodes and then for all deleted nodes.

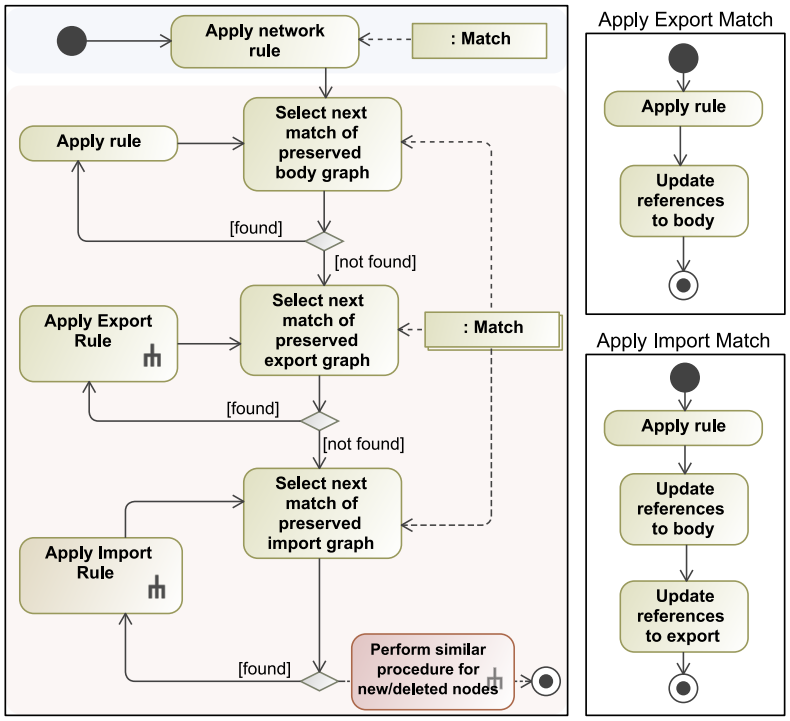


Figure 8.10: Process of applying a CompoHenshin rule.

8.2.4 Tool Environment

Since composite transformation systems containing composite rules are ordinary instances over the CompoHenshin Ecore model, the natural choice for a first editor support is the editor that can be fully generated by EMF. Such a tree-based editor is available [76] providing basic CRUD functionality and validation. Synergies arise when existing editor customizations of the original Henshin editor are reused, e.g., reusing custom commands for local rules such as the deletion command which deletes not only `Node`s but all associated `Mapping`s and `Edge`s as well.

Example 8.3. Fig. 8.11 shows an example CompoHenshin rule in its tree-based editor representation. Conforming to the meta model shown in Fig. 8.5, this rule consists of a network rule and two object rules as refinements. The rule name and the network LHS graph reveal that an export interface shall be deleted. The lower area shows the properties of the selected body node. They indicate that the refining model conforms to the department management Ecore model and that the graph *LHS2* is the left-hand side of the belonging rule on object layer. △

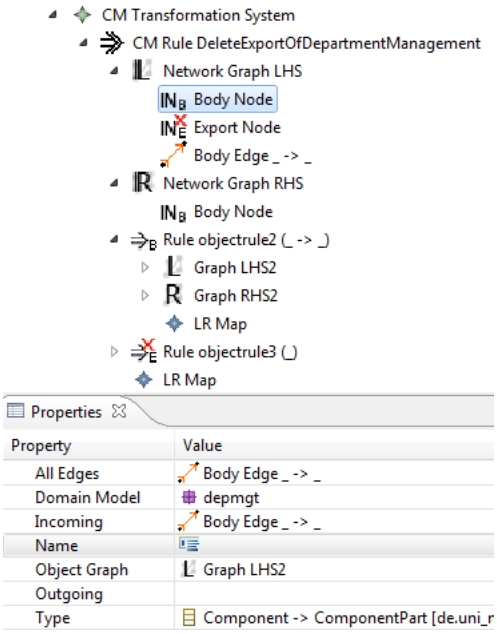


Figure 8.11: Sample CompoHenshin rule shown in the tree-based editor.

Looking at this small example it is already evident that the tree-based editor reaches its limitations very fast with regard to the presentation of graph structures. A more promising and particularly natural editor for graph-like structures would be a graphical one. To this end, the CompoHenshin tooling includes a graphical editor with multiple views on a CompoHenshin rule which is based on GEF analogously to the Henshin editor of the TFS group (see Sec. 8.1). In fact, the sources of that editor serve as a good starting point for the development of the more advances graphical editor for CompoHenshin rules.

Example 8.4 (Visual editor for CompoHenshin transformation systems)**.** Figures 8.12 - 8.14 present different views on the rule already shown in a tree-based manner now by means of the graphical editor for CompoHenshin transformation systems.

Figure 8.12 gives an overview of the whole editor and shows the network layer of the rule.

Analogously to the TFS Henshin editor, a tree-based outline of the transformation system is given on the left which includes imported meta models and a list of available rules. Note that *to import meta models* in this context means that the editor is acquainted with these meta models and their classes and references. Only then, when creating rules on object layer those classes and references are available to be selected as node types or edge types. At the very right, a tool palette is given which allows to place nodes on the content

pane, to connect them and to map nodes of LHS and RHS. The network graph arranged in the center revisits the previous example and declares the export interface of type `DepExport2` of a department management component to be deleted.

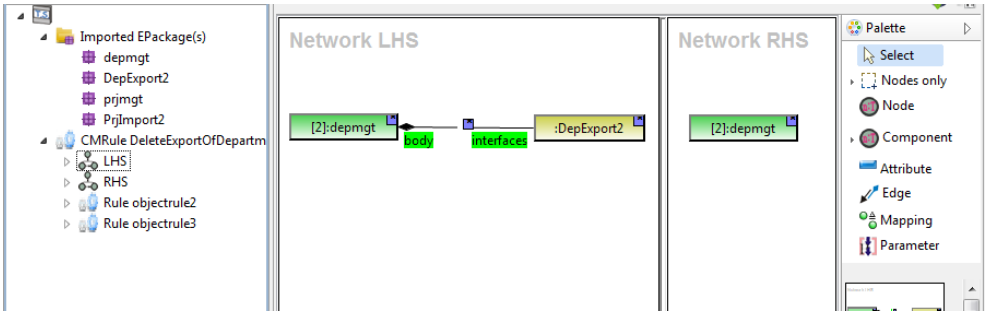


Figure 8.12: Network layer of a sample CompoHenshin rule shown in the CompoHenshin visual editor.

A rule on object layer is presented in Fig. 8.13. In detail, the rule representing the export interface is shown. Since the interface is to be deleted, the RHS of the rule has to be empty. Nevertheless, the LHS indicates that the LHS of the body object rule at least consists of three equivalent nodes which are exported so far.

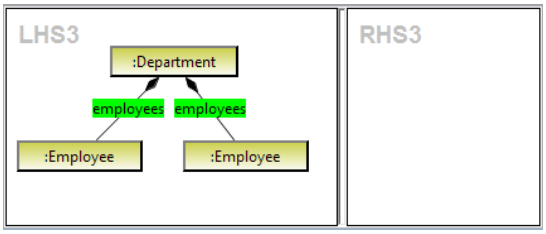


Figure 8.13: Export node (cf. Fig. 8.12) on object level being a local rule.

The refinement of network edges is also visualized in form of an aggregated model which contains all elements of the edge's source and target model. Figure 8.14 illustrates the refinement of the export-body network edge between the `:DepExport2` export interface and the `:depmgt` department management body as shown in Fig. 8.12. The three nodes in the upper area belong to the export model while the four nodes below belong to the body model. This is also denoted by colors corresponding to the color in the network view. Edges between nodes of different models constitute the mappings which are the actual refinement of the network edge. Obviously, aggregated models may yield huge models very fast such that the handling in this view may become uncomfortable. Generally, since composite models and composite rule are complex

structures it is up to future work to find more adequate visualizations and tools to handle them in a convenient way.

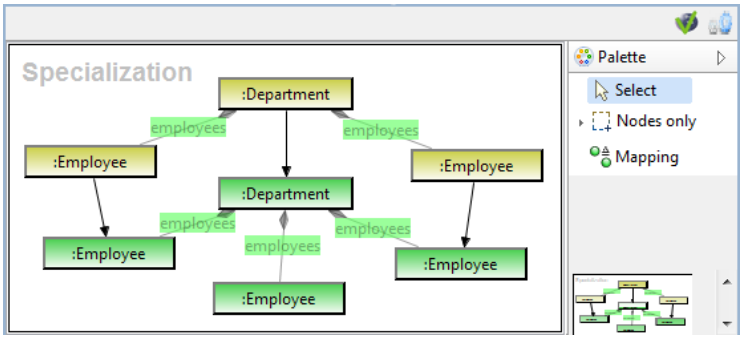


Figure 8.14: Mappings graph as object layer of the body-export edge in Fig. 8.12.

△

Chapter 9

Related Work

In the following, the approach presented in this thesis is compared with other concepts for the distribution and composition of models and graphs. Furthermore, a generic component framework for system modeling is considered as well.

Section 2.1 already presents a number of selected approaches along the given classification which essentially relates to structural aspects. In contrast, the considerations below compare the present approach of composite modeling with explicit import and export interfaces from a more general perspective.

9.1 EMF Models with Remote References

EMF already offers the possibility to spread models over a number of so-called resources, e.g., files (compare [27]), exploiting so-called remote references. Such a remote reference is an ordinary association whose source is an object node in the local model while its target is an object node in a remote model. Since each object node can be referred to by remote references, no information hiding is taking place here. References solely restrict the type of the target object node while any reference may be utilized in a remote manner. In accordance to composite models with explicit interfaces, EMF models with remote references can be considered as body models with interfaces being defined implicitly. Target nodes of remote references induce the implicit import while all model elements are exported implicitly. The imported element is identical to the corresponding exported element.[1] A single imported model element can be exploited to reveal connected elements of the remote model. Consequently, several physically distributed EMF models can be considered as a single undistributed model from a logical perspective.

[1] In fact, proxy objects represent remotely targeted nodes until they are accessed for the first time. In that case a proxy is resolved and replaced by the original node.

The present approach differs from the one in [27] in a number of aspects. In particular, composite models with explicit interfaces are a component-oriented distribution of models. Being still on meta level, the declaration of body, export, and import component types and their interconnections already predefine possible distribution shapes on instance level. Node types, edge types as well as mappings between them along different component types predefine permitted objects, links, and relationships. Nevertheless, only objects and links being actually part of an export interface are visible to other components. Vice versa, only explicitly imported objects and links, i.e., such being part of an import interface, enable the access of entities of foreign components.

This explicit structuring of models is not trivial and requires careful consideration by the engineers. As a side effect, it requires a structured design process to be followed by the engineers. In contrast to EMF models with remote references, the present approach considers elements in interface models to be some kind of delegate objects. In particular, they do not need to have the same type as the targeted one and are especially not the same objects on instance level. Defined features such as containment, references, and attributes (not formally described but included in CompoEMF) just need to be mapped to corresponding features of the target. Accessing such an imported feature leads to a delegation to the corresponding exported feature. Delegate objects have further advantages: In addition to imported features, they may have a number of own features. As a consequence, a composite model is not considered as a single huge model but as a composition of individual component models. Consequently, EMF-typical constraints such as acyclic containments must hold in each component of a composite graph, but might not hold if they are merged into one big graph (which is usually not done in the composite modeling concept proposed in this thesis).

9.2 Model composition approaches

In [16], the composition of separate design models is specified by composition relationships identifying overlapping elements of different design models and specifying their integration. Thus, explicit interfaces supporting information hiding are not present. Estublier and Ionita [28] distinguish two types of model compositions: (1) Composing different views of the same system and (2) composing the same view of different systems. In general, the approach in this thesis allows different views of different systems and thus, a generalization of case (2). Case (2) is treated by a merge-by-name strategy as in [16] and thus does not provide an information hiding concept though.

In [36], Heidenreich et al. propose an approach to extend a meta model inversely in order to compose its instance models. This merging is done along predefined interface-like structures. In contrast, the present approach describes a component concept for models consisting of separate and especially individ-

ual components with possibly different meta models sharing explicitly revealed information only.

A model composition technique has been introduced by Kelsen and Ma in [50] where meta models are extended by explicit denotations of elements as glue for a composition created later. Instances of such meta models achieve the composition by referring to other models. Referred elements need to be exposed explicitly by the target model in a manner similar to an export interface. This enables information hiding. The authors consider a single meta model only, while the present approach allows a number of meta models. Furthermore, interfaces are essentially subsets of the original meta model. The approach in this thesis, however, is more flexible by providing interfaces being possibly simpler than the structure of the original meta model with respect to inheritance.

It is common to all model composition approaches that transformations of composite models are not considered.

9.3 Graph-Oriented Approaches

In [67], an approach to distributed graphs and graph transformations is presented which allows the distribution of graph parts, but does not support explicit interfaces. In this sense, the distribution concepts of that approach and of EMF models with remote references are quite similar. In [59], Mezei et al. present distributed model transformations based on graph transformation concepts. Model transformations are not distributed logically, but in the sense that they are performed in a distributed way in order to increase efficiency. This means that transformations are distributed automatically. Again, interfaces are handled implicitly. View-oriented modeling has already been specified by distributed graph transformation in [35] using the same formal basis as composite graphs. However, the structuring of graphs is elaborated more deeply in this paper, since components with explicit interfaces are considered.

Knirsch et al. present an approach of modeling distributed graph transformation systems by transformation units in [52]. While the approach in this thesis considers the application of single distributed/composite rules only, distributed graph transformation units essentially consist of a set of distributed rules which are, in addition, applied by taking a control condition into account. Such units and their rules are structured similarly to composite rules in this paper, i.e., they consist of a set of local transformation units which are connected by interface units. However, their underlying distribution structure does not distinguish import and export interfaces but single boundary graphs only, representing the intersection of two interconnected graph.

Hierarchical graphs as presented in [21] are hypergraphs where distinguished hyperedges contain graphs that can be hierarchical again. Components with interfaces are not explicitly considered but can be defined as a subclass of hierarchical graphs with hierarchy depth 2. Moreover, the upper layer would contain network graph structures only. Drewes, Hoffmann, and Plump do not only present hierarchical graphs but also their transformation following the algebraic approach. However, typing of graphs including node type inheritance as well as containment structures are not considered in that approach what makes it difficult to use as formal basis for composite EMF modeling. In [15], an algebraic view on hierarchical graphs is established; however, the above mentioned restrictions are also valid in that approach.

9.4 Generic Approaches towards Composite Modeling

The approach of mega modeling focuses on model interrelations which may be of arbitrary kind, e.g., model transformations, integration rules, and typing relations between typed models and their meta models. A number of more concrete settings for mega modeling are discussed in [41] and [8]. For instance, in [41] relations between meta models of different modeling languages are elaborated. Tool support concentrates on model integration (AMW [2]) and model based reverse engineering (MoDisco [61]). So far, there have been no considerations according to component models with explicit interfaces and reasoning according to potential inconsistencies between related models.

Another generic approach is called macro modeling [71] which comprises a formal framework able to interconnect models of arbitrary domain and language. The use of specialized model relationships then yields a new type of model, called macro model, with a formal semantics. Macro modeling aims at convenient handling of multiple models and a related consistency management. The consistency management is implemented in form of global consistency properties [69] which are formulated and checked by means of logic theory [70]. Tools are still in a prototypical state and are not integrated in popular environments like Eclipse. Furthermore, (macro) model transformation is not yet considered.

In [25], a generic component framework for system modeling is presented. It abstracts from concrete modeling techniques but establishes constraints describing a compositional semantics for components. Components have explicit import and export interfaces to provide information hiding. However, the numbers of interfaces are restricted to exactly one import and one export interface while composite models as presented in this thesis allow arbitrary many interfaces. Furthermore, in [25] the composition of components to new ones is presented. Transformation concepts of composite structures are not yet considered though. The framework is assembled on a number of properties which

have to be shown for each instantiating modeling technique in order to apply the framework. Several instantiations are considered such as Petri nets, graph transformation systems, and visual modeling techniques. However, typing of graphs including node type inheritance as well as containment structures are not considered as instantiation yet.

Chapter 10

Conclusion

This chapter summarizes the main results of this thesis first and closes with an outlook to future work.

10.1 Summary

Model-driven development is a promising paradigm to tackle the development of nowadays complex software systems. Models are an ideal means for abstraction allowing developers to focus on certain aspects of the system. Details are added by (code) generators later. Nevertheless, steadily growing complexity may lead to large models. Moreover, various models representing different views on a software system need to be kept consistent to each other somehow.

To meet these challenges, this thesis introduces a model composition concept, called composite models with explicit import and export interfaces. Composite models consists of a set of component models where each is constituted by a body model and an arbitrary number of interface models. Component models can interrelate in a structured way by connecting their export and import interfaces. This also offers information hiding because shared elements have to be declared explicitly. Furthermore, possible composition shapes can already be defined on meta model level. Composite model transformation enables the preservation of consistency between each component model by allowing to modify several ones synchronously. This concept of composite modeling essentially targets models based on the popular Eclipse Modeling Framework yielding composite EMF modeling.

A formal foundation is provided which relies on category theory and the theory of graph transformation. Typed graphs with inheritance and containment structures, short typed IC-graphs, describe the essential structures of EMF models while graph transformation described model transformation formally. Formal distribution concepts are based on distributed graphs and distributed graph transformation. Combining them altogether yields the category

of typed composite IC-graphs, CompICGraphs$_{TG}$, and a related composite IC-graph transformation approach based on double pushouts. It has further been clarified under which conditions pushouts in this category exist, i.e., when does a transformation leads to typed component IC-graphs with proper containment structures. Furthermore, a very important result is that composite transformation can be led back to ordinary graph transformation. This fact is exploited later by the tool support. On top of composite graphs, the concept of weak composite graphs further facilitates the usefulness of component models in practice at which foreign components truly expose their export interface only.

The formalized concepts are implemented by the prototype tools CompoEMF and CompoHenshin, both providing dedicated tree-based editors and even more sophisticated visual editors. They represent a proof-of-concept implementation and show the coherence of the composite EMF modeling concepts. CompoEMF allows ordinary EMF models to become the bodies of component models in a noninvasive way, i.e., without the need to modify them. This promotes high reuse. Interfaces are stored in separate EMF models which transparently propagate attribute values of shared objects along the interface chain. CompoHenshin provides the transformation of CompoEMF models and builds up on the EMF model transformation tool Henshin. Each part (body, export, import) of component models is transformed by a single Henshin transformation being coordinated with each other.

10.2 Outlook

Even with an elaborated composition concept and tooling at hand, a couple of challenges arise in a distributed and model-driven setting. For instance, contributors at different locations might be responsible for component models being interconnected with others. Thus, clear conditions and conventions for the editing of models are required to avoid conflicts. Furthermore, the starting point of distributed development might differ, e.g., a big model may already exist which has to be split up somehow. When going to generate code out of a set of distributed models, one (and presently the common) possibility is to merge all models and using the resulting large model as the generator's input. Other strategies are conceivable as well as providing a distributed code generator. To conclude, a comprehensive elaboration on a development process for composite modeling in distributed model-driven software development is needed. This in general includes substantiation of scenarios for distributed model-driven development.

The formalism given in this thesis captures the essentials of EMF models. However, a formal specification of attributes shall be integrated as done in [23] since the tools CompoEMF and CompoHenshin exploit attributes already. Some other aspects of EMF models have not been considered yet as

well and need to be caught up, e.g., bidirectional edges and multiplicities. As for composite IC-graph transformation, the formalism shall be extended to exploit polymorphisms in the sense that objects in rules may match objects with the same type and also objects with a type being a child of the required type. Formalisms particularly enable the development of analysis which should be exploited in the future. Conceivable analysis are such concerning critical pairs and the termination of composite rules.

So far, the given implementations towards composite EMF modeling are of prototypical nature which leaves the door open for a number of improvements. CompoEMF may simplify the specification of composite EMF models with an improved editor support. In accordance with the formalism, CompoEMF shall also support weak composite models. This does not mean a capability to create composite models with export interfaces only but to provide a means to physically hide component model parts except of its export interface. The editors of CompoHenshin may also get improved in a number of ways. A surely meaningful improvement would be some kind of debugging functionality to help the modeler to understand what is going on. Furthermore, CompoHenshin shall support the transformation of weak composite models which is assumed to be rather easy to implement as soon as CompoEMF supports weak composite models. Other conceivable improvements of CompoHenshin are related to the adaption of features already offered by Henshin such as application conditions which further limit matches and transformation units to specify control flows.

Aside from the tool implementations in this thesis, there is room for other ways and technologies to implement the concepts introduced here. Conceivable are implementations which utilize service-oriented architectures, e.g., such that model interfaces are provided by web-services on a system and composite model transformations are performed truly distributed on each systems independently. However, the probably most important next step is the elaboration on a practical model-driven development process in a distributed environment.

Bibliography

[1] AGG. *Attributed Graph Grammar System.* `http://tfs.cs.tu-berlin.de/agg`.

[2] AMW. Atlas Model Weaver. `http://www.eclipse.org/gmt/amw/`, 2012.

[3] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformation. In *Proc. of 13th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*, LNCS. Springer, 2010.

[4] AspectJ. A general-purpose aspect-oriented extension to Java. `http://www.eclipse.org/aspectj/`, 2011.

[5] ATL. ATL Transformation Language. `http://www.eclipse.org/atl/`, 2012.

[6] B. Baudry, F. Fleurey, R. B. France, and R. Reddy. Exploring the Relationship between Model Composition and Model Transformation. In *In Proc. of Aspect Oriented Modeling Workshop, in conjunction with MoDELS'05*, 2005.

[7] J. Bézivin, S. Bouzitouna, M. D. D. Fabro, M.-P. Gervais, F. Jouault, D. S. Kolovos, I. Kurtev, and R. F. Paige. A Canonical Scheme for Model Composition. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Second European Conference*, volume 4066 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2006.

[8] J. Bezivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. `https://gforge.inria.fr/scm/viewvc.php/*checkout*/`

`Publications/Before2009/bezivin-megamodel.pdf?rev=12&root=atlantic-zoos`, 2004. In Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.

[9] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical Definition of Rule-Based Transformation in the Eclipse Modeling Framework. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *9th Int. Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 425 – 439. Springer, 2006.

[10] E. Biermann, C. Ermel, and S. Jurack. Modeling the "Ecore to GenModel" Transformation with EMF Henshin. In *Proceedings of TOOLS 2010 Federated Conferences*, 2010. Workshop "Transformation Tool Contest 2010".

[11] E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Proc. of 11th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *LNCS*, pages 53–67. Springer, 2008.

[12] E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and System Modeling*, 11(2):227–250, 2012.

[13] A. Boronat. *MOMENT: A Formal Framework for Model Management.* PhD thesis, Universitat Politècnica de València, 2007.

[14] B. Braatz, H. Ehrig, K. Gabriel, and U. Golas. Finitary $\mathcal{M}$-adhesive categories. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr, editors, *5th Int. Conference on Graph Transformations, ICGT, Enschede*, volume 6372 of *Lecture Notes in Computer Science*, pages 234–249. Springer, 2010.

[15] R. Bruni, F. Gadducci, and A. Lluch-Lafuente. An algebra of hierarchical graphs. In *Trustworthly Global Computing - 5th International Symposium, TGC 2010, Munich, Germany, February 24-26, 2010, Revised Selected Papers*, volume 6084 of *Lecture Notes in Computer Science*, pages 205–221. Springer, 2010.

[16] S. Clarke. Extending standard uml with model composition semantics. *Sci. Comput. Program.*, 44(1):71–100, 2002.

[17] C. Clasen and H. Bruneliere. Virtual EMF. `http://code.google.com/a/eclipselabs.org/p/virtual-emf/`, 2012.

[18] C. Clasen, F. Jouault, and J. Cabot. VirtualEMF: A Model Virtualization Tool. In O. D. Troyer, C. B. Medeiros, R. Billen, P. Hallot, A. Simitsis, and H. V. Mingroot, editors, *Advances in Conceptual Modeling. Recent Developments and New Directions - ER 2011 Workshops*, volume 6999 of *Lecture Notes in Computer Science*, pages 332–335. Springer, 2011.

[19] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

[20] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, 2007.

[21] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002.

[22] I. Eclipse Foundation. Eclipse - The Eclipse Foundation open source community website. `http://www.eclipse.org`, 2012.

[23] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[24] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. From graph grammars to high level replacement systems. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science, 4th International Workshop*, volume 532 of *Lecture Notes in Computer Science*, pages 269–291. Springer, 1991.

[25] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A generic component framework for system modeling. In *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *LNCS*, pages 33–48. Springer, 2002.

[26] H. Ehrig, F. Orejas, and U. Prange. Categorical Foundations of Distributed Graph Transformation. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2006.

[27] EMF. Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/emf`, 2012.

[28] J. Estublier and A. D. Ionita. Extending uml for model composition. In *16th Australian Software Engineering Conference (ASWEC 2005), 31 March - 1 April 2005, Brisbane, Australia*, pages 31–38. IEEE Computer Society, 2005.

[29] F. Fleurey, B. Baudry, R. B. France, and S. Ghosh. A Generic Approach for Automatic Model Composition. In H. Giese, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Reports and Revised Selected Papers*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer, 2007.

[30] F. Fleurey, R. France, B. Baudry, and M. Clavreul. Kompose. `http://www.kermeta.org/kompose/`, 2012.

[31] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[32] GEF. Graphical Editing Framework (GEF). `http://www.eclipse.org/gef`, 2012.

[33] S. Gêrard. Papyrus UML. `http://www.papyrusuml.org/`, 2012.

[34] GMF. Graphical Modeling Framework. `http://www.eclipse.com/gmf`.

[35] M. Goedicke, T. Meyer, and G. Taentzer. ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *Proc. 4th IEEE Int. Symposium on Requirements Engineering (RE'99), June 7-11, 1999, University of Limerick, Ireland*. IEEE Computer Society, 1999. ISBN 0-7695-0188-5.

[36] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On Language-Independent Model Modularisation. *T. Aspect-Oriented Software Development VI*, pages 39–82, 2009.

[37] Henshin. `http://www.eclipse.org/modeling/emft/henshin`, 2012.

[38] F. Hermann, H. Ehrig, and C. Ermel. Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In M. Wirsing and M. Chechik, editors, *Proc. International Conference on Fundamental Aspects of Software Engineering (FASE'09)*, volume 5503 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 2009.

[39] F. Hermann, H. Ehrig, and C. Ermel. Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks (Long Version). Technical report, Technische Universität Berlin, 2010.

[40] F. Jouault and I. Kurtev. Transforming Models with ATL. In J.-M. Bruel, editor, Satellite Events at the MoDels 2005 Conference, volume 3844 of *LNCS*. Springer, 2005.

[41] F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, and J. Bézivin. Inter-dsl coordination support by combining megamodeling and model weaving. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2011–2018. ACM, 2010.

[42] S. Jurack. Composite EMF Modeling based on Typed Graphs with Inheritance and Containment Structures. In A. Rensink, editor, *Proc. of 5th Int. Conference on Graph Transformation (ICGT 2010)*, LNCS. Springer, 2010. Doctoral Symposium.

[43] S. Jurack and F. Mantz. Metamodel Evolution of EMF Models with Henshin. In *Proc. of 13th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*, 2010. Int. Workshop on Models and Evolution (ME2010).

[44] S. Jurack and G. Taentzer. Towards Composite Model Transformations Using Distributed Graph Transformation Concepts. In A. Schürr and B. Selic, editors, *Proc. of 12th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS 2009)*, volume 5795 of *LNCS*, pages 226–240. Springer, 2009.

[45] S. Jurack and G. Taentzer. A Component Concept for Typed Graphs with Inheritance and Containment Structures. In A. Rensink, editor, *Proc. of 5th Int. Conference on Graph Transformation (ICGT 2010), Enschede*, LNCS, pages 187–202. Springer, 2010.

[46] S. Jurack and G. Taentzer. A Component Concept for Typed Graphs with Inheritance and Containment Structures: Long Version. Technical report, Philipps-Universität Marburg, 2010.

[47] S. Jurack and G. Taentzer. Transformation of typed composite graphs with inheritance and containment structures. *Fundamenta Informaticae*, 118(1-2), 2012.

[48] S. Jurack and J. Tietje. Saying Hello World with Henshin - A Solution to the TTC 2011 Instructive Case. In P. V. Gorp, S. Mazanek, and L. Rose, editors, *Proc. Fifth Transformation Tool Contest (TTC)*, volume 74 of *EPTCS*, pages 253–280, 2011.

[49] S. Jurack and J. Tietje. Saying Hello World with Henshin - A Solution to the TTC 2011 Instructive Case. In P. V. Gorp, S. Mazanek, and L. Rose, editors, *Proc. Fifth Transformation Tool Contest (TTC)*, volume 74 of *EPTCS*, pages 253–280, 2011.

[50] P. Kelsen and Q. Ma. A Modular Model Composition Technique. In *Proc. of 13th Int. Conference on Fundamental Approaches to Software Engineering, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, 2010*, volume 6013 of *LNCS*, pages 173–187. Springer, 2010.

[51] Kermeta. Kermeta. `http://www.kermeta.org/`, 2012.

[52] P. Knirsch and S. Kuske. Distributed Graph Transformation Units. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Transformation, 1st Int. Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *LNCS*. Springer, 2002.

[53] C. Köhler, H. Lewin, and G. Taentzer. Ensuring containment constraints in graph-based model transformation approaches. *ECEASST*, 6, 2007.

[54] D. Kolovos. EuGENia. `http://www.eclipse.org/epsilon/doc/eugenia/`.

[55] D. S. Kolovos, R. F. Paige, F. Polack, and L. M. Rose. Update transformations in the small with the Epsilon Wizard Language. *Journal of Obj. Tech.*, 6(9):53–69, 2007.

[56] S. Lack and P. Sobocinski. Adhesive categories. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures (FoSSaCS), 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2004.

[57] M. T. T. P. (M2T). Java Emitter Templates (JET). `http://www.eclipse.org/modeling/m2t/?project=jet#jet`, 2012.

[58] N. Magic. Magic Draw. `http://www.nomagic.com/products/magicdraw.html`, 2012.

[59] G. Mezei, S. Juhasz, and T. Levendovsky. A distribution technique for graph rewriting and model transformation systems. In H. Burkhart, editor, *Proc. of the IASTED Int. Conference on Parallel and Distributed Computing Networks*, pages 63–68. IASTED/ACTA Press, 2007.

[60] T. Modica, E. Biermann, and C. Ermel. An eclipse framework for rapid development of rich-featured gef editors based on emf models. In *GI Jahrestagung*, volume 154 of *LNI*. GI, 2009.

[61] MoDisco. http://eclipse.org/MoDisco.

[62] Object Management Group. The Essential MOF (EMOF) Model. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01.pdf`, 2010. Sec. 12.

[63] Object Management Group. Meta Object Facility Core (MOF) Specification. `http://www.omg.org/spec/MOF/`, 2012.

[64] Object Management Group. MOF 2 XMI Mapping (XMI) Specification. `http://www.omg.org/spec/XMI/`, 2012.

[65] OMG. UML Resource Page of the Object Management Group. `http://www.uml.org/`.

[66] B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[67] U. Ranger and M. Lüstraeten. Search Trees for Distributed Graph Transformation Systems. *Electronic Communication of the EASST*, 4, 2006.

[68] D. D. Ruscio, R. Lämmel, and A. Pierantonio. Automated co-evolution of gmf editor models. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, volume 6563 of *LNCS*, pages 143–162. Springer, 2010.

[69] M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik. Global consistency checking of distributed models with tremer+. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 815–818. ACM, 2008.

[70] M. Sabetzadeh, S. Nejati, S. Liaskos, S. M. Easterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*, pages 221–230. IEEE, 2007.

[71] R. Salay, J. Mylopoulos, and S. M. Easterbrook. Managing models through macromodeling. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 447–450. IEEE, 2008.

[72] T. Schäfer. Implementierung einer komponentenorientierten Architektur für Software-Modelle basierend auf dem Eclipse Modeling Framework (In German). Master's thesis, Philipps-Universität Marburg, 2012. `http://www.mathematik.uni-marburg.de/~swt/Publikationen_Taentzer/Students/TSchaefer_MA.pdf`. to appear.

[73] T. Schäfer and S. Jurack. Homepage of CompEMF. `http://www.mathematik.uni-marburg.de/~swt/compoemf/`, 2012.

[74] D. Steinberg, F. Budinsky, M. Patenostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison Wesley, 2008.

[75] D. Strüber. Konzeption und Implementierung von komponentenorientierten Modelltransformationen basierend auf der Eclipse-Technologie (In German). Master's thesis, Philipps-Universität Marburg, 2011. `http://www.mathematik.uni-marburg.de/~swt/Publikationen_Taentzer/Students/DStrueber_DA.pdf`.

[76] D. Strüber and S. Jurack. Homepage of CompHenshin. `http://www.mathematik.uni-marburg.de/~swt/compohenshin/`, 2012.

[77] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication Based Systems.* PhD thesis, Technical University of Berlin, 1996.

[78] G. Taentzer. Distributed Graphs and Graph Transformation. *Applied Categorical Structures*, 7(4), 1999.

[79] TFS. Department of Software Engineering and Theoretical Computer Science. `http://www.tfs.tu-berlin.de`, 2012.

[80] TFS Henshin. Visual Multi-View Henshin-Editor. `git://git@github.com:JuergenGall/Henshin-Editor.git`, 2012.

[81] J. Whittle and P. K. Jayaraman. MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. In H. Giese, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, volume 5002 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2007.

[82] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, and J. Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. *T. Aspect-Oriented Software Development VI*, 5560:191–237, 2009.

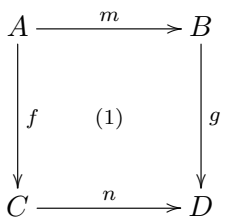
Appendix

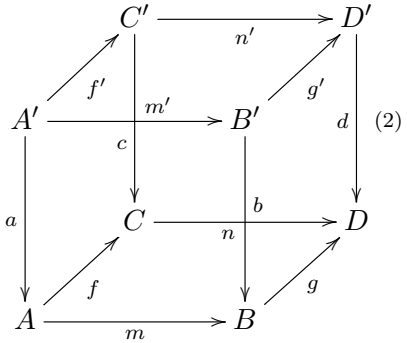
This appendix recalls and adapts definitions of related work used within proofs in this thesis. Furthermore, some helper propositions are presented needed to build a bridge between related work and the presented work.

First of all, the notion of S-reflecting IC-graph morphisms is needed which fully reflect subtypes of original nodes in their images.

Definition 10.1 (S-reflecting morphism (cf. Def. 4 in [38]))**.** An IC-graph morphism $f\colon G \to H$ between IC-graphs G and H is *S-reflecting* if it is injective and has the following property: $\forall (n_{11}, n_1) \in I(H)^*, n_0 \in G_N : n_1 = f_N(n_0) \implies \exists n_{01} \in G_N : f_N(n_{01}) = n_{11} \wedge (n_{01}, n_0) \in I(G)^*$. ◇

Definition 10.2 (van Kampen square (cf. Def. 4.1 in [23]))**.** A pushout (1) is a *van Kampen square*, if for any commutative cube (2) with (1) in the bottom and the left and front faces being pullbacks holds: the top face is a pushout ⇔ the back and right faces are pullbacks.





◇

Definition 10.3 ((weak) adhesive HLR category (cf. Defs. 4.9 and 4.13 in [38])). A category C with a morphism class $\mathcal{M}$ is called an *adhesive HLR category*, if

1. $\mathcal{M}$ is a class of monomorphisms closed under isomorphisms, composition ($f : A \to B \in \mathcal{M}, g : B \to C \in \mathcal{M} \Rightarrow g \circ f \in \mathcal{M}$) and decomposition ($g \circ f \in \mathcal{M}, g \in \mathcal{M} \Rightarrow f \in \mathcal{M}$),

2. C has pushouts and pullbacks along $\mathcal{M}$-morphisms and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks,

3. pushouts in C along $\mathcal{M}$-morphisms are VK squares.

$(\mathrm{C}, \mathcal{M})$ is called weak adhesive if pushouts in C along $\mathcal{M}$-morphisms are weak VK squares, i.e. the VK square property holds for all commutative cubes with $m \in \mathcal{M}$ and ($f \in \mathcal{M}$ or $b, c, d \in \mathcal{M}$) (see Def. 10.2). ◇

In the following, basic definitions of distributed object and distributed morphisms are recalled as well as basic results from [26] (slightly adapted to the notation in this thesis). In addition, a few propositions are presented building a bridge between those results and the present work.

Definition 10.4 (Path morphism and commutative functor (cf. Def. 1 in [26])). Given a graph D, a functor $\hat{D}\colon D \to \mathrm{C}$ (interpreting D as a category) and a path $p : n \xrightarrow{e1} \ldots \xrightarrow{e_k} n'$ in D, the *path morphism* $\hat{D}(p)\colon \hat{D}(n) \to \hat{D}(n')$ of $\hat{D}$ is defined along p as $\hat{D}(p) = \hat{D}(e_k) \circ \ldots \circ \hat{D}(e_1)$. For the empty path $\epsilon_n : n \xrightarrow{0} n$, $\hat{D}(\epsilon_n) = id_{\hat{D}(n)}$.

A functor $\hat{D}\colon D \to \mathrm{C}$ is commutative, if for any two paths $p_1, p_2 : n \xrightarrow{*} n'$ in D it holds $\hat{D}(p_1) = \hat{D}(p_2)$. ◇

Definition 10.5 (Distributed object and distributed morphism (cf. Def. 2 in [26])). Given a category C, a distributed object $\hat{D}$ over C (or just a distributed object, if C is implicit in the given context) consists of a graph D, called network graph, and a commutative functor $\hat{D}\colon D \to \mathrm{C}$, called diagram functor.

A distributed morphism over C (or just a distributed morphism, if C is implicit in the given context) $\hat{f}\colon \hat{D}_1 \to \hat{D}_2$ consists of a graph morphism $f\colon D_1 \to D_2$ and a natural transformation $\hat{f}\colon \hat{D}_1 \to \hat{D}_2 \circ f$.

Distributed objects and distributed morphisms over C form the category DisC. ◇

Proposition 10.6 (Category CompGraphs is a sub-category of category DisC). The category CompGraphs of composite graphs and composite morphisms is a sub-category of the category DisC of distributed objects and distributed morphisms over a category Graphs (being the category of graphs and graph morphisms).

PROOF. It is to show that *composite graphs* and *composite graph morphisms* are valid *distributed objects* and *distributed morphisms* along Def. 10.5.

Let $\hat{G}$ be a composite graph over composite network graph G and a functor $\hat{G}\colon G \to \text{GRAPHS}$ (interpreting G as a category). It is required that the functor $\hat{G}$ is commutative (cf. Def. 10.4), i.e., for any two paths $p_1, p_2 : n \overset{*}{\to} n'$ in the composite network graph G, their path morphisms satisfy $\hat{G}(p1) = \hat{G}(p2)$. The following cases can be distinguished:

- If n is a body node, all paths have length 0, since body nodes do not have outgoing edges.
- If n is an export node, there is exactly one path with length 1 running to its body node.
- If n is an import node there is
  - exactly one path with length 1 to an export node,
  - exactly one path with length 1 to a body node, and
  - exactly one path with length 2 to a body running over an export node. Here a triangle constellation may occur if a component's import refers to the same component's export. Consequently, there would be two paths between an import node and a body node. However, the present definition of composite graphs (cf. Def. 5.9) requires for such cases $\hat{G}(e_{IB}) = \hat{G}(e_{EB}) \circ \hat{G}(e_{IE})$ such that $\hat{G}$ is a commutative functor.
- Other cases are not possible in composite network graphs.

It follows that the properties according to commutative functors are satisfied.

Composite morphisms are valid distributed morphisms as they define graph morphisms restricted to composite network graphs and a natural transformation according to network and object structure illustrated in Fig. 5.6.

Consequently, category COMPGRAPHS complies with all properties of category DISC as defined in Def. 10.5. □

FACT 10.7 ((Co)completeness of category DISC (cf. Theorem 3 in [26])). If the category C is (co)complete, so is the category DISC.

In the following, the notion of persistent network graph morphisms is revisited (see [26, Prop. 4]). Furthermore, it is shown that composite network morphisms are always persistent. Persistence is needed to show that pushouts in DISC can be constructed component-wise.

DEFINITION 10.8 (Persistent morphism). A graph morphism $f\colon G \to H$ is *persistent* if for all nodes $n, n' \in G_N$ the following property holds:
If there exists a path $f(n) \overset{*}{\to} f(n') \in H$ then there exists a path $n \overset{*}{\to} n'$ in G (see [26, Prop. 3]). ◇

Proposition 10.9. Injective composite network graph morphisms are persistent.

Proof. The following shows that the constraint in Def. 10.8 is satisfied due to the specific structure of composite network graphs. Consider an injective morphism representing the evolution of a network graph, i.e., the image graph is the newer one, while all preserved network nodes and edges are mapped. That is, elements of the image graph are considered to be created if they do not have a preimage. Network edges connecting network nodes are always created together with their source nodes, i.e., non-connected network nodes in the preimage graph cannot be connected by a path in the image graph. In fact, the following cases in terms of creation may occur only:

1. Creation of a body node: This does not affect existing paths.
2. Creation of an export node: This operation requires a body node to which an edge is created with the export node as source. New paths between two existing nodes are not created.
3. Creation of an import node: This operation requires a body node and an export node to exist to which an edge is created each with the import node as source. Again, new paths between two existing nodes are not created.
4. Creation actions can be combined which does not affect the argumentation concerning paths.

Injectivity prevents the gluing of nodes and therefore the circumvention of these cases. These are all cases. Consequently, new paths between two existing nodes are not created. □

Proposition 10.10 (Category CompIGraphs is a sub-category of category DisC)**.** The category CompIGraphs of composite I-graphs and composite I-morphisms is a sub-category of the category DisC over a category C as defined in Def. 10.5, with C being the category of I-graphs and I-morphisms IGraphs.

Proof. The proof is analogously to the one of Prop. 10.6 above while the underlying category C has to be changed to IGraphs only. □