Detecting Adaptation Conflicts at Run Time using Models@run.time

vorgelegt von
Dipl.-Inform.
Frank Trollmann
geb. in Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender:	Prof. Dr. Stefan Jähnichen	(TU Berlin)
Berichter:	Prof. Dr. Dr. h.c. Sahin Albayrak	(TU Berlin)
Berichter:	Prof. Dr. Hausi A. Müller	(University of Victoria)
Berichter:	Prof. Dr. Julia Padberg	(HAW Hamburg)

Tag der wissenschaftlichen Aussprache: 11.Dezember 2014

Berlin 2014

Zusammenfassung

Selbstadaptive Softwaresysteme sind in der Lage, sich an die Heterogenität heutiger Computersysteme anzupassen. Adaptionen ermöglichen eine korrekte Funktionsweise trotz unterschiedlicher Computerplattformen, Fähigkeiten, Konfigurationen und Kontextsituationen. Hierfür muss das Softwaresystem trotz der Änderungen durch Adaptionen immer korrekt funktionieren. Um dieses Ziel zu erreichen kann Verifikation zur Designzeit mit einer Analyse zur Laufzeit kombiniert werden. Eine Verifikation zur Designzeit ermöglicht die frühzeitige Erkennung und Beseitigung von Problemen. Allerdings ist der Zustandsraum, welcher durch Adaption erreicht werden kann, oft sehr groß oder, in einigen Fällen, zur Designzeit nicht vollständig bekannt. Dies macht eine komplette Verifikation zur Designzeit unpraktisch oder sogar unmöglich. Analyse zur Laufzeit hat den Vorteil, dass ein großer Teil der zur Designzeit noch unbekannten Variablen feststeht. Statt einer Analyse des gesamten Zustandsraumes kann die Laufzeitanalyse auf dem aktuellen Zustand basieren.
Das korrekte Verhalten der Adaptionen ist einer der Aspekte, die sichergestellt werden müssen. Adaptionen verursachen Änderungen im Softwaresystem. Diese Änderungen können miteinander in Konflikt stehen. Das laufende Softwaresystem muss diese Konflikte erkennen können, um auf sie zu reagieren. In dieser Doktorarbeit beschreiben wir den Trollmann Ansatz zur Erkennung von Adaptionskonflikten. Dieser Ansatz kann zwei Typen von Adaptionskonflikten erkennen: Adaption-Adaption Konflikte und Adaption-Konsistenz Konflikte. Adaption-Adaption Konflikte treten auf, wenn Adaptionen sich gegenseitig beeinflussen. Dies führt dazu, dass verschiedene Reihenfolgen der gleichen Adaptionen unterschiedliche Ergebnisse liefern. Es kann auch vorkommen, dass nach einer Adaption eine andere nicht mehr durchgeführt werden kann. Adaption-Konsistenz Konflikte sind Situationen, in denen die Adaptionen das System in einen inkonsistenten Zustand führen. Um Adaptionskonflikte zu lösen muss ein selbstadaptives Softwaresystem in der Lage sein, diese zu erkennen.
Modellgetriebene Softwareentwicklung kann verwendet werden, um die Komplexität der Entwicklung und Handhabung von Softwaresystemen zu reduzieren. Bei einigen Ansätzen bleiben die Modelle zur Laufzeit erhalten und werden mit dem laufenden Softwaresystem synchronisiert. Der Trollmann Ansatz basiert auf einem solchen Ansatz. Adaptionen und Konsistenzbedingungen können auf diesen Modellen beschrieben werden. Als Formale Basis definieren wir einen Formalismus, der in der Lage ist, mehrere Modelle und deren Relation zu repräsentieren. Dieser Formalismus kann mit Graph Transformation und mit Nested Conditions kombiniert werden, um Adaptionen und Konsistenzbedingungen zu beschreiben. Basierend auf dieser Kombination von For-

malismen definieren wir eine Analysemethode für Adaptionskonflikte. Die Analyse ermöglicht die Erkennung beider Konfliktarten. Für erkannte Konflikte werden die beteiligten Adaptionen, Modellelemente und elementaren Adaptionsoperationen (Erstellen und Löschen von Modellelementen) extrahiert.
Das korrekte Verhalten des Trollmann Ansatzes wird durch Theoreme beschrieben und in der Arbeit bewiesen. Zusätzlich beschreiben wir die Implementierung des Trollmann Ansatzes im Rahmen des Eclipse Modeling Frameworks. Diese Implementierung wurde genutzt, um die Konflikterkennung im Rahmen eines Forschungsprojektes an der TU Berlin zu evaluieren.

Abstract

Self-adaptive software systems aim to cope with the heterogeneity of todays computing environments by adjusting themselves. Adaptations enable the software system to function correctly despite different computing platforms, capabilities, configurations and context-of-use situations. Self-adaptive software systems need to assure correct behaviour despite the changes imposed by adaptations. To achieve this goal design time verification can be complemented with run time assurance. Design time verification enables the detection and resolution of problems at design time. However, due to its adaptations the state space of a self-adaptive software system is often very large or, in some cases, partially unknown at design. This makes a complete verification impractical or even impossible. Run time assurance methods profit from the fact that a lot of the variables that are free at design time are bound at run time. Instead of a verification of the complete state space such methods can concentrate on assuring that there are no failures in the current state of the software system.

One aspect that needs to be assured at run time is the correct behaviour of the adaptation itself. Adaptations denote changes in the software system that can be in conflict with each other. The running software system needs to be able to detect these conflicts to resolve them. In this thesis we present the Trollmann approach to adaptation conflict detection. This approach is able to detect two types of adaptation conflicts: adaptation-adaptation and adaptation-consistency conflicts. Adaptation-adaptation conflicts are situations in which adaptations impact each other, leading to different results when applied in different order. The application of one adaptation can even disable another adaptation. Adaptation-consistency conflicts are situations in which adaptations leave the software system in an inconsistent state. The running software system needs to be able to detect and resolve these conflicts.

Model driven engineering has been applied to tame the complexity of the development and management of software systems. The Trollmann approach assumes a model driven engineering approach in which the models are available at run time and reflect the current state of the software system, its adaptations and its consistency requirements. The detection of adaptation conflicts is based on these models. As formal foundation for the approach we define the formalism graph diagrams that can be used to represent multiple models and their relation. This formalism is combined with graph transformation and nested conditions to describe adaptations and consistency requirements. Based on this combination of formalisms we define analysis methods for adaptation conflicts. These analysis methods are able to detect both types of conflicts and to point out the adaptations, model elements and elementary adaptation actions (i.e., the creation and deletion

of model elements) that are involved in the conflict. This information can be used by a conflict resolution mechanism.
The correct behaviour of the Trollmann approach is described by a set of theorems that are proven in this thesis. In addition, we present an implementation of the Trollmann approach in the scope of the Eclipse Modeling Framework. This implementation has been used to evaluate the performance of the analysis algorithms and test the approach in the scope of a research project at the TU Berlin.

Acknowledgments

I would like to thank my adviser Prof. Sahin Albayrak. The stimulating and productive environment at the DAI-Labor of the Technische Universität Berlin and your constant support enabled me to conduct this research. Our research projects provided the inspiration and experience that lead to this thesis.
I would also like to thank the other members of my committee: Prof. Hausi Müller and Prof. Julia Padberg. The invaluable advice you provided aided me in improving myself and my work. This thesis would be significantly worse without your critique.
In my time at the DAI-Labor I worked in the SPA-Team on developing and applying our model based approach in several research projects. I would like to thank all members of this team and especially Maximilan Kern, Johannes Fähndrich, Stephan Spiegel and Siyun Li for their ideas, feedback and the creative working atmosphere. It was a great pleasure to work with every single one of you during the last years.
Our main research has been conducted within the project *"BeMobility 2.0"*, which I was fortunate to coordinate for the DAI-Labor. I would like to thank the German Federal Ministry of Transport and Digital Infrastructure for funding this project and thus enabling our research.
A lot of my colleagues have supported my work in one way or the other. Your valuable suggestions, fruitful discussions, productive criticism and kind words helped me to develop my research. These are Andreas Rieger, Grzegorz Lehmann, Veit Schwartze, Marco Blumendorf, Sebastian Ahrndt, Daniel Freund, Mathias Wilhelm, Michael Quade, Cyrille Martin, Paul Zernicke and Nils Masuch.
I also want to thank my friends and family for their continued support. You provided a basis that I could hold on to when I was overwhelmed by the work before me and provided distractions when I needed them the most.
Last but certainly not the least I would like to thank my wife. Your constant support enabled me to keep going. Thank you for always being there for me and putting up with me using so much of our free time to work on this thesis.
It is almost impossible not to forget to acknowledge someone when finishing a project of this size and duration. I would like to thank and apologize to everyone I forgot to mention.

Contents

List of Tables

List of Figures

1. Introduction

Mark Weiser coined the notion of ubiquitous computing, predicting that *"In the 21st century the technology revolution will move into the everyday, the small and the invisible."*[1]. In this vision computing can happen everywhere and anywhere. A software system in such environments can run on various platforms with different properties and a software system that has been designed with one specific platform in mind can be suboptimal on other platforms. Accordingly, there is huge potential for software systems to optimise themselves to fit the environment, the capabilities of available devices, the user or the current situation. Software systems that are able to *"adjust their behaviour in response to their perception of the environment and the system itself"*[2] are called self-adaptive software (SAS) systems. One example of an adaptable aspect of a software system is its user interface (UI). Applying the notion of ubiquitous computing to this domain leads to ubiquitous user interfaces that can adapt themselves to the current context of use [3].

The development of SAS systems is a complex task. Zhang et al. refer to a non-adaptive software system as a steady-state program. A steady-state program is *"a non-adaptive program suited for a specific set of environmental conditions"*[4]. In this view a SAS system consists of a set of steady-state programs and adaptations, which represent transitions between the steady state programs. The development of a SAS system requires the additional effort of analysing potential environment conditions and specifying how the SAS system adapts to them.

All reachable steady-state programs of a SAS system need to function correctly. Thus, the assurance of SAS systems is also considerably more complex than the assurance of steady-state programs. The state space of potential steady-state programs can be large and most context variables are unknown at design time. Due to these problems Zhang et al. state that static design time techniques alone are *"insufficient to provide assurance for complex adaptive programs"*[4]. This is even more evident in open-adaptation approaches which change the adaptive behaviour and thus the state space of reachable steady-state programs at run time [5]. For these reasons design time analysis methods need to be complemented with run time assurance methods. At run time most of the context variables that are unknown at design time can be derived from the current context. The analysis can be limited to the current steady-state program and some of the surrounding states and adaptations to reduce complexity. Since the developer of the software system is usually not available to resolve or avoid detected problems run time assurance mechanisms need to be complemented with the capability to resolve any detected issues automatically or in cooperation with the user.

Model driven engineering (MDE) [6] has been applied to manage the complexity of the development of software systems such as SAS systems. In this approach the development

of a software system is based on models at various levels of abstraction. Such models can also be used for assurance, for instance via model checking [7], which is traditionally used to check models of hardware systems but has also been applied to software systems [8]. Model-based methods also play an important role in run time assurance in combination with run time modelling approaches like models@run.time [9]. Such approaches utilize models during the runtime of a software system and make them available for tasks like adaptation, analysis or assurance. The use of models@run.time for assurance has been subject to considerable discussion. Recently the models@run.time community proposed a research roadmap for the use of models@run.time for run time assurance [10].

In this thesis we deal with a sub problem of run time assurance: the detection of adaptation conflicts. Adaptation conflicts can result from the application of adaptations to the current steady-state program. Two kinds of adaptation conflicts can be distinguished: conflicts between adaptations and conflicts between adaptations and consistency requirements. Consistency requirements can be interpreted as a set of conditions that all steady-state programs need to fulfil. The aim of this thesis is to detect these conflicts and provide information that can be used by a conflict resolution mechanism. Different resolution strategies can be based on this information. Developing, analysing and comparing these strategies will be done in future work.

Our approach to conflict detection is called the Trollmann approach. This approach is based on models@run.time. It assumes that the current steady-state program, its adaptations and its consistency requirements are represented by models that can be subject to analyses. The analysis algorithms assume that these models are described in a formal framework that is part of the approach. This formal framework is able to express models of different modelling language and is thus independent of the models used to describe the current steady-state program in a specific software framework.

In the following sections we detail the topic, goals and contributions of this thesis (cf. Section 1.1) and describe how they are related to the structure (cf. Section 1.2).

1.1. Goals and Contributions

Our main goal is to develop a framework for runtime adaptation conflict analysis in a models@run.time based SAS system. The analysis needs to be able to detect adaptation conflicts and derive information about reasons for detected conflicts.

Adaptation conflicts can be grouped in two categories: adaptation-adaptation conflicts and adaptation-consistency conflicts. An adaptation-adaptation conflict occurs when different orders of the same adaptations lead to different steady-state programs or are not possible at all. In these situations the running software system needs a way to decide in which order to execute these adaptations. Adaptation-consistency conflicts occur when adaptation leaves the software system in an inconsistent state (i.e. violating some consistency requirement). The running software system needs to detect them to preserve or re-establish consistency.

A complete analysis of adaptation conflicts at design time is not always possible. The state space of steady-state programs determined by adaptations can be large, infinite or

partially unknown at design time. In such cases a run time analysis is required to avoid adaptation conflicts.

Models@run.time are a good basis for the run time detection of adaptation conflicts. In a models@run.time approach the models are connected with the running software system in such a way that the models always reflect the current state of the software system and can be manipulated to cause adaptations. Accordingly, adaptations and consistency requirements can be expressed based on these models. This high level representation of the software system and its adaptations and consistency requirements provides suitable input for conflict analysis. Further details on the models@run.time approach are presented in Section 2.1.6. The analysis should provide information about detected conflicts. The purpose of this information is to inform run time conflict resolution mechanisms about reasons for detected conflicts.

Our main research question is the following:

How can models@run.time be used for the run time detection of adaptation conflicts and the provision of information about the detected conflicts?

To answer this research question we define the Trollmann approach to conflict detection. This approach is based on a formalism for run time models, adaptations and consistency requirements. Our methods for detecting adaptation conflicts and providing information about detected conflicts are based on this formalism.

In current model driven engineering approaches there is a variety of models. Among others there are goal modelling approaches like RELAX [11], focusing on the goals of the software system; behavioural modelling approaches that aim to model the behaviour of the software system by using state-based models like state charts in the unified modeling language (UML) [12], architecture models that define components and their connection like the Wright architecture description language [13], control models that model system behaviour like the feedback loops for adaptive software systems [14], and user interface modelling approaches like the models in the Cameleon reference framework [15, 16]. Each of these models can be subject to adaptation and consistency requirements. Thus, a general approach to conflict detection should be independent of the modelling language used in the analysed models.

The goal of the formalism in the Trollmann approach is to achieve this independence. This formalism is not intended to replace existing models and their techniques and languages. It can represent different modelling languages and can be considered an alternate representation of the models similar to the idea of an abstract syntax. The model itself can be implemented in the language and technological space that is best suited and has an alternative abstract syntax for detection of adaptation conflicts.

Adaptations and consistency requirements are not always restricted to one model at a time. Models that coexist at runtime are usually related. For example, in UsiXML [17], a model-based approach to the development of user interfaces, three types of models are used. A task model describes the tasks of the user and software system. These tasks are related to interaction elements in an abstract user interface model that enable the user to accomplish these tasks. The interaction elements describe user interaction independent of the interaction modality. An additional concrete user interface model contains the

implementation of these interaction elements in specific modalities. All three models are related and should be adapted, constrained and analysed together.

Accordingly, the formalism should be able to represent, constrain and adapt multiple models and their relations. The formalism should also be formally founded to enable the formulation of analysis methods. In the Trollmann approach we define a formalism, called graph diagrams, for the representation of models and model relations. We also show the compatibility of this formalism with graph transformation [18] for representing adaptations and nested conditions [19] for representing consistency requirements. This formalism is our first contribution.

Contribution 1. *A formalism that is able to represent structure, adaptations and consistency conditions for multiple related models independent of their modelling language*

The second contribution is the analysis method for the detection of adaptation conflicts. The analysis is based on a representation of the current state of the software system, the consistency requirements and the adaptations that are supposed to be applied in the current state. It can be used to detect conflicts whenever one or more adaptations are selected to be applied. Our analysis methods are an extension of existing methods in the area of graph transformation and nested conditions.

Contribution 2. *An extension of existing analysis methods to enable the detection of adaptation conflicts at run time*

Our third contribution is an implementation of the Trollmann approach in the scope of the Eclipse Modeling Framework (EMF) [20], a widespread framework for model-based development. This implementation maps our formalism to EMF models and implements the analysis methods on this basis. The result is a Java library that can be used in projects that are based on EMF.

Contribution 3. *A proof-of-concept implementation of the analysis methods based on the Eclipse Modeling Framework*

We use this implementation as a proof of concept and to evaluate the complexity of our analysis methods based on a running example and a case study. The purpose of the running example is to test the performance of the implemented methods. For this the running example is amplified to test the influence of several factors on the performance of the analysis. The purpose of the case study is to evaluate the applicability of the analysis methods and the proof-of-concept implementation in the scope of a realistic problem. This evaluation is the fourth contribution.

Contribution 4. *An evaluation of the run time behaviour of the implemented methods*

In the next section we describe the structure of the thesis and locate the contributions within this structure.

1.2. Structure

In this section we explain the structure of the thesis. We also locate the goals and contributions within the chapters.

In Chapter 2 we present the fundamentals that are required to understand the content of this thesis. The fundamentals are twofold. On the one hand, model driven engineering and models@run.time approaches form the conceptual basis of the approach. Section 2.1 describes the author's views on the main terms used in this thesis. On the other hand, the formalism and analysis methods defined in this thesis make use of concepts from graph transformation theory. Section 2.2 describes graph transformation, nested conditions and existing formal results. These results that form the basis of the formalism in Contribution 1 and the analysis methods in Contribution 2. The main definitions and theorems referenced in Chapter 2 are provided in Appendix A.

We give a more detailed description of the problem statement in Chapter 3. This chapter describes adaptation conflicts in detail. It also indicates mechanisms to resolve such conflicts and derives a set of questions that need to be answered by the analysis methods from Contribution 2.

In Chapter 4 we review related work. This chapter describes approaches for representing models and their relations as potential alternatives for the formalism in Contribution 1. The related work chapter also describes alternatives for representing adaptations and structural requirements and describes existing approaches that deal with adaptations and the detection of adaptation conflicts.

The case study in Contribution 4 has a central place in this thesis. It is used for the evaluation of the approach and the implementation. We also use a simplified version of the case study as running example for the Trollmann approach. Chapter 5 present the case study and its models.

We describe the Trollmann approach to conflict detection in Chapter 6. This chapter first presents the prerequisites (cf. Sections 6.1) and the high-level approach (cf. Section 6.2) in general terms of model driven engineering. We then describe the simplified running example (cf. Section 6.3). Subsequently, we define the formalism for representing models, adaptations and conditions (cf. Section 6.4). In this scope we introduce Graph Diagrams (Contribution 1) to represent multiple related models. Based on this formalism we describe the algorithms for the detection of adaptation conflicts (cf. Section 6.5). This algorithm is Contribution 2. We then elaborate how the formalism and analysis methods can be used to provide knowledge for conflict resolution (Section 6.6). This enables us to answer the questions derived in Chapter 3.

The correct behaviour of the analysis methods are formulated as theorems. We provide proof for these theorems in Appendix B.

Chapter 7 presents the implementation of the analysis methods in the Eclipse Modeling Framework (Contribution 3). In this chapter we discuss the basics of the Eclipse Modeling Framework and its relation to the formalism of Graph Diagrams. We also describe the implemented utility classes that allow the execution of the analysis methods described in Chapter 2. For testing the implementation we apply it to the running example from Chapter 6.3. To test the influence of different factors on the performance of our

algorithms we integrate a way to artificially increase the model size, the number of rules and the number of conditions. We systematically test different configurations of these factors and record execution times. Based on these test results we analyse the influence of the factors on the performance. This analysis is the first part of Contribution 8.

The second part of Contribution 3 is the application of the implementation of the analysis methods in the scope of the case study from Chapter 5. In Chapter 8 we describe this evaluation. We present the evaluation criteria and the test setup and give an interpretation of the results. In this chapter we also discuss the main theorems of the Trollmann approach and their significance.

Chapter 9 concludes the thesis.

2. Fundamentals

This chapter describes the conceptual framework for this dissertation. This framework includes two general topics. The first topic is model driven engineering. In this area of research several definitions for the main terms exist. Section 2.1 provides the definitions used in this thesis. In this section we explains and relate the terms model, modelling language, meta model, model driven engineering, run time, models@run.time, model transformation and model reconfiguration.

The Trollmann approach is based on results in the framework of $\mathcal{M}$-Adhesive Categories. Section 2.2 describes $\mathcal{M}$-Adhesive Categories and the relevant results. In this section we also describe the formalisms of graph transformation and nested conditions. Finally, we give a brief conclusion of the chapter.

2.1. Model Driven Engineering

Model driven engineering is a wide field of research that is of interest in several research communities. However, depending on the intended use of models, the definition and meaning of terms vary. As Favre and Ngyuen observed, the *"concepts of model, meta model and transformation are usually ill-defined in industrial standards like the MDA or XML"*[21]. This section provides the authors understanding of these terms. This understanding is presented as a set of model relations that are based on the work of Favre and Mahr. We present the relevant relations step by step.

Section 2.1.1 discusses the definition of models. The definitions of modelling languages and meta models are given in Section 2.1.2. We describe model transformations and model reconfigurations in Section 2.1.3. Section 2.1.4 describes model driven engineering. In this thesis we focus on the run time of a software system. Section 2.1.5 goes into detail on the term run time and special run time requirements. Models@run.time is a specific model driven engineering approach that utilizes models during run time of a software system. We describe this approach in Section 2.1.6.

2.1.1. Models

The term model is widely used, even outside of the software engineering domain. Several authors have tried to give a description on what makes a model. The Model Driven Architecture manual defines a model as *"a representation of part of the function, structure and/or behaviour of a system."*[22], whereas Seidewitz considers any *"set of statements about some system under study"*[23] to be a model. Favre describes a general understanding on what a model is as *"a system that enables to give answers about a system under study without the need to consider directly this system under study (SUS)"*[24].

Our definition of a model is a synthesis of the definition of Favre [25] and the "Modell des Modellseins" (english: "model of what a model is") by Mahr [26].

Both definitions define models based on a set of relations. Favre calls a model that represents model artifacts and model relations a mega model [25] and distinguishes this term from a meta model, which is a representation of a modelling language, as discussed in the next section. We also make use of mega models for model definitions. The mega model given in this section is extended with further entities and relations in the following sections. The graphical representation of these mega models is used for a conceptual description of the approach in Section 6.2.

Both Favre and Mahr concur in that anything can be a model. In accordance with Favre we use the term *system* to refer to anything. A system is not recognizable as a model because of its structure or some inherent properties but because of its relation to other systems. The definitions of Favre and Mahr focus on these relations. A combination of both mega models is depicted in Figure 2.1. It contains two relations: *Representation Of* and *Purpose Of*.

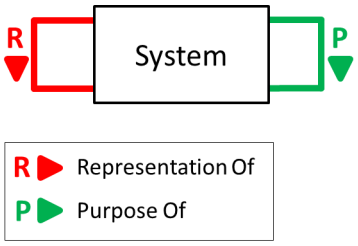


Figure 2.1.: Two relations that make a model.

The relation *Representation Of* reflects that a model represents another system, called its system under study (SUS). The model can be used to derive knowledge about the SUS. The representation is usually not complete but restricted to certain aspects of the system under study. Thus, the model is an abstraction of the system under study that contains knowledge on some relevant aspects. Both the mega models of Favre and Mahr contain a notion for this relation. Muller et al. further researched and categorized this relation into different classes of *Representation Of* relations [27, 28]. However, for the purpose of this thesis that level of detail is not required.

The mega model of Mahr also contains a relation called *Purpose Of*. This relation denotes that a model, as an abstraction of some other system, is created with a purpose in mind. This purpose determines which aspects of the system under study need to be reflected in the model. Favre's mega model does not contain an explicit notion for the purpose. However, he states that models *"are not expected to be perfect, but just 'good-enough' for a given purpose"*[25].

One example of a model is the blueprint of a house. The example is shown in Figure 2.2. The blueprint (model) represents the house (system under study). It can be used to derive the layout of the rooms or locate walls, staircases and windows. However, it does not represent all aspects. The colour of the walls is for example not derivable

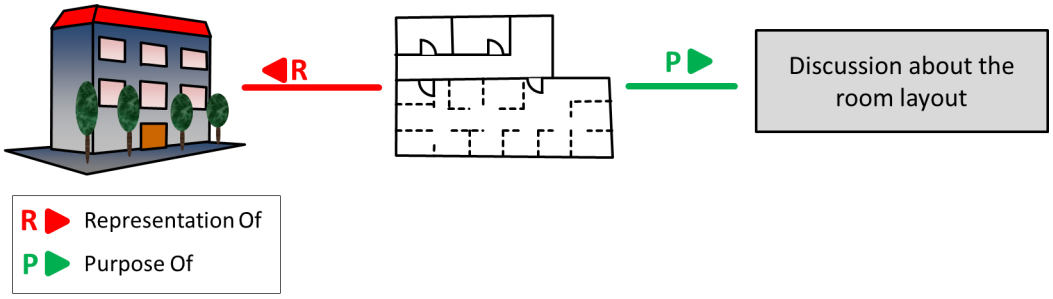


Figure 2.2.: Example: A blueprint of a house is a model.

from the blueprint. The blueprint can have several purposes. One example purpose is shown in the figure. In this case the blueprint is used to discuss the layout of rooms before building the house. The construction company is not concerned with painting the walls. For this reason the model does not need to contain knowledge about the colour of the walls to be used for this purpose.

Depending on the purpose and nature of their models different authors focus on different properties of models. The following properties are usually mentioned:

- *Abstraction*: This property usually means that a model does not represent all information about the SUS. This usually refers to the fact that the model only contains the information that is relevant to the purpose and abstracts from irrelevant information.

- *Simplification*: Simplification can also be understood as abstraction since the model can be considered simpler as it does not contain irrelevant information. In some cases this property is also understood as a simplification of access, meaning it is easier to retrieve the information from the model than from the system under study itself.

- *Description and Specification*: Some approaches also distinguish between descriptive models and specification models (or prescriptive models) [23, 25]. This is a property of the purpose of a model. If the purpose of a model is to build the SUS from the model we say the model is a specification model. If the model is solely used to retrieve information about an existing SUS it is descriptive. This distinction is further developed in Section 2.1.6 to characterize models@run.time.

The point of view taken in this section is a very broad one. For example, it also enables the classification of a Java or C Program as a model as both represent a software system and are used for the purpose of specifying how the system will behave. As stated above, this definition can be argued. However, for our purpose this general understanding of the term model is a sufficient basis.

2.1.2. Modelling Languages and Meta Models

Modelling languages are an important mechanism for structuring models as they enable to group models with similar structural properties. We use a simplified view on modelling languages. This view is in accordance with Favre [24]. Instead of considering syntax, semantic or constructing grammars, a modelling language is defined as the set of models that are part of this language.

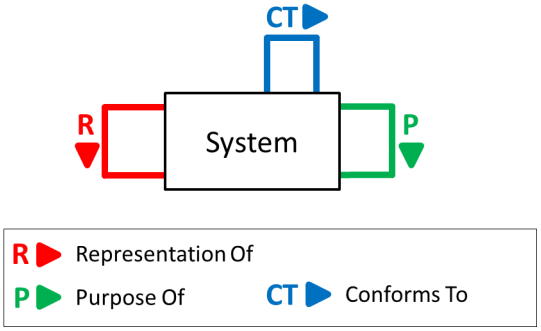


Figure 2.3.: The meta model for mega models with the relation *Conforms To*.

A meta model is a model of a modelling language. Different from a mega model, which describes how models relate to each other and to other artifacts, a meta model describes the commonalities of models that belong to the same modelling language. A meta model can describe a common syntax or semantic. This view concurs with Favre who adds a relation *Conforms To* to his mega model to denote when a model is conform to a meta model. Figure 2.3 shows the enhanced mega model with the relation *Conforms To*. The mega model in this figure is actually a meta model that can be used to express mega models. The examples conform to this meta model.

According to Favre a model conforms to a meta model if the meta model represents the modelling language of the model [24]. If the modelling language is expressed as a set, the model is an element of the modelling language in set-theoretical sense. This situation can be seen in the scope of the blueprint example in Figure 2.4. In this example the modelling language of blueprints is the set of all blueprints and the meta model is the legend of the blueprint. The blueprint conforms to its legend. Note that the relation *Element Of* is not mentioned in the meta model for mega models in Section 2.3 as it will not be directly referred to in the approach section. It can, however also be used as a mega-modelling relationship as done by Favre.

A meta model is also a model. Meta models can also be structured in languages and meta models of their own. The meta model of a meta model is called a meta meta model. In theory there can be endless meta levels. However in most software engineering approaches two or three levels are sufficient.

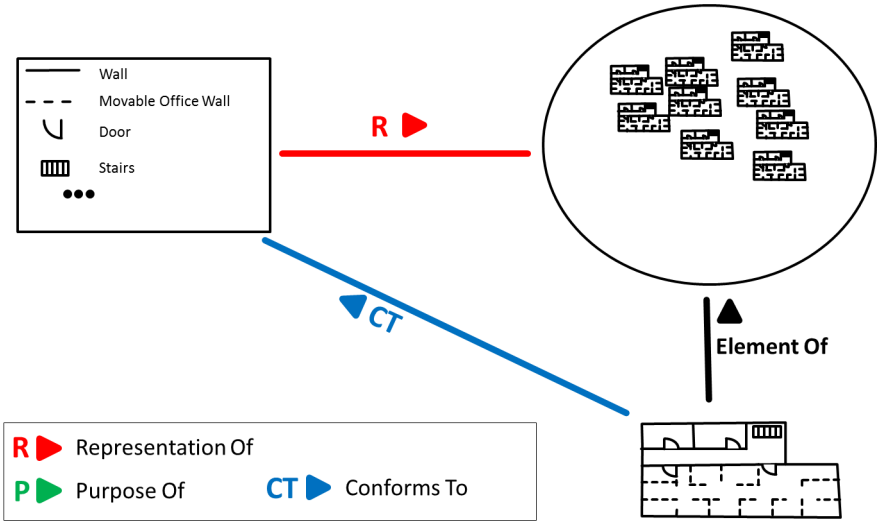


Figure 2.4.: Example: A blueprint is conform to its legend.

2.1.3. Model Transformation and Model Reconfiguration

Model transformation can also be defined on a general level. Any transformation of a set of models into a set of other models is called a model transformation independent of the modelling languages of the source and target model, their relation or purpose. In order to structure the world of model transformation Mens et al. define a taxonomy [29]. This taxonomy enables to classify model transformation approaches with regards to properties of the transformation, e.g., the relations of the source and target model.

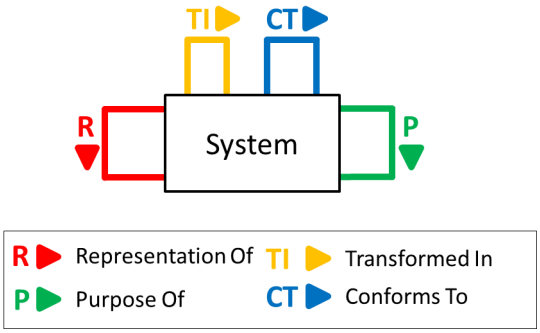


Figure 2.5.: The meta model for mega models with the relation *Transformed In*.

Favre and Nguyen use a mega-modelling relation to denote that a model is transformed into another one [21]. They also discusses that this relation (denoting a so-called transformation instance) can be treated like any other system, enabling a language of transformation instances (transformation language), models of transformation languages (transformation models or transformation specifications) and languages for models of

transformations. For this thesis the relation *Transformed In* is sufficient. An extended version of the meta model for mega models is shown in Figure 2.5.

For the Trollmann approach a special kind of model transformation is important. The Trollmann approach assumes a run time view and abstracts from how the models have been developed and whether they have been transformed from each other or other models during the development phase. Instead, it focuses on the run time models and their joint evolution. It assumes a fixed set of models that is interpreted at run time and that run time adaptation is caused by an evolution of these models. This evolution of models forms a special kind of model transformation, called model reconfiguration.

The properties of a model reconfiguration can be specified by referring to some of the categorisation characteristics identified by Mens et al. [29]. A model reconfiguration is a model transformation that is:

- **endogenous:** The transformation preserves the conformance to meta-models. This property assures that after the reconfiguration the models still are an element of the same modelling language.
- **horizontal:** The transformation leaves the models on the same level of abstraction. In some modelling languages there is a choice on how detailed the system under study is modelled. Since the models are supposed to represent the same SUS and be usable for the same purpose after the transformation they are required to stay on the same level of abstraction.

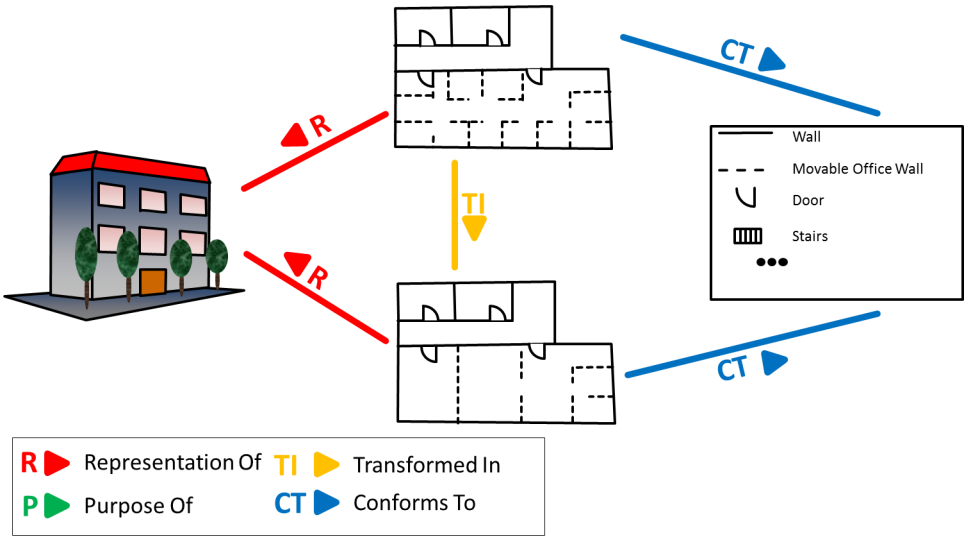


Figure 2.6.: Example: The reconfiguration of a blueprint.

In terms of the classification of model transformation approaches given by Czarnecki and Helsen [30] model reconfiguration can be seen as an in-place transformation that

changes an existing model instead of creating a new one. Several other authors also use this terminology (e.g., Bierman et al. [31]).

An example for a model reconfiguration is shown in Figure 2.6. One version of the blueprint of the house is transformed into a different version with changes in the wall-configurations. The result of the transformation conforms to the floor plans legend. This is ensured by the reconfiguration being endogenous. The horizontal property cannot be seen in the mega model directly as it refers to which knowledge can be derived via the *Representation Of* relationship. It can be interpreted as the model being usable for the same purpose after the transformation.

Model transformations and reconfigurations play an important role in model driven engineering. The next section describes this approach.

2.1.4. Model Driven Engineering

One of the most prominent uses of models in software engineering is model driven engineering. Favre describes MDE as *"a subset of system engineering in which the process heavily relies on the use of models and model engineering"* [25]. Model driven engineering being a subset of system engineering refers to the goal of MDE, which is to engineer some kind of system and not the models itself. This system might or might not be a software system. In MDE this process makes heavy use of models and model engineering (the process of engineering models).

Favre and Ngyuen [21] state that *"the goal of model driven engineering is to drive the process through a set of reusable transformation functions"*. Thus, MDE is closely related to model transformation. In most model driven engineering approaches the development of the software system spans multiple models that are transformed from each other. These models are often related because they represent similar views on the same aspects of the SUS or have been transformed from each other.

Several approaches to structure this development process in different domains do exist. However, since the focus of this thesis is on run time aspects we will not go into detail on model driven engineering and a classification of development approaches.

Although in classical MDE the purpose of the models is to generate the final software system, it can be advantageous to keep the models alive even while the system is running. One such approach is the models@run.time approach. Runtime and models@run.time are described in the next two sections.

2.1.5. Run Time

The lifecycle of a software system can be divided into several phases. Among others, we can distinguish the time when a system is designed (design time), developed (development time), loaded into memory (load time), tested (test time) and when it is executed (run time). These phases can sometimes intersect and the development process can in general go through these phases in an arbitrary order and in multiple iterations. Each phase can rely on models. This section characterizes the implications of run time for modelling approaches.

In most MDE approaches models are used to support the developer during the engineering process. The finished product, usually a complex software system, does not contain the original models but is created from them via model transformations. For instance, the Model Driven Architecture [22] specifies that the development starts with a computation independent model (CIM) of the software system and via model transformation stepwise generates and defines a platform independent model (PIM), a platform specific model (PSM) and a platform model. In software engineering the platform model is usually some byte code which can be executed. The purpose of the CIM, PIM and PSM is to ease the development process. They are not available during the run time of the developed software system.

The reason why most conventional MDE approaches do not carry over the models to run time is that the models have been designed to be used by the developer of the software system who isn't available at run time. The purpose of run time models on the other hand is to aid the running software system. This different purpose and the use at run time imply the following set of potential limiting factors for run time models:

- *high level of automation:* At run time the user is usually the only human agent that can interact with the models. The software system cannot assume the user to be a domain expert. Thus, processes involving run time models usually involve minimal or no user involvement. Accordingly, these processes need to be highly or completely automated.

- *low resource use:* In many running software systems resource consumption is an important concern. Depending on where the software system is deployed it may be severely restricted in resources like memory or energy consumption. Run time processes involving models should also regard this aspect.

- *low execution times:* One particular resource that is often critical is execution time. Processes involving models in a running system are usually more constrained in this parameter than design time processes.

Naturally, all three characteristics highly depend on the specific software system and its domain. For example, the brake mechanism should react within a matter of milliseconds (or even less) and, apart from the user triggering the process, should be fully automated. Interaction inside of a graphical user interface on the other hand can take up to several seconds [32] and might even involve additional interaction with the user if the software system needs information to perform its tasks.

In addition to these limiting factors, the use of models during run time can have considerable benefits. During the design of the software system the actual environment is often unknown. At run time many of the free variables are bound which enables more precise analysis and reasoning processes based on these models.

2.1.6. Models@run.time

One approach for using models during the run time of a software system is called models@run.time [9]. Although the exact understanding on what makes a model@run.time

is still under discussion in the community, Blair et al. have a preliminary definition. According to this definition, a model@run.time is *"a causally connected self-representation of the associated system that emphasizes the structure, behaviour, or goals of the system from a problem space perspective"*[9]. Starting from this perspective I and Grzegorz Lehmann developed a general understanding and classification framework for models@run.time. The views presented in this section have been developed in this collaboration but have been simplified to fit the purpose of this thesis.

From the definition of Blair et al. several key concepts in models@run.time can be extracted. The first is the model being a self-representation, meaning that the model is a part of the running software system. Blairs definition gives some examples (structure, behaviour, goals) of the aspects the model@run.time can represent. These examples can be considered to be incomplete. In general a model@run.time can represent any aspect of the running system, e.g., its user interface or context or use. The main thing that makes a model during the systems run time a model@run.time is that the model and its system under study are in causal connection.

The causal connection has been defined by Pattie Maes in the context of computational reflection [33]. In the context of models it describes that the model and its system under study can evolve to remain consistent even when one of them changes. This can be decomposed into two directions depending on whether the original change occurs in the model or the SUS. This is similar to the distinction between description and specification (or prescriptive) models made by Favre [25] and Seidewitz [23]. Grzegorz Lehmann, Marco Blumendorf and I applied this distinction to models that co-exist and co-evolve with their SUS, leading to the notion of prescriptive and descriptive causal connection [34]. The direction is interpreted from the point of view of the model. The descriptive causal connection means that the model describes the system under study and accordingly, if the SUS changes the model needs to reflect these changes (if they concern the aspects the model represents). If the model is in prescriptive causal connection to the system under study it can change to cause changes in the SUS. In this case the SUS can adapt to encompass changes in the model. A model can be in both descriptive and prescriptive causal connection, meaning some aspects reflected in the model are represented in a descriptive way and some are reflected in a prescriptive way.

We define a model@run.time as a model that is used at run time and is in causal connection (either descriptive or prescriptive) with its system under study. We use a new relation *Prescriptive For* to represent this connection. If the model is prescriptive for its SUS during the software systems run time they are in prescriptive causal connection. If the SUS is prescriptive for its model they are in descriptive causal connection. Figure 2.7 shows the extended meta model for mega models.

The running example of a blueprint is not a model at run time. Although the blueprint can be used after the building has been finished (i.e., at run time of the building), it will neither change when the building structure changes (it is not in descriptive causal connection) nor will changes of the blueprint lead to changes in the building (it is not in prescriptive causal connection). For this reason a different example is used for models@run.time. This example can be seen in Figure 2.8. It consists of a voodoo puppet, which represents an unfortunate person. The way this puppet is supposed to work is

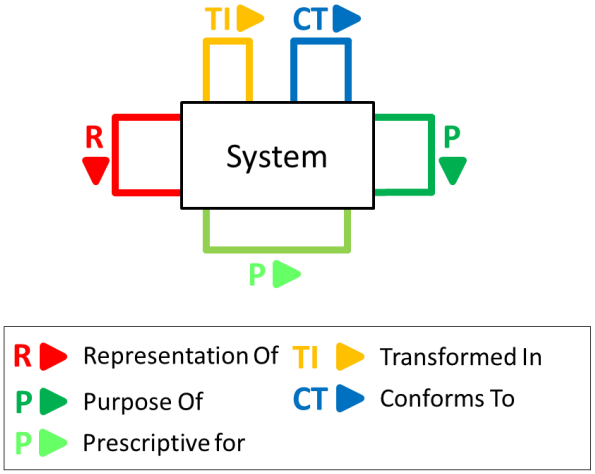


Figure 2.7.: The meta model for mega models with the relation *Prescriptive For*.

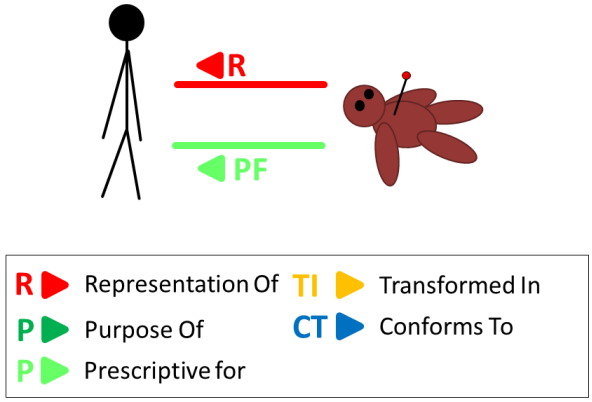


Figure 2.8.: Example: A voodoo puppet can be considered a model@run.time.

that if the puppet is injured, for example by sticking a needle in it, the human will be injured accordingly. Thus, the puppet is in *Prescriptive For* relation to the human during run time (which in this example can be interpreted as either the persons life time or the time of the voodoo ceremony, depending on what is the target system).

The importance of the causal connection for software engineering lies in its role in synchronizing the run time model with its SUS. The current state of the software system is reflected in its models, either because the models have been manipulated to cause the change in the first place (prescriptive causal connection) or because the change is propagated to the models (descriptive causal connection). This is advantageous for run time modelling approaches as it enables the analysis and manipulation of the models in place of the running system based on the assumption that the causal connection propagates changes correctly.

2.2. $\mathcal{M}$-Adhesive Categories

This section describes results from the area of $\mathcal{M}$-adhesive categories [35]. The notion of $\mathcal{M}$-adhesive category has been defined as a generalization of properties that are required to proof the main results in graph transformation theory. A modelling language is an $\mathcal{M}$-adhesive category, if it can be expressed as a category that fulfils a set of conditions. These conditions ensure that main results in graph transformation theory hold for this modelling language.

In Section 2.2.1 we describe $\mathcal{M}$-adhesive categories. In the following sections we define graph transformation (cf. Section 2.2.2) and nested conditions (cf. Section 2.2.3). $\mathcal{M}$-adhesive categories enable the application of several theoretical results. We summarize these results in Section 2.2.4. The formalism of attributed typed graphs is the basis for graph diagrams. We present this formalism in Section 2.2.5. For readability the formal definitions are omitted from this section. The reader can find them in Appendix A.

2.2.1. Definition of $\mathcal{M}$-Adhesive Categories

In order to represent adaptations and structural conditions the Trollmann approach uses the formalisms graph transformation and nested conditions. These formalisms can be applied to several modelling languages. They are formulated on a general basis to avoid repeated proofs for major results in graph transformation theory for each modelling language. This basis is category theory. A category abstracts a modelling language as a collection of objects and morphisms. The objects represent all valid models of the modelling language. The morphisms represent mappings or functions between them.

In order to proof the main results in algebraic graph transformation a category needs to have certain properties. Lack and Sobocinski defined adhesive categories for this purpose [36]. Ehrig et al. extended this type of categories to high-level replacement (HLR) categories [37]. HLR categories have been extended to horizontal and vertical weak adhesive HLR categories, also called $\mathcal{M}$-adhesive categories [35]. The purpose of these extensions is to capture the requirements for proofs in algebraic graph transformations in as weak a form as possible in order to enable the application of these results to as many modelling languages as possible.

The definition of vertical weak adhesive categories is given in Definition 25 in Appendix A.1. It describes the properties that a category, together with a class of $\mathcal{M}$ of monomorphisms, needs to fulfil. Some prominent $\mathcal{M}$-adhesive categories are the category of sets **Sets**, the category of graphs **Graphs** and the category of Petri Nets **PTNets** where in all these examples the class $\mathcal{M}$ consists of all monomorphisms.

Some categorial constructions preserve these $\mathcal{M}$-adhesive properties [38]. Several categories can be derived from existing $\mathcal{M}$-adhesive categories via these constructions. We make use of the construction of a functor category to define graph diagrams.

2.2.2. Graph Transformation

This section describes the basic notions of graph transformation in the double pushout approach. The main definitions are given in Appendix A.2. For a more detailed review of graph transformation the reader may refer to [38].

Traditionally there are two main approaches for graph transformation. The double pushout approach [18] and the single pushout approach [39]. In this thesis we utilise graph transformation in the double pushout approach. The definitions are formulated on the level of category theory assuming an $\mathcal{M}$-adhesive category.

The definition of a production, also called graph transformation rule, is given in Definition 26 in Appendix A.2. A production is a description on how to transform a pattern L, called the left hand side, into a pattern R, called the right hand side. This is done via an intermediate model K and two morphisms l and r that map K into L and R respectively. The morphism l describes which elements are removed when applying the production. The morphisms r describes which elements are added.

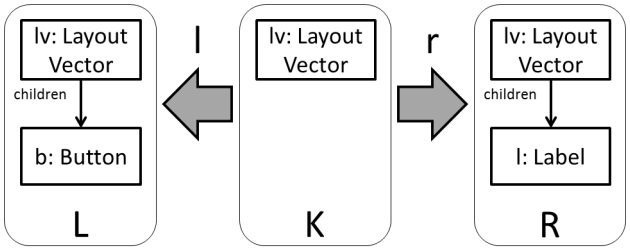


Figure 2.9.: Example: A graph transformation production.

Figure 2.9 contains an example for a graph transformation production. The production operates on attributed types graphs. This formalism will be described in Section 2.2.5. The transformation substitutes a button by a label. Its left hand side contains a layout vector, containing a button. In the morphism l the button is removed. In the morphism r a label is added.

A production can be applied to any occurrence of the pattern specified by L. This occurrence is defined by a morphism that maps L into the transformed model. This morphism is called a match morphism. A transformation can be derived if this morphism exists and fulfils certain properties. During the transformation the occurrence of L in G is substituted by R. This yields an object from the same category (i.e. modelling language). Graph transformation is a heterogeneous technique for model transformation. Thus, it is a suitable technique for model reconfiguration. The application of graph transformation productions is formally defined in Definition 27 in Appendix A.2

Our analysis for adaptation conflicts is essentially an analysis of adaptations implied by graph transformation productions. A production can be applied in multiple ways to a model if multiple match morphisms exist. A specific adaptation is thus represented by a graph transformation production in combination with a match morphism. This pair forms a so-called production application. It can be defined as follows:

Definition 1 (Production Application)**.** *A production application* $pa = (p, m)$ *for an object* G *consists of a graph transformation production* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *and a match morphisms* $m : L \to G$ *such that a direct transformation* $G \overset{p,m}{\Rightarrow} H$ *exists.*

A production application contains a graph transformation production that is supposed to be applied to an object G and the match morphism that determines how it will be applied. This represents a specific transformation and can be used for analysing the effects of the graph transformation production before applying it. Definition 1 assumes a direct transformation to exist. This is fulfilled if the match satisfies a certain condition, called gluing condition [38], which implies that the pushout complement of $K \xrightarrow{l} L \xrightarrow{m} G$ exists. This assures that the removal operation is well-defined.

2.2.3. Nested Conditions

This section gives an overview about the formalism of nested conditions. This formalism is able to describe constraints on objects of a category as well as application conditions that can be appended to graph transformation productions to restrict their applicability. The definitions in this thesis are based on the definitions in [19]. A more detailed description of nested conditions can be found there. Nested conditions are described on the basis of category theory and can be used in any $\mathcal{M}$-adhesive category.

A nested condition consists of a Boolean formula over model structures. The elements $true$, $false$, $\wedge$, $\vee$, $\neg$ and $\Rightarrow$ are defined as expected and can be used to build complex formulas. The quantors $\forall$ and $\exists$ are the only elements that are evaluated based on the model structure. Both consist of a pattern, described by a morphism, and an inner (nested) condition. The existence quantifier states that for at least one occurrence of the pattern in the model structure the inner condition has to be fulfilled, while the universal quantifier states that the inner condition has to be fulfilled for all occurrences. The complete definition of nested conditions is given in Definition 28 in Section A.3.

A nested condition can be satisfied by a morphism or an object. The satisfiability of objects can be used to formulate global constraints. These constraints can be evaluated on any object of the respective modelling language. The satisfiability of a morphism is important when formulating application conditions for graph transformation productions. Nested conditions can be used as left application conditions (evaluated before applying a production) or right application conditions (evaluated after the application of a production).

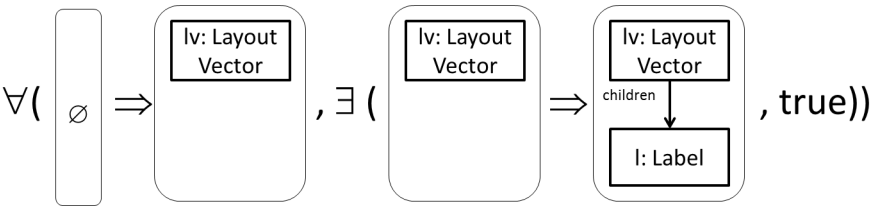


Figure 2.10.: Example: A nested condition.

Figure 2.10 contains an example for a nested condition. The condition consists of two nesting level. The outer level is an all-quantifier. It is interpreted as: without precondition (left hand side of the morphism is empty) for all layout vectors (right hand side of the morphism) the inner condition holds. The inner condition is an existence quantifier, requiring all layout vectors to contain a label. Thus, the overall condition requires each layout vector to contain a label.

2.2.4. Existing Theoretical Results

This section describes existing results in the context of $\mathcal{M}$-adhesive categories in combination with graph transformation and nested conditions. The Trollmann approach is based on the following main results:

- *parallel and sequential independence:* A notion for independence of graph transformation productions in parallel and sequential cases. The notion of parallel dependence is used to detect adaptation-adaptation conflicts. The relation between both types of independencies is described in the Local Church Rosser Theorem.
- *concurrency and Parallelism Theorem:* Enables to join multiple parallel applicable production applications into one according to the Parallelism Theorem.
- *generation of preconditions:* Enables to generate preconditions for a graph transformation production from a global nested condition.

The application of the Local Church Rosser and Parallelism Theorem require a finitary $\mathcal{M}$-adhesive category with $\mathcal{M}$-initial object [40]. These restrictions are described in the first Subsection. Subsequently, we present independence, the Local Church Rosser Theorem, the Parallelism Theorem and the generation of left application conditions.

Restrictions

According to Gabriel et al. [40] the application of general results in graph transformation theory to $\mathcal{M}$-adhesive categories requires two restrictions. The category needs to be a finitary $\mathcal{M}$-adhesive category and needs to contain an $\mathcal{M}$-initial object. The finitary restriction (See Definition 29 in Section A.1) requires all objects in the category to be finite. This requirement excludes objects with infinite substructures like graphs with an infinite number of nodes or edges. If this requirement is fulfilled the category is finite.

The finitary restriction is preserved by functor and comma category construction. Accordingly, if the underlying categories are finitary, the result of these constructions is also finitary [40]. We make use of this fact in the definition of graph diagrams over attributed typed graphs.

The second condition requires an $\mathcal{M}$-initial object. This is defined in Definition 30 in Section A.1. For an initial object a morphism to all other objects of the category exists. If all of these morphisms are in the class $\mathcal{M}$ this object is $\mathcal{M}$-initial.

These two conditions enable the application of the Local Church-Rosser, Parallelism, Concurrency, Embedding, Extension, and Local Confluence Theorem and also enables the generation of preconditions from global nested conditions [40].

Independence of Graph Transformation Productions and the Local Church Rosser Theorem

This section describes parallel and sequential independence of graph transformation productions and the Local Church Rosser Theorem. The definitions and theorem are contained in Appendix A.5.

Definition 31 in Appendix A.5 defines the condition for two graph transformation productions to be parallel independent. This condition requires the existence of two morphisms. Each of these morphisms maps the left hand side of one production to the result of the first application step of the other production. Intuitively this can be interpreted as all elements, required to apply one production, still being available after the deletion step of the other production.

The definition of sequential independence, given in Definition 32 in Appendix A.5, is similar to parallel independence. Again, two morphisms are required. One morphism maps the left hand side of the second production to the result of the first step of the first productions. This morphism assures that the elements the second production needs to be applied are not among the elements added by the first production. The second morphism maps the right hand side of the first production to the result of the first step of the application of the second production. This assures that the second production does not remove any element the first production requires to be applied.

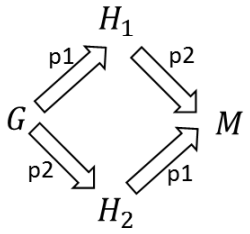


Figure 2.11.: Illustration of the Local Church-Rosser Theorem.

The Local Church-Rosser Theorem is shown in Theorem 11 in Appendix A.5 and illustrated in Figure 2.11. This theorem states that two parallel independent productions $p1$ and $p2$ can be applied in any order yielding the same result and that each of the orders are sequential independent. It also states that two sequentially independent productions are also parallel independent.

The notion of parallel independence and the Local Church-Rosser theorem are the basis for the detection of adaptation-adaptation conflicts in the Trollmann approach.

Parallelism Theorem

The effect of two parallel independent productions can be merged into a single production that encompasses the changes of both productions. In this section we describe the notion of parallel productions and the parallelism theorem. The formal definitions are provided in Appendix A.6. The definition of a parallel production is based on an adhesive HLR

system. An adhesive HLR system consists of an $\mathcal{M}$-adhesive category $(C, \mathcal{M})$ and a set of graph transformation productions.

The construction of a parallel production is described in Definition 33 in Appendix A.6. The parallel production consists of the coproduct of each component of the two productions. In set-based categories the coproduct in sets is a component-wise disjoint union. Given a set of productions, the left hand side, right hand side and context objects are united and form the parallel production.

It is also possible to extend the notion of production application to a parallel production application. This is described in Definition 2. It can be constructed if the productions contained in the production application are parallel independent. The production in the parallel production application is the parallel production for the contained production applications. The match of the parallel production is derived from the matches of the joint productions. According to the universal property of the coproduct, a match morphism with this property always exists.

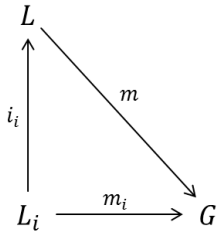


Figure 2.12.: Construction of the match morphism for parallel production applications using the universal property of the coproduct.

Definition 2 (Parallel Production Application)**.** *Given a set of n production applications* $p_i = ((L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i), m_i)$ *with* $i \in \{1, ..., |Prod|\}$ *that are pairwise parallel independent, the parallel production application* $p = ((L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R), m)$ *consists of the parallel production* $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *of all* $(L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i)$ *and match* m*.* m *is induced by the universal property of the coproduct for* m_i *as shown in Figure 2.12.*

The parallelism theorem is given in Theorem 12 in Appendix A.6. The theorem states that instead of applying a sequence of transformations, the parallel production can be applied. In combination with the Local Church-Rosser Theorem this means that instead of two parallel independent productions the parallel production can be applied.

Since the approach and proof sections often consider the result of the application of graph transformation productions we define a shortcut notation for it. This shortcut defines the operator *apply* to denote that a set of parallel independent production applications is applied to the same object. This function is defined as follows:

Definition 3 (apply)**.** *Given an object* G *and a set of parallel independent production applications* $Prods$ *that are matched to* G*,* $apply(G, Prods)$ *denotes the result of the application of the parallel production application for* $Prods$ *to* G*.*

Generation of Left Application Conditions from Nested Conditions

As shown by Habel and Pennemann [19] global nested conditions can be transformed into right application and left application conditions. This technique can be utilised to generate left application conditions for a global condition. The two theorems that describe this behaviour are contained in Appendix A.7.

Theorem 13 describes that global conditions can be converted into right application conditions such that the result of the transformation fulfils the global condition exactly when the comatch fulfils this right application condition.

It is also possible to transform right application conditions into equivalent left application conditions. This is stated in Theorem 14. In combination both theorems enable the transformation of a global nested condition into a right application condition and then into a left application conditions. The advantage of this is the option to check the left application conditions before applying the production. The construction operations for both theorems can be found in [19].

2.2.5. Attributed Typed Graphs

The formalism defined in the Trollmann approach is based on attributed typed graphs. It makes use of this formalism to represent models and their modelling languages. Attributed typed graphs have already been used to describe a variety of modelling languages. A detailed definition of attributed typed graphs is given in [41]. The main definitions are contained in Appendix A.8. Attributed typed graphs are based on attributed graphs, E-graphs, and plain graphs. This section describes these formalisms.

The basis for attributed typed graphs are graphs as defined in Definition 34 in Appendix A.8. A graph consists of a set of nodes and a set of edges. Each edge connects one node with one other node.

E-graphs form the basis for annotating graphs with data. Definition 35 in Appendix A.8 defines E-graphs. An E-graph is a graph that, in addition to its graph nodes, contains a set of data nodes. These data nodes represent pieces of data that can be annotated in the graph. These annotations are described as data edges. In addition to the normal graph edges an E-graph contains two types of data edges. Node attribute edges relate graph nodes and data nodes and thus annotate data to the nodes of the graph. Similarly, edge attribute edges relate the graphs edges with data nodes in order to annotate data to the edges of the graph. Accordingly, an E-graph is a graph with data annotated to its nodes and edges.

A morphism on E-graphs consists of mappings between the respective graph nodes and edges and between the data nodes and edges. This mapping has to be consistent with the source and target of the graph and attribute edges. Accordingly, the morphism can only map between graphs that are compatible in their E-graph structure and in the annotated node and edge attributes.

One typical way to represent data structures are algebraic signatures and algebras. Attributed graphs are a combination of this concept and E-Graphs. This combination is established via the data nodes in the E-Graph, which consist of elements from the data

in the algebra. The definition of attributed graphs and morphisms on them is given in Definition 36 in Appendix A.8. Morphisms map between attributed graphs that contain algebras over the same signature. They are E-Graph morphisms that contain an additional algebra homomorphism that describe the relations between the data elements.

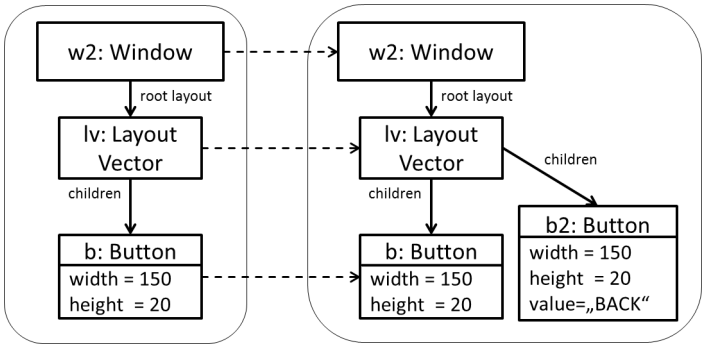


Figure 2.13.: Example: An attributed graph morphism.

Figure 2.13 shows an example for an attributed graph morphism. The morphism is indicated by dashed arrows. It maps an attributed graph representing a user interface window with some layout elements to another one, containing the same elements and an additional button. The dashed arrows indicate the mapping of nodes. In addition, the edges of both graphs as well as attributes are mapped.

One important concept in higher level languages is the concept of typing. It enables the definition of specific types of elements. This concept is also useful on attributed graphs as it enables the definition of node and edge types and the definition of structural restrictions, concerning which edge types can connect which node types and which attribute types can be annotated to them. Definition 37 in Appendix A.8 describes a typing concept on attributed graphs leading to attributed typed graphs (ATGs). The types are specified by a dedicated attributed graph, called the type graph. The nodes, edges and attributes of this graph represent node, edge, and attribute types. The typing relation is established via a morphism between the typed graph and the type graph. Morphisms between attributed typed graphs need to preserve all types.

The example in Figure 2.14 shows one attributed typed graph and its typing morphism. The attributed graph on the right hand side is the type graph and defines node types like $Window$, edge types like *root layout* and attribute types like *width*. All elements in the ATG on the left hand side are mapped to the corresponding elements in the type graph. As a shortcut notation this mapping is indicated by the node types and the names of the attributes and edges.

In many modelling languages the concept of type inheritance allows for an easier definition of concept that are shared amongs a group of types. An inheritance concept, as described in Definition 38 in Appendix A.8, can enable a more convenient description of a modelling language in the type graph. This is achieved by a set of inheritance edges, describing the inheritance structure for node types. Morphisms have to map all nodes and edges in accordance with their type. Figure 2.15 shows an example for a type graph

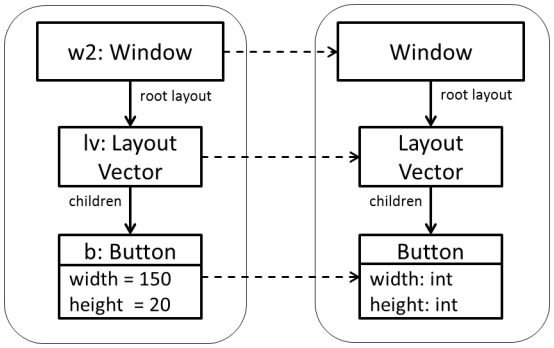


Figure 2.14.: Example: Typing in attributed graphs.

with inheritance. The node types *Window*, *Layout Vector* and *Button* all inherit from the type *UI Element* and thus can all have the attributes *width* and *height*.

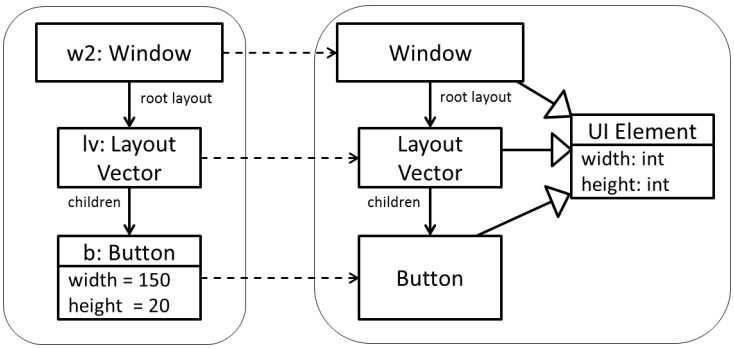


Figure 2.15.: Example: A type graph with node type inheritance.

For a given attributed type graph (without inheritance edges) there is a category $AGraphs_{ATG}$ of all attributed graphs typed over ATG [41]. This category in combination with the class $\mathcal{M}$ of all injective attributed typed graph morphisms with isomorphic data type is an $\mathcal{M}$-adhesive category [38]. The category of ATGs with inheritance can be formally founded in this category. Ehrig et al. show that a flattening construction can be used to map attributed typed graphs with node type inheritance to normal attributed typed graphs [38].

Regarding the required restrictions from Section 2.2.4 the category $(AGraphs_{ATG}, \mathcal{M})$ is not finitary and does not have $\mathcal{M}$-initial objects. The category is not finitary as it contains attributed typed graphs with an infinite number of nodes or edges. The category needs to be restricted to finite objects to apply the existing results. This restriction implies that it is only possible to represent models with a finite structure in ATGs.

According to Gabriel et al. [40] the category $(AGraphs_{ATG}, \mathcal{M})$ does not have an $\mathcal{M}$-initial object. Its initial object, the empty graph with the term algebra as data types, is not $\mathcal{M}$-initial as its initial morphisms do not have isomorphic data components and thus are not in $\mathcal{M}$. To fulfil this property the category needs to be restricted further.

In the Trollmann approach we make these two restrictions. This allows us to extend attributed typed graphs to graph diagrams and apply the existing theoretical results.

2.3. Summary

In this chapter we described model driven engineering and formal results from $\mathcal{M}$-adhesive categories as a foundation for this dissertation. Model driven engineering is the research area our approach is placed in. We described several concepts from this area including models, meta models, model transformation, model driven engineering and models@run.time. These concepts where defined based on a set of model relations that will be used to describe the problem and conceptual approach on the basis of general model driven engineering in the next sections. $\mathcal{M}$-adhesive categories form the basis for the Trollmann approach to conflict detection. This approach is based on several existing formal definitions that are presented in this chapter. In the next chapter we define the problem statement in more detail based on the understanding of MDE we presented in this chapter.

3. Problem Statement

The problem approached in this thesis is in the scope of run time assurance of self-adaptive software systems. One of the aspects that need to be assured is that the adaptation process is consistency-preserving. This assures that the running software system is always in a consistent state, despite adaptations. Design time analysis methods for the adaptive behaviour of software systems usually aim to cover all or a large part of the space of steady-states programs that can be reached via repeated adaptation. However, design time analysis methods are only sufficient if this state space is fully known at design time and can be analysed completely in a reasonable amount of time.

Run time analysis methods can be used to complement these techniques in cases where not all of the state space can be covered at design time [10]. The problem approached in this thesis is the detection of adaptation conflicts at run time and the provision of suitable information about the detected conflicts. The results can be used by a conflict resolution mechanism to resolve the detected conflicts.

Run time methods usually have more knowledge about the current state but harder time constraints than design time methods. They can be used to assure that the current and potential next states of the SAS are valid. The aim of the Trollmann approach is to analyse whether a set of adaptations that are supposed to be applied lead to a deterministic and consistent result. This analysis is based on the following information:

- S - *the current steady-state program*: The steady-state program that represents the current state of the software system. This steady-state program can also contain a representation of its context-of-use [42].
- Ads - *the currently activated adaptations*: A set of adaptations. These adaptations have been selected to be applied to S based on the current situation.

If there are no adaptation conflicts all adaptations in Ads can be applied to S and any sequence of these adaptations leads to the same steady-state program. If this is not the case one of two types of adaptation conflicts has occurred. Both types are described in detail in Section 3.1. It is also possible that an adaptation which already has been applied previously is in conflict with an adaptation that is in Ads. This is called a sequential conflict as opposed to a parallel conflict where all conflicting adaptations are contained in Ads. Section 3.2 describes this distinction.

The Trollmann approach enables the detection of adaptation conflicts. The purpose of conflict detection is to provide information that can be used as a basis for conflict resolution. We indicate potential approaches for conflict resolution in Section 3.3 to derive the information they require from a conflict detection mechanism. We present the requirements drawn from these considerations in Section 3.4. Section 3.5 summarizes and concludes the chapter.

3.1. Adaptation Conflicts

In this section we describe the two types of adaptation conflicts. A conflict-free situation for three adaptations is shown in Figure 3.1. All three adaptations Ad_1, Ad_2 and Ad_3 can be applied to the initial steady-state program S. The result is a consistent steady-state program S_{123}. The same result is reached for all six orders of adaptations.

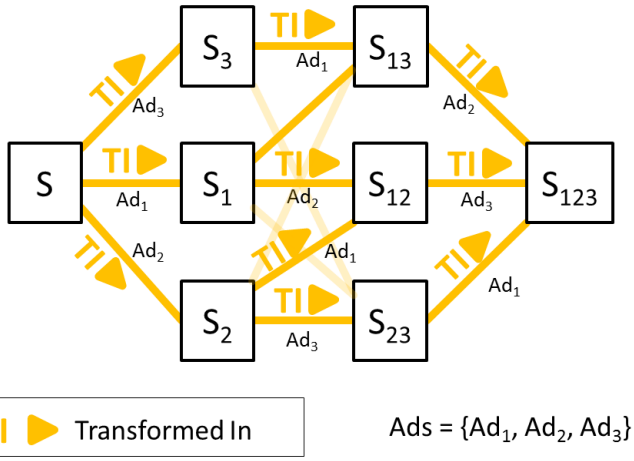


Figure 3.1.: Application orders of three adaptations.

The following two types of conflicts can occur in this setup:

- *Adaptation-Adaptation Conflict*: It is not possible to reach the same end-state for all orders of adaptations.
- *Adaptation-Consistency Conflict*: The reached end-state or one of the intermediate states is inconsistent.

An adaptation-adaptation conflict is a situation in which different orders of the same set of adaptations lead to different results. Such a situation can occur when adaptations overlap in the aspects of the steady state program they adapt, leading to a situation where changes of one adaptation are overwritten by another adaptation. It can also happen that adaptations influence each other in a way that some of the adaptations are not applicable at all. This is also considered to be an adaptation-adaptation conflict. An adaptation-adaptation conflict represents a non-determinism in adaptive behaviour since the end result varies depending on the order of adaptations.

Figure 3.2 shows an example of two adaptations that are in adaptation-adaptation conflict. These adaptations concern the graphical user interface of a software application. In its original state S the user interface displays two elements A and B in a Tablet sized screen. A button labelled *next* can be used to switch to the next screen. Two adaptations change this user interface. Adaptation Ad_1 distributes the elements A and B to two Smart Phone sized screens where the screen for B is reachable via *next* button

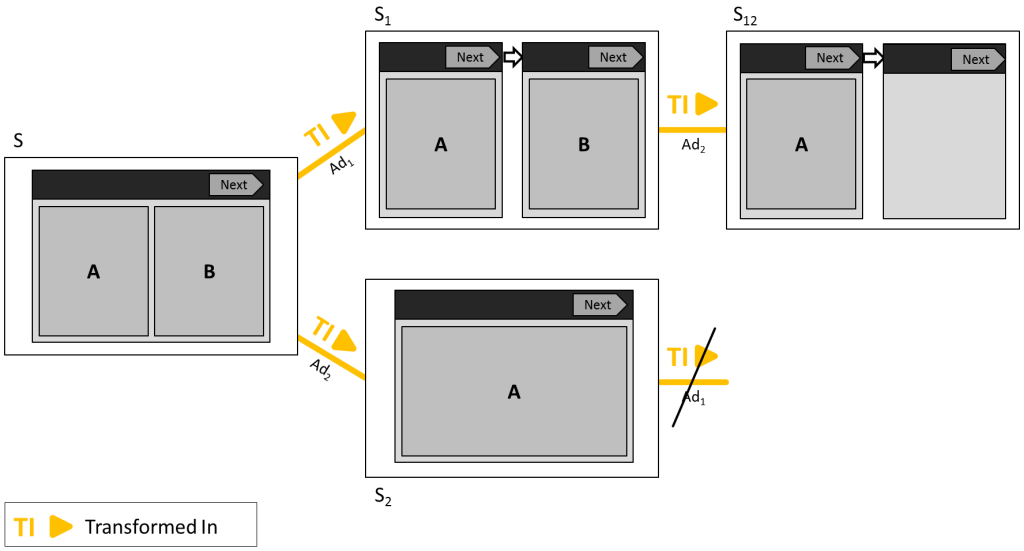


Figure 3.2.: Example: An adaptation-adaptation conflict.

from the screen for A. Adaptation Ad_2 removes the element B from the screen. Each of the two adaptations results in a valid state of the user interface (S_1 and S_2). However, applying them in different order does not yield the same result. Applying Ad_2 after Ad_1 leads to state S_{12}. In this state the screen for B is empty, because this element has been removed by the second adaptation. It is not possible to apply adaptation Ad_1 after Ad_2 because the element B has been deleted and can therefore not be moved to a new screen. These two adaptations are in adaptation-adaptation conflict.

An adaptation-consistency conflict occurs when some of the states in the adaptation sequence are not considered a consistent steady-state program. This requires some notion of consistency that can be checked on the steady-state program. In the easiest form all possible steady-state programs can be considered consistent. In this case an adaptation-consistency conflict cannot occur. However, it may be necessary to impose additional requirements on the steady-state program to assure correct behaviour of the software system. These requirements can originate from various sources. For example, two kinds of requirements can be distinguished:

- *Framework Requirements*: These are requirements that are imposed by the runtime framework to assure that the software system can run. An example is the existence of a main method in an executable Java program without which the program cannot be started.

- *Application Specific Requirements*: These are requirements which are specific to the software system and often stem from its specific domain. For example, a program that explains a process to the user may need to assure that all steps are described in the right order.

These requirements are not comprehensive. e.g. requirements can also be imposed by the user or by the platform the system is running on. For conflict detection the source of the requirements is inconsequential as long as the fulfilment/nonfulfillment of these requirements can be detected. However, the conflict resolution may treat different kinds of requirements differently.

The example for an adaptation-adaptation conflict in Figure 3.2 can also be considered an example for an adaptation-consistency conflict. We use the following consistency requirement: *"There are no empty screens"*. This requirement is fulfilled in states S, S_1 and S_2. However, it is not fulfilled in S_{12}. Thus, this state is not considered a valid state for this requirement and has to be avoided, even if the adaptation-adaptation conflict would not exist.

In some cases intermediate steady-state programs can be inconsistent even if the result of the adaptation is consistent. In some frameworks this is not a problem if the end result is consistent. On the other hand, intermediary steps might be interpreted for a short duration while the next adaptation is prepared and thus need to be consistent. The developer of a run time framework has to decide whether intermediary steps need to be consistent or not. In our analysis we assume that they need to be because this is the harder case of the two.

Adaptation-adaptation and adaptation-consistency conflict can also occur in combination, i.e., a situation can occur where adaptations are dependent and some of the adaptation states are inconsistent. The running example is such a case. Such situations can prove complex to resolve and require knowledge about the cause of the conflict. Furthermore, adaptations in Ads can be in conflict with adaptations that have already been applied. This is further elaborated in the next section.

3.2. Parallel and Sequential Conflicts

Most adaptation conflicts are the effect of the combination of multiple adaptations. Since the current steady-state program has been generated by applying a sequence of adaptations to a starting state, the current adaptations can also be influenced by any of the already applied adaptations. Figure 3.3 shows a situation where in addition to three currently active adaptations two adaptations Ad_{-1} and Ad_0 have already been applied starting from an initial state S_{init}.

To detect these situations the setup for an analysis should contain a set of previously applied adaptations. Thus, in addition to the current steady-state program S and the set of currently active adaptations Ads the analysis should be able to access the sequence of previously applied adaptations:

- *Seq - The sequence of applied adaptations*: A sequence of adaptations leading from the initial state of the software system to S.

Based on whether a set of conflicting rules is in Ads or in Seq we can distinguish parallel and sequential conflicts. An adaptation conflict is purely parallel if all conflicting adaptations are in Ads. If at least one of the adaptations is contained in Seq the conflict is sequential.

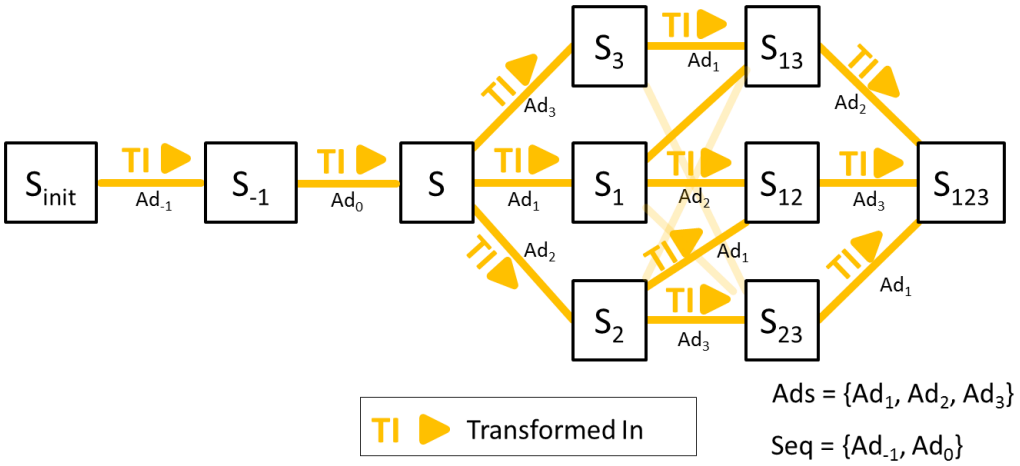


Figure 3.3.: Application orders of three adaptations and two previously applied adaptations.

In the example from Figure 3.2 a sequential conflict would occur if the adaptation Ad_2 has already been applied. In this case the current state of the user interface is S_2 and the adaptation Ad_1 cannot be applied because of the previous adaptation Ad_2.

In practice it is not always possible to revert all adaptations. Some may be imposed by a change in the context-of-use. Others may not be reversible for technical reasons, e.g., because other adaptations depend on them. If these adaptations cannot be reverted anyway it can make sense to restrict the analysis to the adaptations that can actually be reverted instead of all previous adaptations.

The analysis of parallel and sequential conflicts is related in the sense that it should be able to find sequential conflicts by executing an analysis for parallel conflicts based on the starting state S_{start} of the software system and an enhanced version of adaptations $Ads' = Ads \cup Seq$ that contains all adaptations that have been applied in S. In cases where adaptations cannot be reverted these can be applied to S_{start} and removed from Ads' in order to disregard them in the analysis.

3.3. Conflict Resolution

The purpose of detecting adaptation conflicts at run time is to enable the run time framework to resolve these conflicts. The conflict resolution poses requirements to the conflict detection in form of the information it requires. This section is intended as an overview to give the reader a rough idea about potential conflict resolution strategies to derive which information should be provided by a conflict detection mechanism.

Given an adaptation conflict the role of the conflict resolution mechanism is to define an adaptation sequence that leads the software system from S into a steady-state program that is considered optimal in the current situation. This sequence can contain

other adaptations than those in *Ads*. Since *Ads* contains the adaptations that were originally intended to be applied the intuition is that the steady-state program that results from the conflict resolution is as close as possible to the application of all adaptations in *Ads* and thus as many adaptations from *Ads* as possible are applied. However, a conflict resolution mechanism might also remove adaptations that are redundant or alter adaptations to avoid conflicting changes and while maintaining non-conflicting changes.

Depending on the type of conflict the set of adaptation sequences and the selection strategy can vary. Some potential conflict resolution mechanisms for the two types of adaptation conflict are the following:

- *Adaptation-Adaptation Conflict*: To resolve an adaptation-adaptation conflict a conflict resolution mechanism needs some mechanism that selects between the potential adaptation results. This selection can take into account different end states or even intermediate states (for example in cases where it is not possible to apply all adaptations). The mechanism can also choose to apply additional adaptations, for example as a substitution for an adaptation that cannot be applied due to its dependencies, or manipulate the adaptations to avoid conflicts.

- *Adaptation-Consistency conflict*: An inconsistent state can either be avoided or repaired. An avoiding mechanism can choose another result or intermediate state of the adaptation similar to the resolution of adaptation-adaptation conflicts. A self-healing approach, like the one proposed by Ehrig et al. [43], is an example for an approach that repairs inconsistencies. This approach contains a set of additional adaptations that can be applied to repair an inconsistent state. Both approaches can also be combined to select a state that is easy to repair before repairing it. Again, adaptations may be altered to better fit the purpose of the resolution strategy.

In our example of user interface adaptations from Figure 3.2 the conflict resolution could select between one of the states S, S_1 and S_2 since these states fulfil the condition that requires non-empty screens. In addition, the resolution might use additional adaptations. For example, the state S_{12} could be repaired by an adaptation that removes the empty screen, resulting in a Smart Phone sized screen that only contains element A. The same result can be achieved by partially applying adaptation Ad_1 to state S_2. Ad_1 moves the element B to a new screen. In addition, it changes the size of the original screen and the element A. This second change can still be applied, even after the element B has been removed by Ad_2. This change results in a Smart Phone sized screen that only contains the element A.

The selection of a solution can be done automatically or by a human (for example the user). Both strategies require knowledge about the conflict with regards to the existence and reasons for the conflict to find a suitable solution or to convey the reason for the conflict to the user. The next section describes which information the conflict detection mechanism should provide to the conflict resolution mechanism.

3.4. Information for Conflict Resolution

For a sophisticated conflict resolution as much information as possible about the conflict ought to be available. The conflict resolution approach by Badr and Reilly can serve as a basis to identify the required information [44]. In the scope of conflict resolution between actions in agent-based service oriented software they pose three questions that represent information that is required to solve the conflict. According to these questions, the conflict resolution requires information about:

- the type of detected conflict (i.e., is the conflict an adaptation-adaptation or an adaptation-consistency conflict?)
- the actions involved in a conflict (i.e. the involved adaptations)
- elements in the current state of the software system that are involved in the conflict

In the scope of adaptation conflicts the type of the conflict is either an adaptation-adaptation or adaptation consistency conflict. The involved actions are adaptations. Usually an adaptation consists of a set of elementary adaptations. A conflict can be caused by one or more, but not necessary all, elementary adaptations. Accordingly, this question can be split into two for the general adaptations as well as the specific adaptation actions involved in the conflict. The last question refers to the elements in the current model that are involved in the conflict.

Solutions for both adaptation-adaptation and adaptation-consistency conflicts often require finding suitable subsets of the set of all adaptations that can be used as basis for conflict resolution. A fifth question should be answered to enable this derivation. This question asks which subsets of adaptations are conflict-free. This information can be used to derive immediate solutions that do not apply all adaptations.

Given the current steady state program S, a set of adaptations Ads and the set of already applied and reversible adaptations Seq we formulate the following five questions:

- *Which conflicts do occur?* Which adaptation-adaptation and adaptation-consistency conflicts occur?
- *Which adaptations are in conflict?* Which adaptations from Ads and Seq cause which conflict?
- *Which elementary adaptations lead to the conflict?* Which elementary adaptations cause which conflict?
- *Which model elements are involved in the conflict?* Which elements of the current state S are in conflict?
- *Which combinations of adaptations can potentially be applied?* Which adaptations in Ads and Seq can be applied without any conflict?

These questions are answered in Chapter 6 where we discuss the conflict detection algorithm for adaptation-adaptation and adaptations-consistency conflicts.

A conflict resolution mechanism can also use additional adaptations to repair conflicts. Of course, the repairing adaptations can also be in conflict with the original adaptations. Thus, it is necessary to integrate them into the analysis of adaptation conflicts. If they are contained in the set of analysed adaptations in the Trollmann approach the five questions can also provide information about conflicts involving repairing adaptations.

3.5. Summary

In this chapter we provided a detailed discussion about adaptation conflicts. We distinguished two types of conflict: adaptation-adaptation conflicts and adaptation-consistency conflicts. Adaptation-adaptation conflicts are situations in which the application of one adaptation influences the applicability of another adaptation. Adaptation-consistency conflicts are situations in which the end-result or any of the intermediate states of the adaptation are not consistent. In addition, we discussed the difference between parallel conflicts, which involve adaptations that are to be applied at the same time, and sequential conflicts, which involve adaptations that are already applied. From a general discussion of conflict resolution mechanisms we derived requirements for a conflict detection approach in the form of a set of questions that need to be answered.

In the next chapter we discuss research approaches that are related to this problem and its solution.

4. Related Work

This chapter presents related work. First, we discuss existing approaches and classifications in the area of self-adaptive software systems to relate our approach to these approaches (cf. Section 4.1). Subsequently we review formalisms that can be used for the formalization of models, adaptations and consistency requirements (cf. Section 4.2). Finally, we present related work for the detection of conflicts for adaptations (cf. Section 4.3) and conclude the chapter (cf. Section 4.4).

4.1. Adaptations In Software Engineering

Adaptations can apply to a variety of aspects of a software system. Examples are the user interface, behaviour or resource consumption. An example that is close to the case study we use in this thesis is the notion of ubiquitous user interfaces defined by Blumendorf [3]. These user interfaces fulfil five properties. These properties define that a user interface can be shaped to fit the current situation or properties of the user (shapeability), shared between multiple users (shareability), merged with another user interface (mergeability), distributed among multiple devices (distribution) or presented via different input and output modalities (multimodality). The adaptation of user interfaces is also focus of the research area of user interface plasticity [45]. Adaptations for other aspects of a software system have also been subject to research. For example, Zhang and Cheng [46] aim to adapt the execution logic and several approaches aim to implement adaptive architectures [47]. The Trollmann approach abstracts from the adapted aspects by assuming a model that represents these aspects. Since the approach aims to be independent of modelling languages, it is not dependent on what aspects of the software system the model represents.

Execution and adaptation both change the current state of the software system. Zhang et al. [4] use the notion of a steady-state program for describing a non-adaptive software system. They define an adaptation as a transformation from one steady-state program to another. In this notion a steady-state program can be seen as the normal execution of a software system and an adaptation as a change that goes beyond that execution.

This abstract view is useful for considering adaptation on a high level of abstraction, e.g., when reflecting on the state space of adaptations. However, it does not regard the process of adaptation and its control and assurance. These aspects are in focus of autonomic computing [48], where control loops from control theory are used for the management of software systems. Cheng et al. [2] argue for applying the generic model of a control loop [49] to self-adaptive systems and enhancing it with control and data flow. The generic model of a control loop contains four phases. In a *collection* phase data about the system and its environment is collected. This data is analysed in the

analysis phase. Based on this analysis the system *decides* for an adaptation and finally executes the adaptation during the *act* Phase.

Researchers from IBM proposed to make such feedback control loops explicit as an *autonomic element* in self-managing systems. They describe the MAPE-K loop, which consists of the phases *monitoring*, *analysis*, *planning* and *execution*, which access a common *knowledge base* [50, 51]. This feedback loop can be applied for controlling dynamic adaptation of software systems [52, 53]. Villegas et al. defined a reference model incorporating feedback loops for the management of control objectives, adaptation control and context management as well as interactions between these loops [14]. The results of the run time assurance group of the Dagstuhl Seminar on models@run.time [10] also stress the need to have a dedicated feedback loop for adaptations and to incorporate run time models into this loop. The detection of adaptation conflicts can be part of the *analysis* step in such an adaptation feedback loop. It provides information for a conflict resolution that is located in the *planning* step.

The ISATINE framework proposes a different feedback loop specifically for the adaptation of user interfaces [54]. The feedback loop here is between the goals of an adaptation and its execution and aims to enable an involvement of the user. The steps in this feedback loop are forming the *goal* of the adaptation, forming the *intention* to reach the goal, the *action specification* that decides how the adaptation is performed, the *execution* of these actions, the *perception* of the results of the actions, the *interpretation* of these results and the *evaluation* of the adaptation result against the original goals, that could lead to new goals. In this framework the detection of adaptation conflicts could be located either in the *action specification*, if conflicts are taken into account while selecting adaptations, or during the *execution* of these actions.

Salehie and Tahvildari present a survey of the research landscape and challenges in self-adaptive software systems [55]. The MAPE-K loop is seen as central and used to structure this research area. The survey also mentions Self-* properties, which describe properties of self-adaptive systems. Several sets of properties have been proposed over the years. An initial set has been defined by IBM in the scope of autonomous computing [48]. These properties can be decomposed into three layers. On the top layer general properties are situated. For self-adaptiveness several of these properties are important. For example, Kephart and Chess describe the properties *self-managing*, *self-governing* and *self-maintenance* [51]. Hellerstein discusses the use of control theory for systems with the *self-managing* property [56].

The second layer of self* properties contains more specific properties that denote the reason for adaptations. IBM describes the following four properties on this level [48]:

- *self-configuring* means that a system changes its own configuration. This can be caused by technical processes like installing or updating components.
- *self-healing* means that a system detects faults in itself (e.g., a broken component) and adapts itself with the purpose of preventing failures.
- *self-optimizing* enables a system to adapt itself to optimize its properties. Examples for such properties are memory usage or response times.

- *self-protecting* means that a system is able to protect itself from security breaches and recover from the caused problems.

All of these four properties represent reasons why self-adaptive and thus adaptive systems are needed. The properties *self-protecting* and *self-healing* can be interpreted in the scope of adaptation-conflicts, denoting that a system is able to either protect itself from adaptation conflicts or heal the damage after such a conflict occurred.

The third layer of Self-* properties contains low level properties like self-awareness or context-awareness that form the technical basis for implementing the high level properties. Again, we can map properties like self-awareness to the adaptations itself and denote that the system needs to be aware of adaptation conflicts.

Cheng et al. stress that *"... models need to be built and maintained at run time..."*[2] to implement and assure the correct behaviour of self-adaptive systems. If run time models are available an adaptation can be implemented as a transformation of models. Such transformations manipulate the structure of the model to cause an adaptation of the software system. Our approach can be applied in such an approach to detect adaptation conflicts in the aspects represented by the run time models.

Salehie and Tahvildari [55] define a taxonomy of self-adaptive systems. Their taxonomy is based on the classifications by Oreizy et al. [57] and McKinley et al. [5]. The complete taxonomy contains four main categories, which denote the *object to adapt* as the part of the software that is subject to changes, *realization issues* in the technical approach for adaptation, *temporal characteristics* of the adaptation and *interaction concerns* with regards to users of the adapting system. For this thesis the realization issues are of special interest. The relevant classifications for this category are:

- *static and dynamic decision making:* This facet describes whether the decision on when to adapt and which adaptation to choose is already prescribed by the developer or is generated dynamically at run time, for instance due to some learning strategy. Dynamic decision making is the more complicated case for design time analysis as less information is available at design time. This presents a strong motivation to do analysis of adaptation conflicts at least partially at run time.

- *closed and open adaptation:* This facet describes whether new adaptations can be added at run time (open Adaptation) or whether the set of potential adaptations is predetermined and cannot be subject to change (closed adaptation). Newly added adaptations can for example originate in newly installed software components, in user configuration or in dynamic decision making that creates new adaptations. Since open adaptation approaches can contain adaptations that are not foreseeable and thus not analysable at design time they are an even stronger motivation to do analysis at run time.

- *model-based and free adaptations:* One frequently taken approach to the implementation of adaptive systems is the use of models and their analysis and adaptation in place of the software system. There are also model-free approaches to adaptation. E.g., switching one component with a different implementation. The Trollmann

approach relies on a model-based adaptation approach. The models are required as basis for the analysis of adaptation conflicts.

Salehie and Tahvildari [ibid.] also name 6 questions that need to be answered about each adaptation in a self-adaptive software system. This set is based on the questions described by Laddaga [58]. These questions contain the questions *Where?* and *What?* to identify which part of the system and which specific detail the adaptation changes, the question *When?* to identify when, how often and how long the adaptation will occur, the question *Why?* to identify the reasons for the adaptation, the question *Who?* to identify whether a human is the cause for the adaptation and if so, which human and the question *How?* to identify how the adaptations are implemented. The Trollmann approach is restricted to approaches that manipulate runtime models of the software system to cause adaptations. This means, the *Who?*, *Why?* and *When?* is not restricted. The *Where?* and *What?* aspects are restricted to parts of the software system that are represented by models and can be changed by manipulating these models. One goal of the adaptation conflict analysis is to answer these two questions in case of a conflict to determine *what* changes are in conflict and *where* in the model the changes take place. In addition, the question *How?* is restricted to approaches where the models are adapted to cause changes in the software system.

Kephart and Walsh define three policies for autonomic systems: Action, Goal and Utility-function policies [59]. Hussein et al. [60] apply these three policies to mechanisms that select adaptations in a self-adaptive system. *Rule-based mechanisms* specify a condition on the context and an adaptation that is to be executed when the condition is fulfilled. *Goal-based mechanisms* define the goal of the adaptation and determine the required adaptation actions to fulfil this goal. *Utility-based mechanisms* specify a utility function that can be used to assign a value to a steady-state program to find the most suitable next one. According to Hussein et al. [ibid.] the goal- and utility-based approaches can be verified but are difficult to build and suffer from state space explosion in domains with a large state space. Balasubramanian et al. [61] give a mathematical formalization of these policies that can be used to select one policy and reason about solution qualities. The Trollmann approach is not restricted to any of these policies. It operates on the basis of the intended adaptation actions. These actions could represent the activated adaptations in the rule-based policy or the potential adaptations that are available for goal- or utility based reasoning.

4.2. Formalisms

The foundation for our analysis of adaptation conflicts is a formalism that is able to represent models, relations between models, adaptations of models and consistency requirements based on the structure of these models. The formalism should be able to represent a variety of modelling languages and relations between models of different modelling languages. In this section we present techniques and languages that are potential options for this formalism. First, we discuss formalisms for representing models and model relations (cf. Section 4.2.1). Subsequently, we review formalisms for expressing

consistency conditions (cf. Section 4.2.2). Finally, we describe formalisms for representing adaptations (cf. Section 4.2.3). In a summary (cf. Section 4.4) we indicate which combination of languages we intend to use for the Trollmann approach.

4.2.1. Model Languages

This section describes languages for representing models. Our approach poses three requirements to these languages:

- **Language Independence:** The language needs to be able to represent models of an arbitrary modeling language.
- **Multiple Models:** The language needs to be able to represent an arbitrary number of models at the same time.
- **Model Relations:** The language needs to be able to represent relations between coexisting models.
- **Formal Basis:** The language should be defined in a manner that allows the application of formal analysis methods for finding adaptation conflicts.

The state of the art approaches presented in this section are summarized in Table 4.1.

The formalism of attributed typed graphs, as described in the fundamentals (cf. Section 2.2.5), is a widespread technique for representing models. Most modelling languages can be represented as a typed graph structure with node and edge attributes. One indication for this is that an XML tree, which is a format widely used for storing models, can be represented by an ATG. Thus, we consider ATGs to be a good basis to represent models of most, if not all, software modelling languages. They have already been applied to represent several modelling languages. Examples are UML class and sequence diagrams [74] and different kinds of petri nets [75]. ATGs can be combined with graph transformation and similar formalisms. Attributed typed graph transformation systems [76] and other concepts have already been used to represent modelling languages for purposes of formal analysis. For example, Taentzer et al. base an analysis of conflicts in model versioning on this combination of formalisms [77]. Accordingly, attributed typed graphs are a suitable formal basis for the detection of adaptation conflicts.

One attributed typed graph usually represents a single model. Together with some collegues I explored the idea to merge multiple related models into one attributed typed graph and add relations in the form of additional edges [78]. This approach requires additional effort to distinguish the models in the ATG. Although, models as well as their relations can be represented in this approach, the unison in one big model can become cumbersome to handle. A clear separation of concerns between the related models would be a great improvement in model management and reduce search times as it enables to focus on the relevant models and relations.

The formalism of triple graphs [62] is a promising extension of attributed typed graphs for the representation of multiple models. In triple graphs two models are represented as attributed typed graphs and a relation between them is represented as an additional ATG

Table 4.1.: Evaluation of approaches for representing models and model relations. Uncertain properties have been left blank.

Language	**Language Independence**	**Multiple Models**	**Model Relations**	**Formal Basis**
Attribted Typed Graphs [41]	✓	∼	∼	✓
Triple Graphs [62]	✓	X	✓	✓
Distributed Graphs [63]	✓	✓	∼	∼
Traceability (Galvao and Goknil) [64]	X	✓	✓	X
Traceability (Ramesh and Jarke) [65]	✓	✓	✓	X
Traceability (ATL) [66]		X	X	
Traceability (KerMeta) [67]		✓	X	
Mappings (Sottet et al.) [68, 69]	✓	✓	X	X
Mappings (Wolfe et al.) [70]	X	X	X	X
Meta Object Facility [12]	✓	✓	✓	X
UML formalization (Diskin) [71]	X	✓	✓	✓
Model Weaving [72, 73]	✓	✓	✓	X

that is matched into the two models by ATG morphisms. This enables the representation of two models and their relation while keeping the models clearly separated. However, the formalism is only defined for two models at a time and thus is severely restricted in the number of models that can be represented.

Distributed graphs [63] also implement the idea of using attributed typed graphs and morphisms between them to represent a set of models and their relations. Distributed graphs form a graph structure of ATGs and ATG-morphisms. The distribution property of this formalism has been used by Jurack and Taentzer to represent models that are distributed over multiple platforms and connected via some common interfaces [79]. Distributed graphs are closely related to the formalism of graph diagrams defined in this thesis. There are two main differences. Distributed graphs enable to change the structure of the diagram, meaning they allow adaptations to add and remove whole models whereas graph diagrams assume a fixed set of models and only allow changes of the structure of these models. Although, the option to change the set of existing models can be ignored if not required, it complicates the formalism. To our knowledge the $\mathcal{M}$-adhesive properties have not been proven for distributed graphs. Accordingly, the existing results for these categories (cf. Section 2.2.4) cannot be applied directly. The second difference to graph diagrams is the requirement for morphisms in the diagram to commute. This is not a necessity in models@run.time since relations between model elements can express various aspects which do not necessarily commute with each other. Accordingly, distributed graphs are too restricted for representing arbitrary relations.

Distributed graphs have been extended with a containment and inheritance concept to represent distributed EMF models [80]. The addition of a containment concepts leads to the non-existence of certain pushouts, which complicates the application of some theoretical results further.

The necessity to represent relations between models also arises in traceability approaches. Galvao and Goknil [64] present a survey of traceability approaches with focus on model driven engineering and a comparison to non-model driven engineering approaches. All of these approaches focus on design time and on keeping track of relations among models. Galvao and Goknil distinguish two kinds of such relations (or mappings). *Inter-level mappings* describe relations between models in different development phases. They are for example used to track the impact of requirements or soft-goals, described in their own models, along the modelling process. *Intra-level mappings* are relations between models in the same development phase. The more relevant mappings for this thesis are intra-level mappings as they relate multiple models that coexist at the same time. However, most of the approaches for traceability mentioned in this survey are specific to an application domain (e.g., requirement engineering) and cannot be generalized to arbitrary models and relations.

Also within the scope of traceability approaches, Ramesh and Jarke [65] derived a reference model for traceability of requirements. This reference model gives strong motivation why artifacts need to be connected and describes which kind of information needs to be available in the connection in the case of requirement traceability. They also state that in most state of the art cases in this area the connections between artifacts are done using relational databases in a table-like manner. While relations between models

can also be represented in this manner, the fact that separated formalisms are used for models and relations complicates the joint transformation, constraints and analysis.

Jouault describes an approach for traceability in transformations using the Atlas Transformation Language (ATL) [66]. In this approach an ATL transformation that generates a model from another model generates an additional set of traceability links between these models. The links describe which source and target models are related and contain an attribute to mark the ATL transformation rule that generated these dependencies. The approach is only applicable to inter-level mappings. It does not provide information on how these relations can be produced and maintained for models that coexist in the case of intra-level mappings.

One of the open issues identified from the state of the art for traceability approaches by Galvao and Goknil [64] is whether trace models and incremental model transformation can support each other. In incremental model transformation a target model is not completely generated from a source model but both evolve over time. Adaptation of multiple models at run time is such as a case.

Falleri et al. describe a traceability approach based on the KerMeta language [67]. In this approach traceability is established via a bipartite graph, describing which elements result in which other elements via the transformation. The representation of Falleri et al. is used specifically to keep track of KerMeta transformations and thus is used for inter-level mappings.

The term mapping is closely related to model relations in the scope of model transformations. In literature this term is used for different aspects of model transformation. Sottet and Favre compare existing definitions of this term to clarify this ambiguity [68]. They develop a mapping model that keeps track of changes between transformed models as well as the cause of these changes. Their mappings relate elements in the source and target model of a transformation. This can be seen as a relation between models but requires them to be transformed from each other. For example, the mapping also keeps track of thehaydens poldwh transformation rules and the elements in the transformation that correspond to source and target of the mapping. Again, this approach is more suited to capture inter-level mappings than intra-level mappings. In [69] the approach is applied to usability. It is extended with the annotation of usability criteria to propagate changes along mappings. However, the close relation to model transformation remains.

Wolfe et al.[70] present an approach for maintaining consistency between a conceptual and a distribution model. The distribution model is generated from the conceptual model via model transformation. The consistency is represented by a mapping that consists of the transformation path for this transformation. Changes are propagated along this path. The approach is also applicable to other models that are transformed from each other in a way that enables the generation of a transformation path. However, the relation is only possible if the models have been transformed from each other and is thus not a general way to relate two models. Furthermore, this indirect relation of model elements is not a good foundation for a formal analysis that involves model relations.

The language Meta Object Facility (MOF) and its reduced version eMOF are widely used languages for representing models [12]. These language have already found applications in a variety of tools and frameworks, e.g., the Eclipse Modelling Framework [20].

They are the foundation for most of the mapping and traceability approaches presented in this section. MOF is the meta meta-model used as basis for the Unified Modeling Language and can also be used to represent other models. In MOF models can reference each other to establish relations. This can be used to represent intra-level mappings. MOF can also be combined with several formalisms for adaptation and consistency conditions, e.g., the Object Constraint Language (OCL) and ATL. These formalisms are described in the next sections. The language MOF is defined in an intuitive way via a specification in natural language. However, as stated by Rutle et al., *"... the conformance relation between models and meta models is not formally defined for MOF-based modelling languages, especially when OCL Constraints are involved."*[81]. This missing formalization makes it hard to apply formal analysis methods directly to MOF.

Diskin proposes a mathematical formalisation of UML that could remedy this problem [71]. The formalisation is done via sketches in category theory and could be combined with existing theoretical results in this area of research. Diskin formalises each diagram type in the UML separately. Accordingly, the approach is restricted to the diagram types that have been formalized. It is not trivial to extend to arbitrary modelling languages.

Another area of research that relates multiple models is model weaving. The purpose of model weaving is to combine multiple models based on some common elements. Del Fabro et al. [72] describe the AMW framework and a conceptual description of model weaving based on the terms in the mega model described in the conceptual framework (cf. Section 2.1). In this view a weaving model refers to two meta models and represents how their elements can be mapped. The conforming models can then be related according to this weaving model. The approach is not formally defined. The implementation in the AMW framework is based on EMF and can thus be seen as operating on the eMOF language. It thus also suffers from the missing formalization of this language. Besova et al. [73] propose a separated weaving model that operates on the model-level and relates a set of models. The weaving model can be jointly transformed with these models into different target models. In this approach, the weaving model represents the relations between models using references. However, the approach does not define a formal representation of these relations.

4.2.2. Condition Languages

In this section we discuss languages for the representation of consistency requirements. The Trollmann approach requires a language that:

- **Model Language:** can be combined with a language for models that fulfils the requirements described in Section 4.2.1.
- **Formal Basis:** is formally well-founded and expressive enough to formulate consistency conditions.

Table 4.2 summarises the reviewed condition languages.

One language that is frequently used in formal specification is nested conditions (cf. Section 2.2.3). Nested conditions are a generalization of positive and negative application

Table 4.2.: Evaluation of condition languages.

Language	**Model Language**	**Formal Basis**
Nested Conditions [19]	$\mathcal{M}$-adhesive categories	✓
OCL [82]	MOF	$\sim$
Schematron [83]	XML	✓
Alloy Conditions [84]	Alloy Models ($\approx$ MOF)	✓
Mapping Consistency [85]	Mappings(Sottet)	X

conditions [19]. The purpose of the generalization is to enhance the expressiveness of these formalisms to be equivalent to first order logic in graphs [86]. Nested conditions can be combined with any $\mathcal{M}$-adhesive category. Of the modelling approaches described in the previous sections attributed typed graphs and triple graphs fulfil this condition. Several formal results have been established for nested conditions. Some of them have been described in Section 2.2.4. Thus, nested conditions can be considered a good formal basis for analysis of adaptation conflicts.

One widespread language for constraints in the context of UML models is the Object Constraint Language [82]. OCL is a declarative language. It operates on the MOF formalism. The relations between OCL and several adaptation techniques have been explored. Cabot et al. describe a technique that enables the generation of OCL pre-conditions for graph transformation productions from OCL post-conditions [87]. These preconditions can be checked to derive whether the result of the transformation will satisfy the post-condition. This is similar to results that have been achieved for nested conditions. The declarative textural syntax of OCL makes a direct analysis of this formalism very complex. For this reason OCL is frequently converted into other formalisms to enable analysis of the defined statements. For example, Kuhlmann and Gogolla translate OCL into relational logic to enable the application of efficient SAT solving techniques [88]. Winkelmann et al. translate a restricted form of OCL constraints into graph constraints to make use of formal concepts in graph grammars [89].

Schematron [83] is a constraint language that is used for specifying constraints on XML documents. Model driven engineering is often closely related to the XML technological space as XML is frequently used as format to store and transmit models. Schemtron is able to describe complex structural constraints within XML documents and could thus be used to restrict the structure of models expressed in this language and analyse these conditions. Malý and Nečaský [90] propose an approach for translating OCL constraints based on a platform independent model into Schemtron constraints on a platform specific model represented in XML. The goal is to preserve the constraints during the engineering process. Schemtron is formally well-defined and could thus be used as a foundation for an analysis of adaptation-consistency conflicts. However, XML is usually not the primary representation of models. Furthermore, references to non-child elements in XML documents are usually encoded indirectly into attributes, for example

via an id system or an XPath expression. Accordingly, XML is not an optimal basis for a formal analysis.

Alloy [84] is a modelling language that is similar to the Unified Modeling Language but has a formal foundation in relational logic. This formal foundation also enables the formulation of constraints, queries and invariants. The Alloy Analyser is a tool that enables verification of an Alloy model by translating it into Boolean logic and applying SAT solvers. Yujing He compares the combination of OCL and UML with Alloy models and conditions [91]. According to this comparison, Alloy is more accurate due to its formal foundation, while UML is more expressive. The formal foundation of alloy in relational logic is a sufficient formal basis for analysis of adaptation conflicts. However, according to He, Alloy is less expressive than the UML and can represent only a limited set of modelling languages.

In the context of mappings Sottet et al. explore a notion of consistency and its relations to mappings [85]. They propose to utilize mappings as structural relations and functions for propagating relevant changes along models. However, *"modeling consistency along the net of models and mappings"* [85] is still identified as an open issue.

4.2.3. Adaptation Languages

In this section we describe state of the art adaptation languages. The Trollmann approach requires a language that:

- **Model Language:** can be combined with a language for models that fulfils the requirements described in Section 4.2.1.
- **Condition Language:** can be combined with a language for conditions that fulfils the requirements in Section 4.2.2.
- **Formal Basis:** is well-formalized such that it can be used as the foundation for an analysis of adaptation conflicts.

The reviewed languages are summarized in Table 4.3.

Graph transformation is a formal approach for which a rich body of analysis methods has been established. It has already been used to model self-adaptive systems, e.g., by Becker and Giese [98]. Two main approaches to formulate graph transformation exist. The double pushout approach has been described as part of the fundamentals in Section 2.2.2. The second approach is the single pushout approach. The main difference is that the single pushout approach can delete dangling edges, while the double pushout approach requires the edge to be deleted explicitly during the transformation. Major results in graph transformation are formulated based on $\mathcal{M}$-adhesive categories. Accordingly, graph transformation can be combined with attributed typed graphs and triple graphs. It has also been applied to distributed graphs [80]. Graph transformation can be combined with nested conditions since both are formulated in the same formal framework. The combination with OCL conditions is also possible, as described in the previous section.

Table 4.3.: Evaluation of adaptation languages.

Language	**Model Language**	**Condition Language**	**Formal Basis**
Graph Transformation [38]	$\mathcal{M}$-adhesive categories, Distributed Graphs	Nested Conditions, OCL	✓
ATL [92]	MOF	OCL	X
Maude$_{ATL}$ [93]	MOF	OCL	✓
QVT [94]	MOF	OCL	X
XSLT [95]	XML	Schematron	∼
YATL [96]	MOF	OCL	X
BOTL [97]	MOF		X

The atlas transformation language has been defined as a standard for model to model transformation in the scope of MOF. Thus, it can be combined with MOF models and OCL conditions. ATL has been enhanced with the keyword *refining* to enable model reconfiguration [92]. It is described in an intuitive manner using a specification in natural language, rather than a formal specification. As pointed out by Troya and Vallecillo *"... this lack of rigorous description can easily lead to imprecisions and misunderstandings that hinder the proper usage and analysis of the language, and the development of correct and interoperable tools."* [99]. Due to this missing formalization we do not consider ATL a suitable foundation for the analysis of adaptation-adaptation conflicts.

Clavel et al. define a formal semantic of ATL using rewriting logic and Maude [93]. This formalization includes the keyword *refining* and could thus also be used for model reconfiguration. In the formal semantic the changes done by an ATL program are formally described as Maude rewrite rules. This formal foundation of ATL (called Maude$_{ATL}$ in Table 4.3) could be used as a foundation for the detection of adaptation-adaptation conflicts. The integration of OCL conditions is also possible in this approach.

The transformation language query/view/transform (QVT) is a result of a call for proposals by the Object Management Group with the purpose of defining a standard model transformation language [94]. QVT contains sub-languages, enabling the declarative as well as imperative description of model transformation. Like ATL, QVT is based on MOF models and can be combined with OCL constraints to represent consistency requirements. The specification is given in natural language and thus cannot be used as formal basis for an analysis directly. In addition, QVT contains a concept called *Black Box* that enables the execution of arbitrary external transformation programs. This concept is impossible to analyse, since the external transformation program can be expressed in any language. An approach that excludes this black box concept and uses a formal semantic for QVT could be used and analysed in a similar scope as Maude$_{ATL}$, provided all required structures for the transformation are formally represented. There are several approaches that aim to provide such a formal semantic for QVT, e.g., Gi-

andini et al. [100] apply the theory of problems for this purpose. A formal semantic of QVT has similar properties as Maude$_{ATL}$ with respect to our criteria.

XSLT [95] has also been used as transformation language in the scope of model driven engineering, e.g., in the MTRANS framework [101]. XSLT is based on XMI files. In order to counter problems concerning poor readability, large transformation descriptions and unintuitive mappings via XPath the MTRANS framework adds an abstraction layer to XSLT. This layer enables a developer to specify a transformation based on MOF meta models rather than the conforming models. The meta models are also expressed as XMI files. Since XSLT is part of the XML technological space it can be combined with XML conditions, such as Schematron. Several methods for the analysis of XSLT transformations have been developed. For example, Dong et al. derive a graph representation of an XSLT template in order to analyse certain properties of the transformation, e.g., unreachable sub-transformations. A significant portion of XSLT has been formally founded by Bex et al. [102], thus making it available for formal analysis. Unfortunately, XSLT does not support in-place transformation directly as it always creates a new XML tree from an existing one. Model reconfiguration is expressible by creating a new model in the same modelling language and overwriting the source model after the transformation. However, the overwriting is not optimal as all elements that are unchanged by the reconfiguration have to be handled with identical XSLT transformations. This makes analysis more complex.

Patrascoiu developed the model transformation language Yet Another Transformation Language (YATL) [96]. This language is intended to be applied to MOF models for model to model transformation. It contains OCL constraints for restricting matches. One of the requirements that YATL aims to fulfil is a formally founded syntax and semantic. However, the language allows the execution of external transformation functions (for instance Java code). Accordingly, the effects of the transformation are not completely formally captured and the language cannot be used as a foundation for analysis.

The bidirectional object transformation language (BOTL) [97] has been developed as an extension of XSLT and graph transformation. The main purpose of this language is a more expressive model to model transformation. However, due to the differences between BOTL and graph transformation the existing analysis methods are not applicable.

4.2.4. Summary

The combination of graph transformation and nested conditions is a suitable basis for the detection of adaptation conflicts. Both languages are formally founded and a rich body of theoretical results exists. Maude$_{ATL}$ and OCL are a potential alternative. However, they are based on MOF, which has the problem of a missing formal foundation.

Both graph transformation and nested conditions are based on $\mathcal{M}$-adhesive categories. Accordingly, they can be applied to attributed typed graphs and triple graphs. Distributed Graphs are not $\mathcal{M}$-adhesive, but it has been shown that several of the results from $\mathcal{M}$-adhesive categories also hold in distributed graphs.

All three of these languages for models have certain disadvantages. Attributed typed graphs can only be used to represent multiple models and relations by merging them into

one model. Triple graphs are able to represent models and relations as separate entities and thus enables a better separation of concern. However, in this formalism only two models can be related, which is a serious restriction. Distributed graphs are able to represent an arbitrary number of models. However, they require relations to commute, which is too restrictive.

Since none of the above formalisms are optimal, we define a new formalism, called graph diagrams. Graph diagrams are similar to distributed graphs in that they represent a generalization of triple graphs to multiple models and relations. However, they relax the requirement of commuting morphisms and can be shown to be $\mathcal{M}$-adhesive. Thus, they can be combined with graph transformation and nested conditions.

4.3. Conflict Detection

Zhang and Cheng state that the development of SAS systems is more complex and more error prone than the development of a non-adaptive system and accordingly *"In order to be trusted, it is important to have mechanisms to ensure that the program functions correctly during and after adaptations"*[46]. Conflicts between adaptations and between adaptations and structural consistency are one of the aspects that need to be handled by run time assurance. In this section we describe other definitions of conflicts involving adaptations and approaches for detecting them.

In rule-based approaches to adaptation multiple adaptations can be applicable at the same time because of overlapping conditions. Hussein et al. [60] categorize different problems that can occur in this situation independent of the adaptation technique. One of them is *Adaptation Behaviour Inconsistency* which represents situations where one element is added and removed or changed to two different values at the same time. This can be seen as a more specific definition of adaptation-adaptation conflicts as handling the same attribute differently in adaptations usually leads to different result in different orders of adaptations or even to non-applicable adaptations.

In the scope of graph transformation this notion of conflicts corresponds to dependencies between graph transformations. Mens et al. [103] study such dependencies in an environment where developers are working on the same artifact concurrently. The editing operations are represented as graph transformation productions. Mens et al. [104] also investigate the necessity to resolve dependencies between graph transformation productions representing adaptations. They apply a static dependency analysis at design time using critical pairs and the Local Church Rosser Theorem. Hausmann et al. [105] utilize dependencies of graph transformation to analyse conflicting requirements that are expressed as use cases. In the Trollmann approach we use dependencies between graph transformation productions to represent dependencies between adaptations that indicate adaptation-adaptation conflicts.

In the area of model versioning conflicts between the merged models have been subject to research. Taentzer et al. [77] rely on graph transformation and nested conditions as formal representation of conflicts in the scope of model versioning. They define two types of conflicts: operation-based conflicts and state-based conflicts. These conflict situations

correspond to adaptation-adaptation and adaptation-consistency conflicts. Taentzer et al. are able to detect these conflicts in the scope of single models, represented as attributed typed graphs, for two adaptations. They do not extract explicit reasons for the found conflicts. Ehrig et al. define a merge-operator for resolving operation-based conflicts that prefers insertion over deletion [106]. The operator produces one potential solution of the conflict which can to be revised by a user. The conflict resolution mechanism is limited to two conflicting adaptations.

Rutle et al. describe a notion of conflict in the scope of model versioning in the copy-modify-merge paradigm [107]. The approach is formalized in category theory but the notion of conflict is based on one specific category and cannot be generalized. The approach can also be formalized using a span of graphs and as is done in [106].

Cicchetti et al. also describe a conflict detection mechanism for model versioning [108]. They distinguish between syntactic and semantic conflicts. Syntactic conflicts are conflicts that can be detected between two adaptations on the basis of their structure. They conceptually correspond to adaptation-adaptation conflicts. A semantic conflict requires a technique to specify this conflict. The conflict detection is done on the basis of a change model, which describes the changes between the current model version and the edited one. The detection of syntactic conflicts seems to be equivalent to dependency analysis in graph transformation. It is based on two adaptations at a time and does not directly support the extraction of reasons for the conflict. Semantic conflicts are detected on the basis of a conflict description, which is a specification of a conflicting situation. This concept is similar to critical pairs but seems more expressive as it can also build Boolean formulas over conflict descriptions. The authors do not elaborate on how a global condition can be expressed correctly and completely by their conflict description or whether the conflict description can be used to extract reasons for the conflict.

In the context of dependencies between graph transformation productions, critical pair analysis [38] is frequently applied to analyse the existence of dependencies representing adaptation-adaptation conflicts. Hausmann et al. [105] as well as Mens et al. [103] utilize this technique for finding conflicts. Critical Pair analysis is a technique where minimal examples of conflicts (so-called critical pairs) are calculated based on a set of graph transformation productions. Each critical pair represents a potential parallel dependency between two productions and thus a conflict in the sense of Hausmann et al. and Mens. Critical pair analysis is typically done at design time. It requires checking every existing pair of adaptations for critical pairs. This analysis does not state whether this critical pair will actually occur in any state at run time or whether the conflicting productions will ever be activated at the same time. It represents potential adaptation-adaptation conflicts that need to be evaluated with respect to whether they can occur or not. Thus, critical pair analysis suffers in adaptive systems that contain a large set of adaptations or whose state-space contains a large set of steady-state programs.

Becker and Giese [98] utilize attributed graphs and graph transformation as formal representations of SAS systems for the purpose of checking correctness. Correctness is defined by a set of invariants that need to be fulfilled. Invariants are expressed as graph patterns that represent forbidden situations. This could be seen as a detection of adaptation-consistency conflicts using graph patterns as consistency requirements.

In some cases the applicability of graph transformation productions is also utilized to detect inconsistencies that result from adaptive behaviour. Goedicke et al. aim to detect inconsistencies in different viewpoints represented as distributed graphs [109]. Consistency is expressed by two sets of graph transformation productions that represent positive and negative consistency requirements. The existence or non-existence of match morphisms determines whether the distributed graph is considered consistent. The concept of requiring the existence or non-existence of a match morphism is also the basic concept of nested conditions. The approach of Goedicke et al. is similar to the self-healing approach of Ehrig et al. [43]. In this approach graph transformation production applicability is utilized to detect inconsistent states. The graph transformations themselves are used to correct these states and thus resolve consistency problems. Ehrig et al. utilize this setup to formally analyse self-healing with regard to deadlock-freeness and liveness. This notion of consistency is not as expressive as formalisms like nested conditions. It only enables to detect inconsistencies after they occur.

There are also several approaches that are specific to modelling languages. For example, Van der Straeten et al. [110] describe an approach for analysing consistency between certain kinds of UML models. The goal is to find inconsistencies in models that jointly evolve during design time due to changes imposed by the developers. The approach is specific to the used models and guarantees the consistency of these models. This notion of conflict is loosely related to adaptation-consistency conflicts as it requires describing and detecting inconsistencies among models.

Badr and Reilly [44] propose an architecture for resolving conflicts in an agent-based service oriented framework. The notion of conflict in this work is not related to adaptation as they research conflicts that can occur when an agent receives a request that cannot be handled. However, despite this difference in scope they define information that needs to be delivered by a diagnostic component to resolve this conflict. This information is general enough to be of interest for resolution of adaptation conflicts. Badr et al. identify the following information:

- *Identification of conflicting parts:* Originally this information concerns the parts of a control rule that raised a conflict. In the scope of adaptation conflicts it can be interpreted as: Which adaptations are responsible for the conflict?
- *Identification of conflict cause:* In the agent-based approach this is done via inspecting the service attributes. On the level of adaptation conflicts this information can be interpreted as: Which elements in the current state of the software system are concerned by the conflict?
- *Classification of conflict type:* There can be several types of conflicts and depending on which type of conflict occurred a different solution strategy might apply.

The questions we defined in the problem statement (cf. Chapter 3) to indicate information that needs to be provided by the conflict detection are based on these questions.

4.4. Summary

In this chapter we reviewed related work for the Trollmann approach. We discussed several approaches to adaptation in software engineering. Several of these approaches provided motivation for a run time detection of adaptation conflicts, e.g., the open adaptation facet, defined by Salehie and Tahvildari [55], denotes that new adaptations can be added at run time and thus requires a conflict detection at run time.

In order to find a suitable formal foundation for the Trollmann approach we reviewed formalisms for models, adaptations and consistency requirements. The combination of attributed typed graphs, graph transformation and nested conditions is the most promising one but does not fulfil all requirements. Our approach builds on these formalisms and extends them to represent multiple models and relations between them.

We also reviewed several notions of conflicts and inconsistencies involving adaptations and explored their relation to adaptation conflicts. From the conflict resolution approach of Badr and Reilly [44] we identified several relevant questions for the resolution of adaptation conflicts.

In the next chapter we describe the case study that will be used as running example for illustration and evaluation of the Trollmann approach.

5. Case Study

We illustrate and test the Trollmann approach based on a case study in an existing research-project. We applied the implementation of the Trollmann approach (cf. Section 7), to two research projects at the TU Berlin. The project *"BeMobility 2.0"*, funded by the German Federal Ministry of Transport and Digital Infrastructure,[1] provides a smart phone application for users of carsharing with electric vehicles. This project uses graph transformation to adapt the user interfaces of this software system to different screen sizes. We applied our analysis to detect conflicts between these adaptations. Similarly, we applied our approach to an application for adaptive heating control in the project *"System zur Optimierung des Innerhäuslichen Energiebedarfs"* (in english: *"system for optimising the energy consumption at home"*) (SOE) funded by the Federal Ministry of Economics and Technology.[2] The case study is based on the project SOE. We chose this project because it contains the more complex models and is thus a more interesting showcase.

In this chapter we describe the project SOE and the foundations of the case study. First, we present the project and its topic (cf. Section 5.1). Subsequently, we describe the models used in this project (cf. Section 5.2), the adaptations that will be used in the case study (cf. Section 5.3) and the consistency requirements (cf. Section 5.4). We conclude the chapter with a summary (cf. Section 5.5).

5.1. Description of the Case Study Application

The goal of the project SOE is to develop an adaptive heating control system. This system is adaptive in several aspects. On the one hand, the project aims to learn the times in which users are at home, and the rooms they need to use. Based on this information the heating control can be adapted to heat rooms when they are likely to be used. The learning algorithm is based on the energy consumption.

The user accesses a graphical user interface to review the heating plans derived by the SOE system. She can change either the current heating or the planned heating at a specific point in time. This user interface is available on desktop devices, smart phones and tablets. The user interface is also adaptive. The following situations can cause an adaptation of the user interface:

- **Device Properties:** The user interface adapts to properties of the graphical interaction device. The main properties are the screen size and screen dimension

[1] http://www.bmvi.de/EN/root.html
[2] http://www.bmwi.de/EN/root.html

of the interaction device. This is used to optimize the user interface to bigger and smaller screens to scale it to a smart phone as well as desktop devices.

- **Level of Detail:** The user interface can provide a variety of information about the heating system. However, not all users want to see all available information. For example, some users are contempt with the knowledge that the living room will be heated appropriately between 10 and 12 while others want to know the exact heating level. The user interface provides several levels of detail for each window. The user is able to select the preferred one.

- **Customized Adaptations:** The SOE project allows custom adaptations to enable the user to customize the application. The user is able to select predefined adaptations and to specify new ones.

The control of UI adaptation by the user is an explicit goal of SOE. This is achieved via a meta-user interface (Meta-UI) — an additional user interface that enables the user to control the actual user interface [111]. Via the Meta-UI the user is able to choose between different levels of detail for each window. The Meta-UI also enables the user to understand the adaptive behaviour of the SOE application. It explains the existing adaptations and visualizes their application status. The user can influence the adaptive behaviour by activating / deactivating adaptations or creating new ones.

Because of the high dynamicity of the user interface, the project SOE is a good example for a system that needs a run time analysis of adaptation conflicts. Since adaptations can be activated and deactivated by the user it is impossible to predict at design time which adaptations will be applied at the same time. In addition, the user is able to add new adaptations that are not known at design time and thus cannot be analysed statically. Thus, the user interface of the SOE project requires safety mechanisms for adaptations. The Trollmann approach can provide a basis for such mechanisms.

The user interface of the SOE project contains the following windows:

- **Current Situation Window:** This window enables the user to access information about the current situation for each room. The user is able to select a room and see the current state of the room. This state contains information on the current temperature and heating level of the room.

- **Planning Window:** This window provides an overview about the heating plan for each room. This plan describes the times and levels for heating.

- **Meta-UI Window:** This window contains the meta user interface.

The case study focuses on the planning window as this window provides opportunity for several complex adaptations. The user interface of this window is shown in Figure 5.1. The window displays the heating plan for one week. The heating plan for each day of this week is displayed individually. For each hour of the day, one rectangle indicates the planned heating by its colour. Black means there is no heating and red means that the

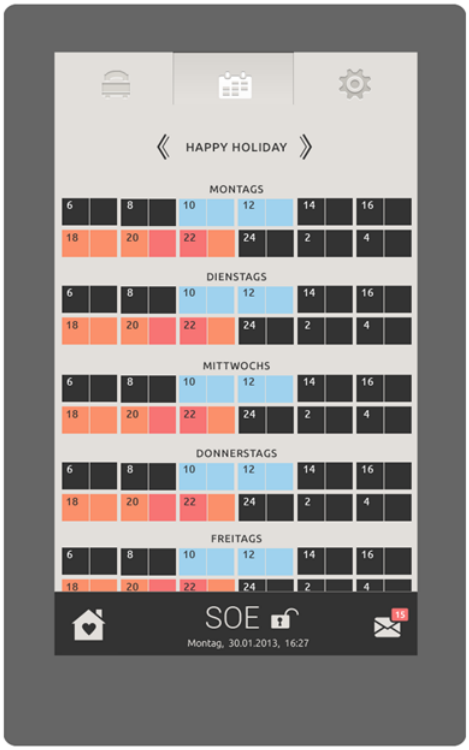


Figure 5.1.: The user interface of the planning window in the case study.

heating system is turned on. The level of heating is indicated by the shade of red. Blue means that the room is cooled. This option has been disregarded later in the project.

The next section describes the modelling languages used for user interface development in SOE. The models are exemplified in the planning window.

5.2. Models

This section describes the models of the project SOE. These models are quite large and not all of their elements are relevant for this thesis. For this reason we omit some model elements to concentrate on the most relevant ones.

An overview of the participating models is given in Figure 5.2. An SOE application consists of a dialog model, a user interface model, a context model and a layout model. The purpose of these models is as follows:

- **Dialog Model:** The dialog model describes the interaction workflow. It describes the states of the user interface and the transitions between these states.
- **UI Model:** The UI model contains all user interface elements.

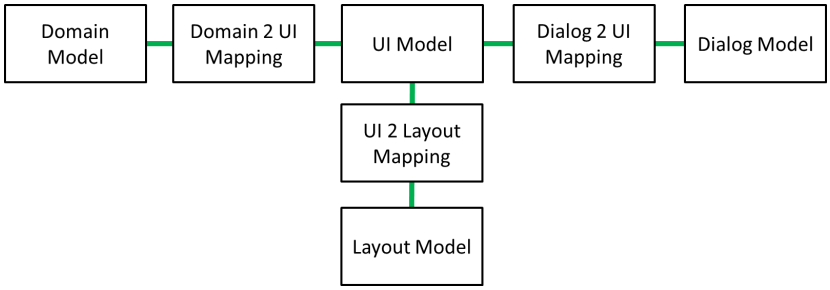


Figure 5.2.: The models of the project SOE.

- **Layout Model:** The layout model describes how the elements in the UI model are positioned and layouted on the screen.
- **Domain Model:** The domain model contains objects that describe the relevant information for the software system. For example, it contains the heating plans and current room temperatures.

In addition, the software system contains a mapping model that describes relations between these models. In the view of Sottet and Favre [68] the mapping model tracks relations between models. In the project SOE the mapping model takes a more active part in synchronizing models at run time. A Mapping in SOE synchronizes two elements from two models. The following general mapping types can be distinguished, depending on which models they connect:

- **Dialog 2 UI Mapping**: This mapping is responsible for establishing the relation between the dialog states and the UI elements that are supposed to be visible when a certain state is active. The mapping sets the UI elements to active/ inactive, whenever the corresponding state is activated / deactivated. Furthermore, a second kind of mapping triggers a transition based on interaction with user interface elements. This enables the implementation of navigational UI elements.
- **UI 2 Layout Mapping**: This mapping relates elements in the UI model to the elements in the layout model responsible for describing their layout. Whenever a UI element becomes active / inactive, its corresponding layout elements become active/inactive as well.
- **Domain 2 UI Mapping**: This mapping propagates values from the domain model to the UI elements that are responsible for displaying these values to the user. All changes in these values are propagated to the UI model.

In the following we provide excerpts from the meta models of these models to give an overview on their structure. The meta models are depicted in EMF diagrams since the SOE framework is implemented in the Eclipse Modeling Framework. Subsequently, the plan window is used to showcase a complete graph of models and mappings.

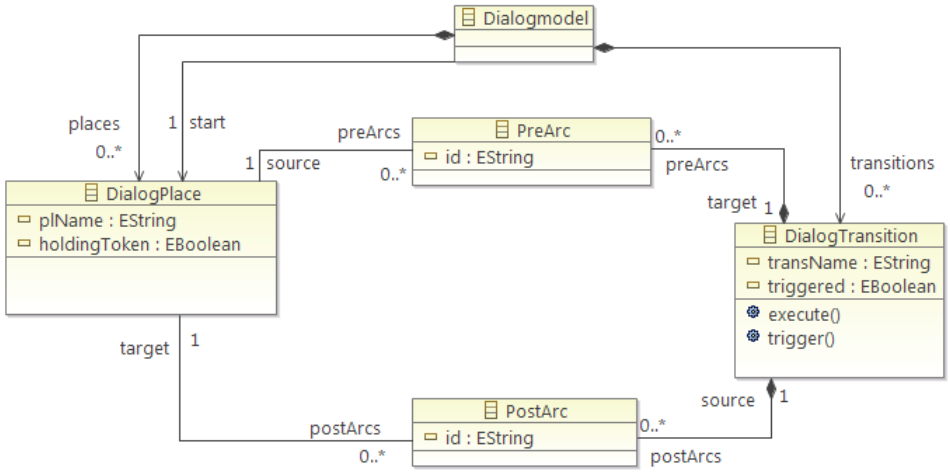


Figure 5.3.: Excerpt from the meta model of the dialog model.

The dialog model in the project SOE is modelled based on Petri Nets [38]. Places in the net represent states of the dialog and transitions represent state changes. An excerpt from the meta model of the dialog model is shown in Figure 5.3. The meta model contains two main classes: *DialogPlace* and *DialogTransition*, which represent places and transitions in the Petri Net. Their connection is represented by objects of type *PreArc* and *PostArc*. Only one token can be placed on each place at a time. This marking represents whether the respective user interface elements are visible or not. UI elements cannot be shown more than once. Thus, all places have a capacity of one. This is represented in the model by the Boolean attribute *holdingToken*.

An excerpt of the meta model of the user interface model is given in Figure 5.4. The UI model consists of a set of *AbstractInteractors* that represent potential interactions with the user in a modality-independent way. The SOE model only uses the graphical modality. It makes use of this abstract representation to switch between graphical elements that allow for the same interaction (e.g., switching from a button to a clickable image). This is similar to the distinction between abstract and concrete user interface in the Cameleon Reference Framework [16]. Some of the derived abstract interactors are listed in the figure. The abstract interactor *OutputOnly* represents an output towards the user, while the abstract interactor *UserInput* represents an input by the user. The abstract interactor *Command* represents a command given by the user, for example by clicking a button. It does not convey additional information. Most of these interactors operate on *DataSlots*, which contain the information that is exchanged with the user (either by displaying or receiving it).

The abstract interactors are implemented by *ConcreteInteractors*. This class, as well as the common superclass *Interactor*, have been excluded from the figure because of their extensive inheritance structure. The figure contains the concrete interactors *GraphicalOutputWidget*, *GraphicalNavigation* and *GraphicalUserInput*. Each abstract interactor references its implementation as a concrete interactor. Several tra-

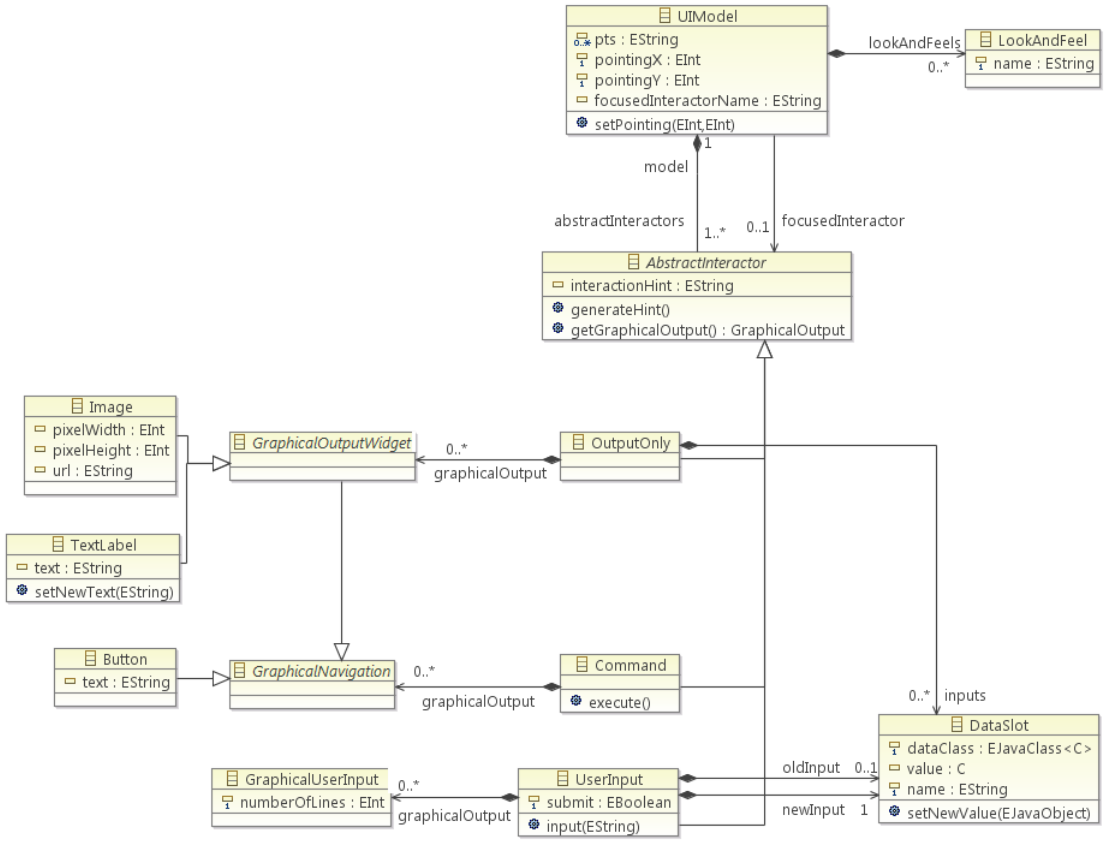


Figure 5.4.: Excerpt from the meta model of the UI model.

ditional graphical UI elements are derived from these concrete interactors. Examples for a graphical output widget are *Image* and *TextLabel*. An example for a graphical navigation is a *Button*. A *LookAndFeel* of the UI model represents some general information about its childrens graphical properties (e.g., a common text colour) that can be annotated to single or groups of UI elements.

An excerpt of the meta model of the layout model is depicted in Figure 5.5. The layout algorithm uses constraint solving to optimize the layout to different screen sizes. The *LayoutModel* contains a set of *Statements* regarding the position, relations and size of its layout elements and a configuration (class $LayoutModelConfiguration$). These elements are input to a constraint solver, which is available via the $ConstaintSolverProxy$. The constraint solver takes into account the properties of the current interaction device (screen size, minimum text height ...) and calculates positions and sizes of its elements as a solution to the constraint system. The layout consists of a tree structure of *Nodes*, which themselves can be *Containers* that contain other nodes, or *Leaves*, which form the leaves of the layout tree and represent single UI elements. *Properties*, *NodePositions*

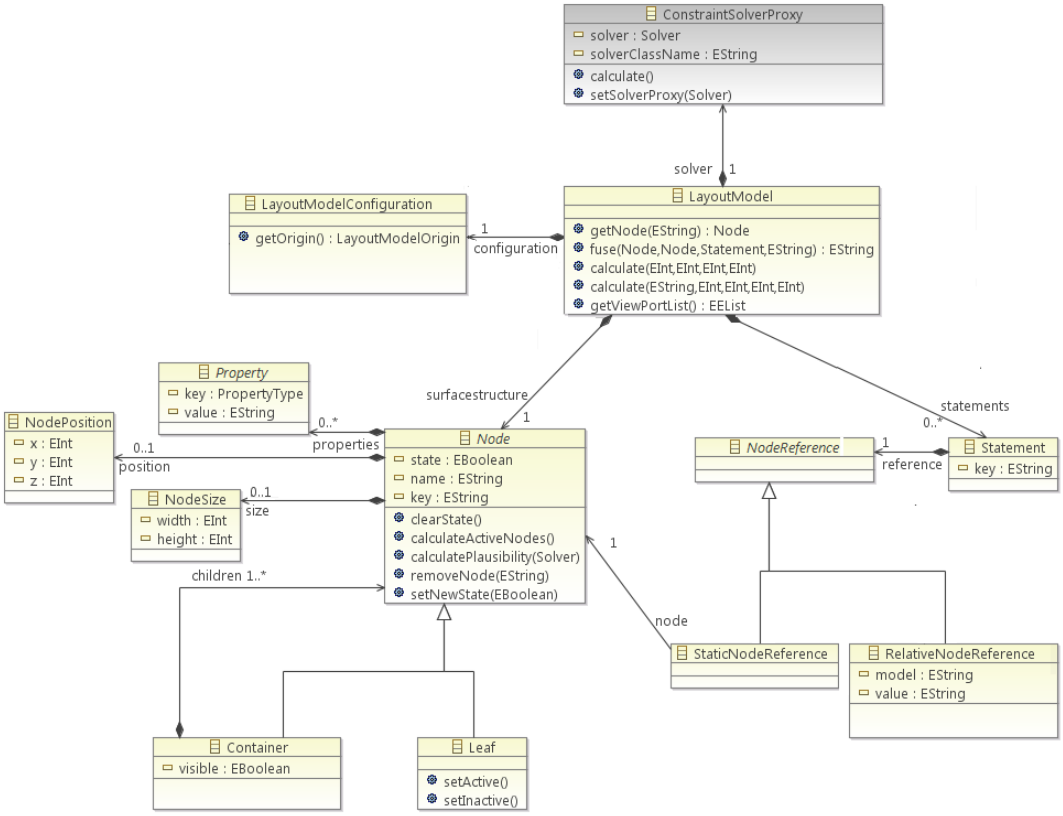


Figure 5.5.: Excerpt from the meta model of the layout model.

and *NodeSizes* are calculated by the constraint solving process. The layout model and constraint solving are specialized to graphical user interfaces.

The foundation of the domain model of the project SOE is depicted in Figure 5.6. This model contains a set of *Properties* that can be connected to sensors that update them if necessary. The subtypes of this class represent properties of certain data types. These properties form the foundation for modelling the domain knowledge of the SOE application. The domain knowledge is stored in a domain model in groups of properties. A group of *DoubleProperties* is used to represent the heating value for each hour of a day. The heating plan of a day is represented by a list of double properties.

Figure 5.7 shows an excerpt of the meta model of the mapping model. This model makes references to the classes *EClass*, *EObject*, *EStructuralFeature* and *EOperation* from the Ecore meta model, which cannot be displayed in an Ecore diagram as they are elements of the meta-meta model. For this reason these classes have been supplemented to the figure. The mapping model contains a set of mapping types and a set of mappings. The mapping types describe a relation of two types, while mappings are responsible for relating two specific objects that have the types described in its mapping type. It would be conceptually cleaner to separate mapping types and mappings into two meta layers.

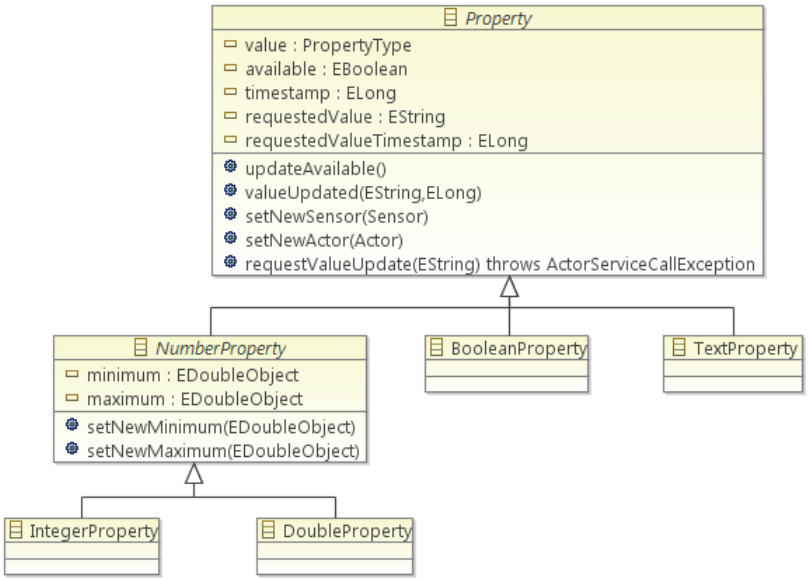


Figure 5.6.: Excerpt from the meta model of the domain model.

However, since EMF only supports a fixed number of meta-layers, mapping types and mappings are both modelled in the mapping model on the same layer. The conformance is represented by an edge of type *type*.

A *MappingType* consists of a set of *MappingLinks*. Each of these links connects two types. In the meta model the according *EClasses* are referenced via *source* and *target*. A mapping link describes that a change in the source object triggers a method in the target object. For this, the mapping link observes one attribute of its source, given by the *EStructuralFeature* specified by the reference *trigger*. If a change in this attribute occurs the *condition* is checked. If it is fulfilled the operation in the target element is called. The operation is specified by the *EOperation* referenced by *action*. Additional *arguments* can be handed over to this operation. These can either be model elements, referenced by static or relative path, or the results of transformations from other model elements. The according subtypes of the class *Argument* and their class structure have been omitted from the figure.

Each *Mapping* has a type. The links in this type restrict which types of objects the mapping can connect. The mapping references two specific objects of these types via the references *source* and *target*. Because of the above mentioned lack of meta-layers it is not possible to enforce *source* and *target* to have the correct types via type conformance to the meta model. Thus, the mapping references two generic *EObjects*. Consistency requirements are used to assure that these objects are instances of the classes given in the mapping type.

For the considerations in this case study five mapping types are of interest. These are shown in Figure 5.8. The following types are contained in this figure:

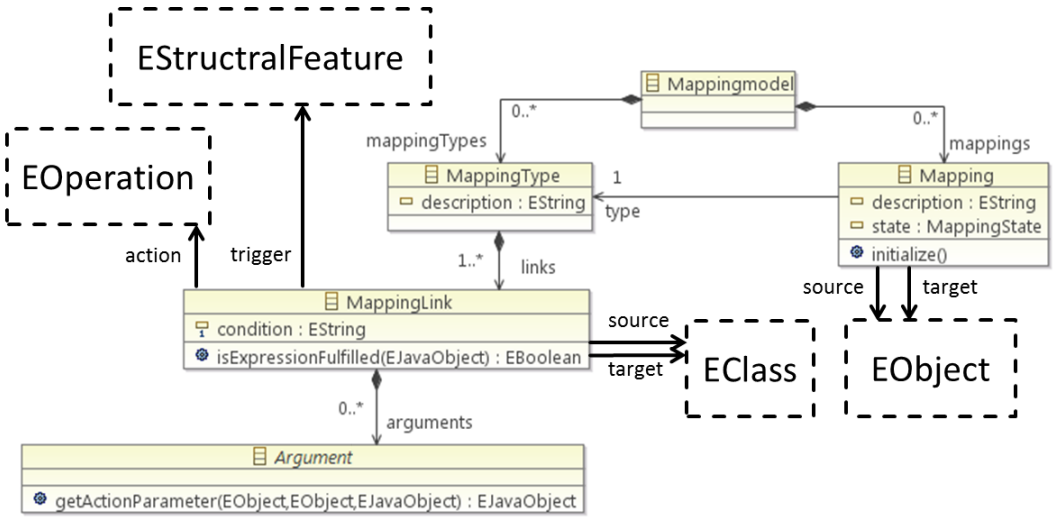


Figure 5.7.: Excerpt from the meta model of the mapping model.

- **Place2UI:** This mapping type connects the types *DialogPlace* in the dialog model and *AbstractInteractor* in the UI model. The purpose of the mapping type is to activate the interactor whenever its corresponding place is marked. This is implemented in two mapping links. The top link triggers whenever the place is deactivated ($holdingToken == false$) and calls the method *deactivate* in the abstract interactor. The bottom link triggers when the place is activated and calls the method *activate* to make the UI element visible.

- **UI2Transition:** This mapping type is responsible for triggering transitions in the dialog model from the UI model. It connects the types *AbstractInteractor* and *DialogTransition*. Whenever the interactor is selected ($state == SELECTED$) the method *trigger* in the transition is called to trigger its execution.

- **UI2Layout:** The mapping type $UI2Layout$ connects the type *Concrete Interactor* in the UI model and the type *Node* in the layout model. *ConcreteInteractor* is the superclass of all graphical UI elements. The mapping is triggered by changes on the attribute *state* and has no condition (the condition is *true*). Accordingly, whenever the *state* of the interactor changes the method $setNewState$ of the *Node* is called. One parameter is handed over. This parameter is *true* whenever the new state is not *INACTIVE*. The respective class structure is abbreviated as attribute *argument* in the figure.

- **Domain2UI_Background:** The mapping type $Domain2UI\_Background$ is responsible for setting the background of a label whenever the value of a corresponding property changes. It connects elements of the classes *DoubleProperty* to a *TextLabel*. Without a condition, each change of the *value*, calls the method *setBackground* with the new style that has the correct background colour for the

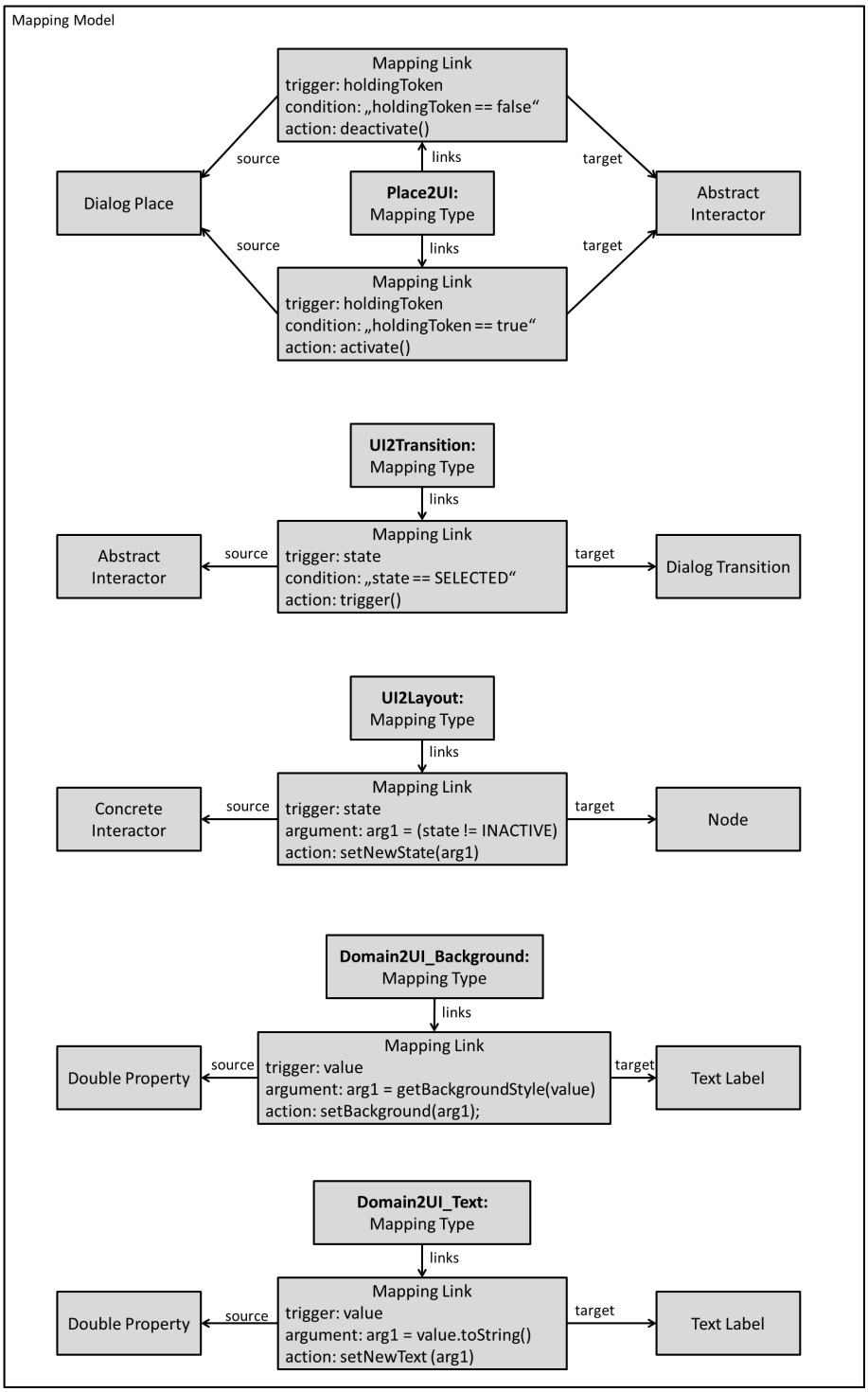


Figure 5.8.: Mappings of the project SOE.

value of the property set. This style is generated by a transformation. In the simplified figure this is represented by the method *getBackgroundStyle*

- **Domain2UI_Text:** The mapping type *Domain2UI_Text* is responsible for setting the value of a text whenever the value of a corresponding property changes. It connects the classes *DoubleProperty* and *TextLabel*. A change in this value triggers the method *setNewText* with the String representation of the new value.

The remainder of this section gives an example for a conforming set of models and mappings by modelling the plan window.

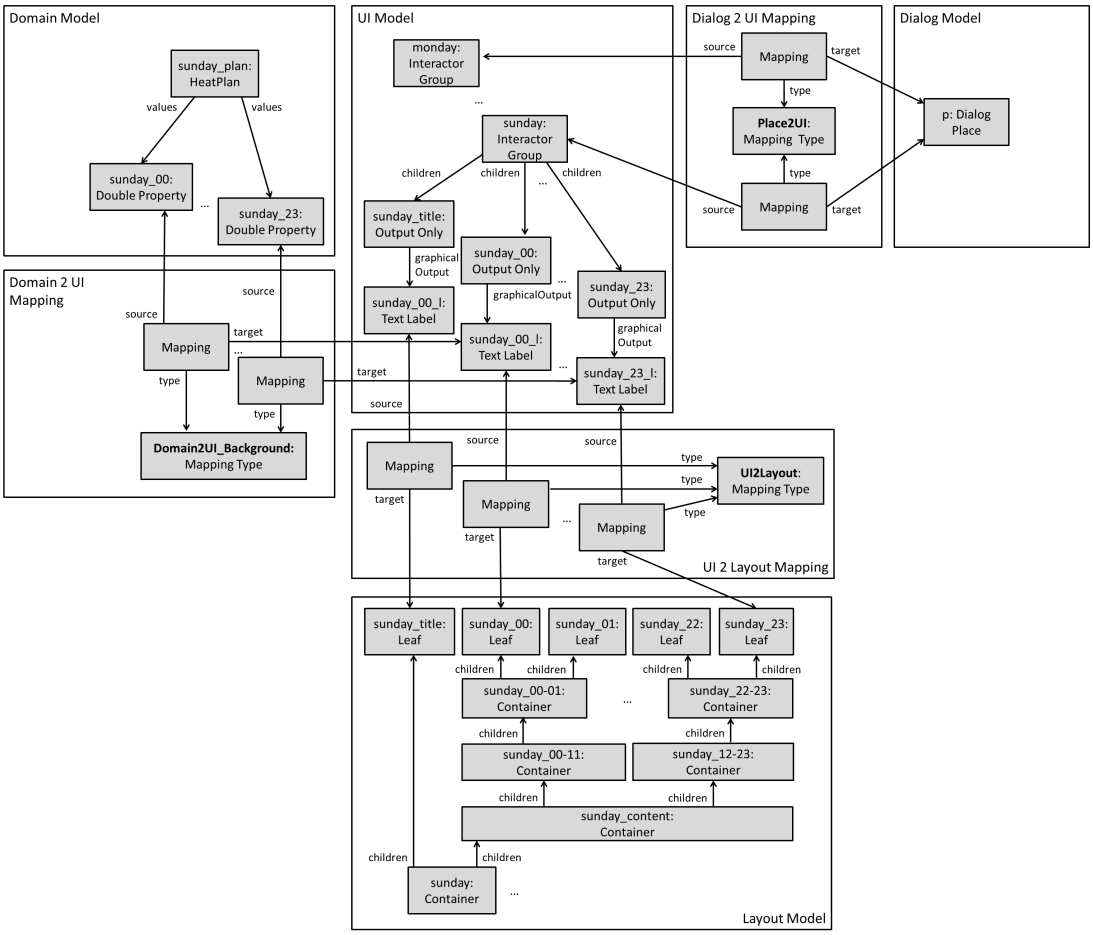


Figure 5.9.: Simplified model of the plan window.

The models of this window are depicted in Figure 5.9 in a condensed version. The dialog model contains the place in which the plan window is visible. This place is connected via mappings of type *Place2UI* to seven elements of type *InteractorGroup*. Each group contains a set of interactors. The groups are called *monday* to *sunday* and each contains the interactors specific to one weekday. Each group has 25 children. This

is exemplified for *sunday* in the figure. The group contains one element *sunday_title* of type *OutputOnly* for the headline of the according day and an additional element *sunday_HH* of type *OutputOnly* for each hour *HH* of the day. These elements are refined by concrete graphical elements of type *TextLabel*. Thus, the user interface essentially contains one headline and 24 labels for each day of the week. These elements are displayed when the according dialog place is marked (visibility of abstract interactors is derived from their *interactorGroup* if no specific mapping exists).

The layout model contains one *Container*, called *sunday* for the whole day Sunday and a container of identical structure for each other day of the week. This container structures its children according to their structure in the layout. On the first layer the leaf *sunday_title* and the container *sunday_content* are situated. *sunday_title* represents the title, displaying the name of the day, and is related to the according label in the UI model by a mapping. The leafs for the remaining UI elements in the interactor group *sunday* are contained in the container *sunday_content*. This container contains two children, each representing a row of twelve hours. Each of them contains six pairs of two hours. This enables the UI to show each two hours as linked rectangles, while there is padding in between each pair of two. The groups $sunday\_00-01$ and $sunday\_22-23$ represent the pairs of hours 0 to 1 and 22 to 23 respectively. The leafs contained in these groups are connected to the text labels in the UI model via mappings of type *UI2Layout*.

The domain model contains the values for the heating plan for each day. For each day there are 24 $DoubleValues$ representing the 24 hours of the day. Again, Sunday is used as example day in the figure. The background for each hour is connected to the according label via a mapping of type $Domain2UI\_Background$. The actual mappings target a graphical property set describing the style of a text label. In the figure this has been abbreviated by a direct *target* reference to the text label.

The next section describes adaptation scenarios for this window.

5.3. Adaptations

This section describes several adaptations of the planning window. The effect of each adaptation is shown in a schematic overview. Furthermore, we indicate the changes in the model triggered by the adaptations.

The first adaptation distributes the heating overview into multiple windows. The schematic overview of this adaptation is shown in Figure 5.10. Two potential distributions are shown. The window at the left hand side shows only one day at a time and adds navigation elements at the top to switch between weekdays. The window at the right hand side enables the user to switch between a weekday and weekend view. Again, navigation is added at the top to switch between both.

This adaptation serves two purposes. Showing less information on the screen results in a better use of space on devices with a small screen size. Thus, it makes sense to show a limited number of days at a time. However, the user might want to group similar information and show the weekdays and weekend, which can have different heating requirements, in different windows.

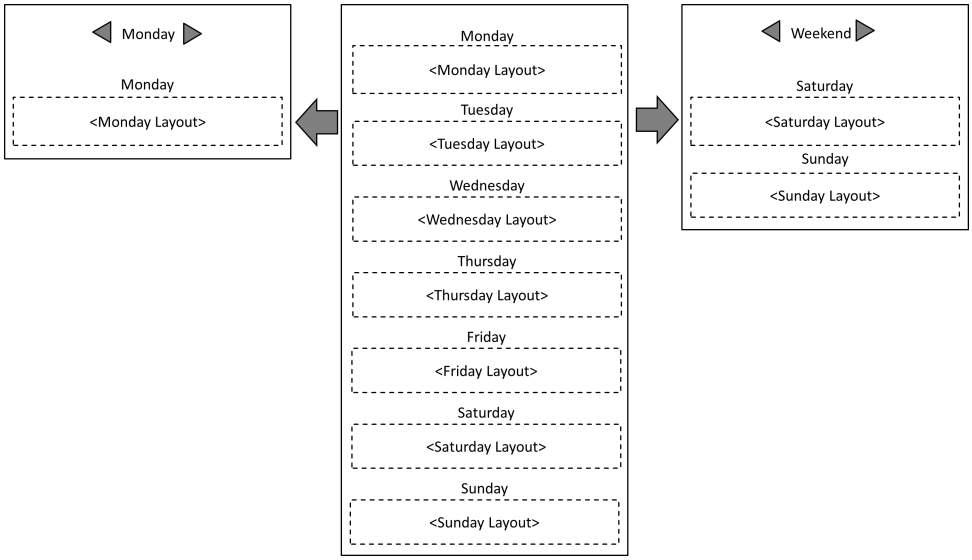


Figure 5.10.: Schematic overview of the first adaptation.

Figure 5.11 depicts the model after the adaptation that causes the user interface to show the weekdays and weekend in separate windows. The figure concentrates on those elements that differ from the original user interface model in Figure 5.9. Furthermore, mappings of the same type that share a common source or target are abbreviated by one mapping with several sources and targets. The dialog model has been changed to contain two *DialogPlaces*. Depending on which of them is activated either the weekend or weekday view is displayed. The modes can be switched via two $DialogTransitions$. Each of the dialog places is related to a set of *InteractorGroups* via mappings. Each of these groups contains the interactors for one day. These groups are structured and connected to the layout and domain model as in the original model. The respective elements have thus been omitted from the figure. The place *weekday* is connected to five such groups for the days Monday to Friday. In addition, it is connected to an element *weekday_header*, which represents the appropriate headline in the user interface. Similarly, the place *weekend* is connected to two interactor groups for Saturday and Sunday as well as a header for the weekend view *weekend_header*. Two commands $left$ and *right* in the UI model represent the navigation between both views. Both elements are connected to both transitions in the dialog model via a mapping of type $UI2Transition$. Accordingly, if either of these commands is interacted with both transitions are triggered. Since only one of them is activated at a time (since there is only one token in the net) only one of them will fire and the state will switch.

In addition to the already known structures, the adaptation adds a container for the navigation to the layout model. Four leafs are situated in this container. They represent the two navigation elements and the two headlines that are connected to the respective concrete interactors via mappings of type $UI2Layout$. Since only one of the headlines is active in each dialog state, only three of these elements are shown at a time.

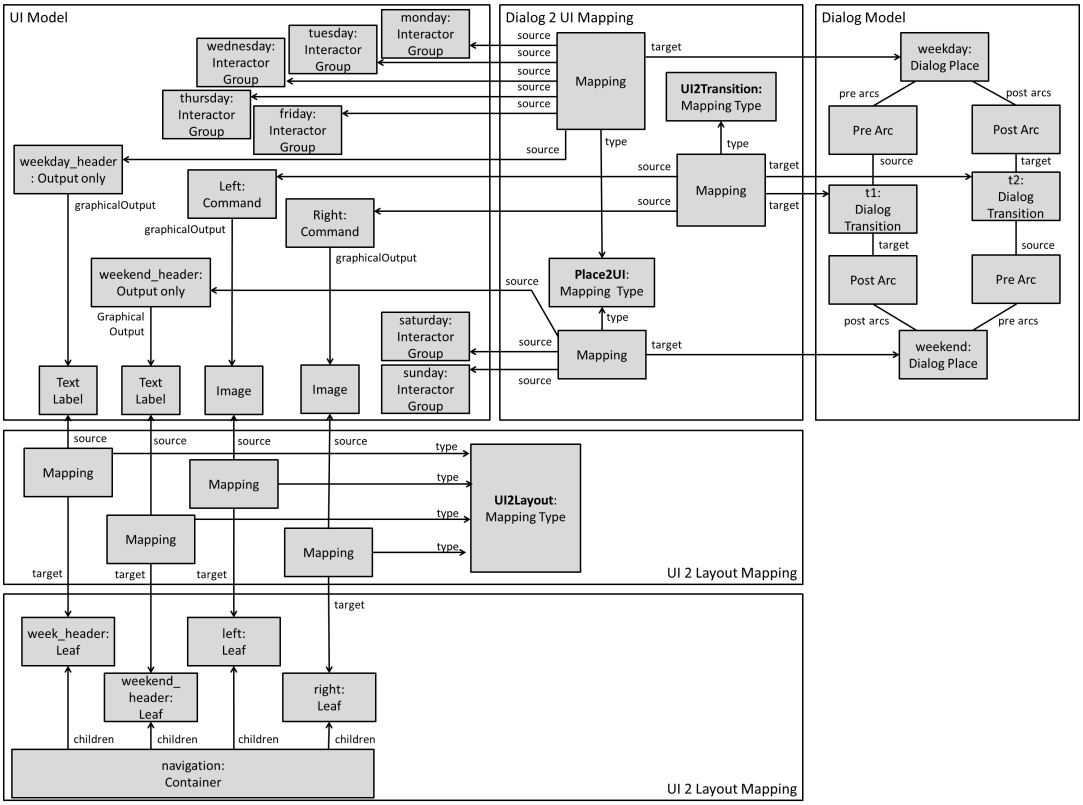


Figure 5.11.: The model after the first adaptation.

The adaptation for showing each day separately looks similar. It adds seven states to the dialog model and the corresponding mappings, headlines and layout elements.

A schematic overview of the second adaptation is shown in Figure 5.12. This adaptation reorders the day layouts. In the original layout they are displayed underneath each other. After the adaptation they are reordered to two columns. The purpose of this adaptation is to adapt the user interface to big screens, like a television monitor. This adaptation can be used to avoid sparse or stretched user interfaces.

The changed elements for this adaptation are shown in Figure 5.13. The change only impacts the layout model and the way the layouts for each day are displayed. Instead of showing these containers underneath each other, the layouts for each two days are wrapped in common parents *mon_tue*, *wed_thu* and fri_sat. These new containers each show two days horizontally next to each other. The outer layout *plan window* contains these layouts and the container for Sunday and displays them vertically underneath each other. This adaptation does not change the other models.

Figure 5.14 shows another adaptation. This adaptation can be used to optimize the layout of a single day in two ways. The first adaptation makes use of more horizontal space by showing all hour labels next to each other to save on vertical space. In the second adaptation the labels for each hour of the day are shown in a grid of 4 x 6 instead

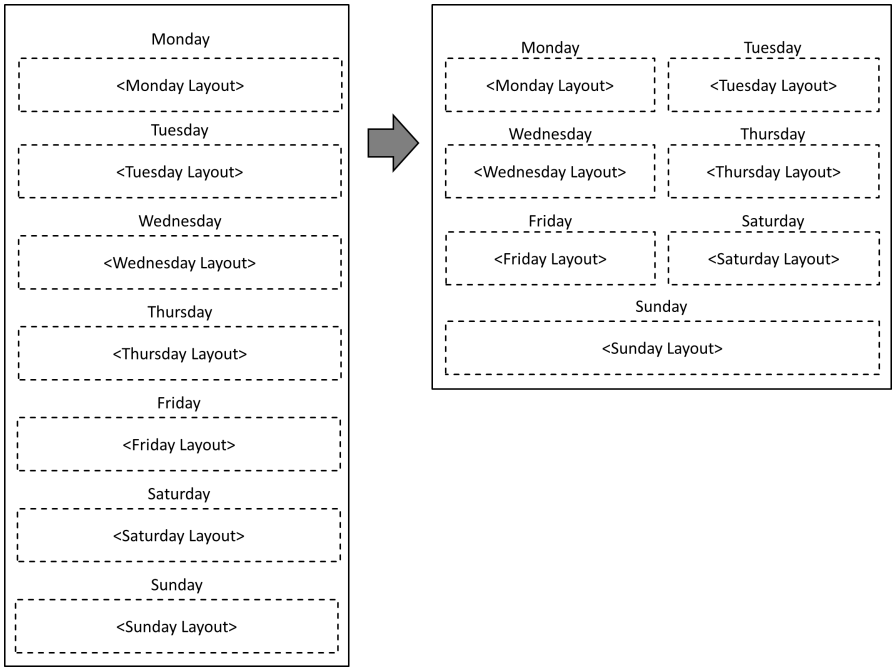


Figure 5.12.: Schematic overview of the second adaptation.

of 2 x 12. This adaptation is used to optimize the layout to give more horizontal space to the labels and avoid small labels on devices with small screens.

Figure 5.15 shows an excerpt of the layout model after the third adaptation. Again, the adaptation does not change the other models. The adaptation changes the part of the layout model that is responsible for displaying the leafs for each hour. It yields an outer layout *monday_content* that contains four containers $monday\_00-05$, $monday\_06-11$, $monday\_12-17$ and $monday\_18-23$. Each of these containers represents a row in the layout and holds six leaves, as indicated for the container $monday\_00-05$. The layout elements above the container *monday_content* are not altered by the adaptation.

The schematic overview of the fourth adaptation is shown in Figure 5.16. This adaptation causes the user interface to show more details about the heating plan. In addition to the colour of the labels, which indicate the heating level, the level is shown as text in the respective labels. This adaptation represents a more detailed level of the semantic zoom for this user interface, which can be triggered by the user.

The changed model elements of this adaptation are shown in Figure 5.17. The adaptation changes the mapping between the UI model and the domain model. The original model contains mappings of type $Domain2UI\_Background$ that are responsible for setting the correct background colour for each hour label to indicate its heating level. The adaptation complements each of these mappings with additional mappings of type $Domain2UI\_Text$ that also set the text of the label to the corresponding heating level. These additional mappings are the only change necessary for this adaptation. All other models remain unchanged.

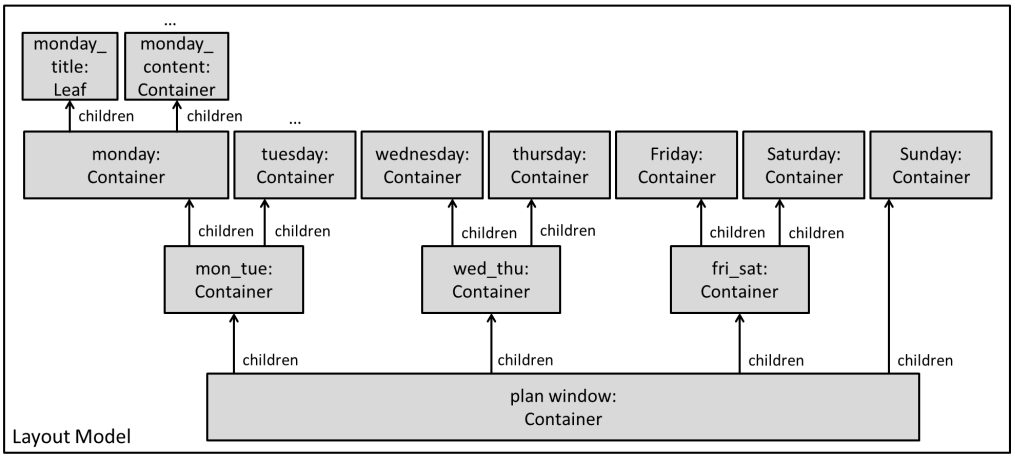


Figure 5.13.: The model after the second adaptation.

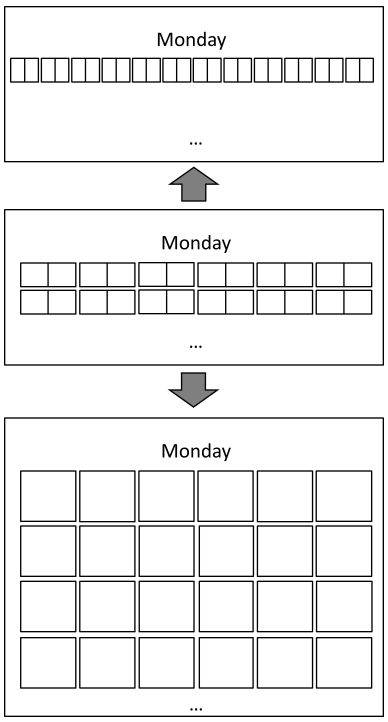


Figure 5.14.: Schematic overview of the third adaptation.

The adaptations presented in this section are implemented in the project SOE. The project implements a rule-based approach that activates these adaptations in the described situations. In addition, they can be activated or deactivated from the Meta-UI. Thus, arbitrary combinations of these adaptations are possible at run time.

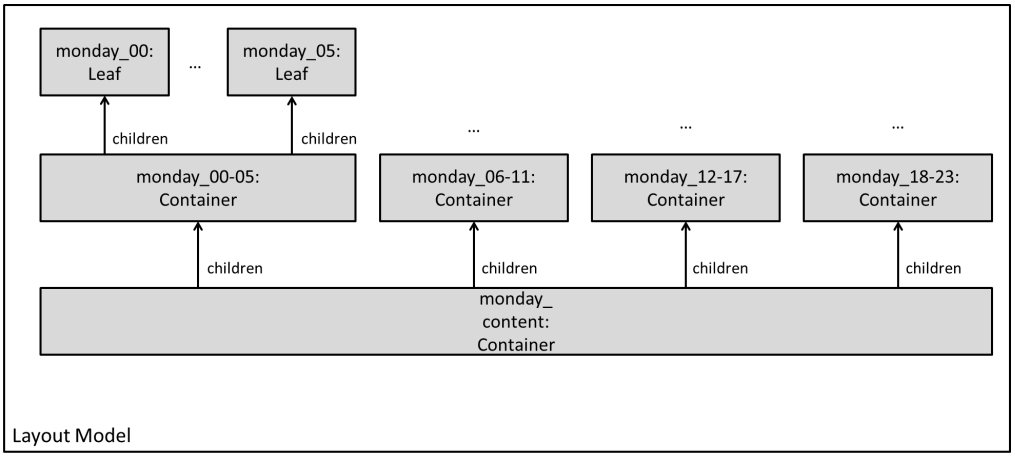


Figure 5.15.: The model after the third adaptation.

.

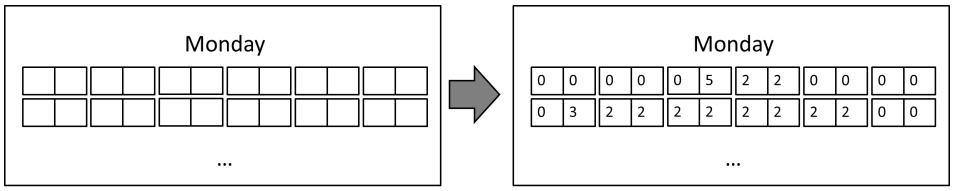


Figure 5.16.: Schematic overview of the fourth adaptation.

The next section describes consistency requirements in the project SOE.

5.4. Consistency Requirements

This section lists several consistency requirements that exist in the models in the project SOE. These requirements concern the models as well as the mappings between them. Several conditions are already contained in the Ecore meta model. For example, this meta model is able to express multiplicities of relations and the eOpposite relation that are effectively constraints on the occurrence of edges in the model. The remainder of this section describes the additional conditions required in this project.

The following consistency requirements exist for the dialog model:

1. **All places are connected to the UI model**: All elements of type *Dialog Place* in the dialog model are connected to at least one *Abstract Interactor* in the UI model via a mapping of type $Place2UI$

2. **All transitions are connected to the UI model**: All elements in the dialog model of type *Dialog Transition* are connected to at least one *Command* in the UI model via a mapping of type $UI2Transition$.

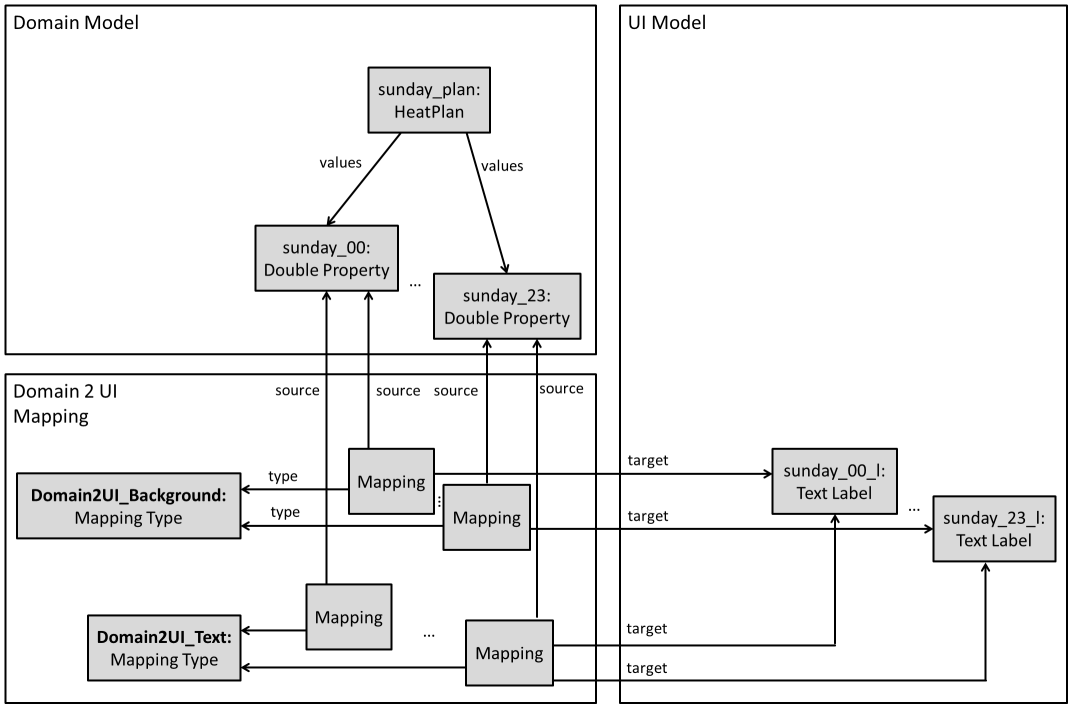


Figure 5.17.: The model after the fourth adaptation.

These two conditions require dialog elements to have meaning. A place in the dialog model needs to be connected to at least one element in the user interface that is shown when this place is activated (Condition 1). Else the place does not have any influence on the software system. Furthermore, each transition needs to be connected to a command that enables it to fire (Condition 2). If this connection does not exist the transition can never fire and is thus obsolete.

In the UI model the following conditions exist:

1. **Each navigation element has influence on the dialog model**: Each element of type *Navigation* is connected to at least one *DialogTransition* in the dialog model via a mapping of type $UI2Transition$.

2. **All UI Elements are connected to exactly one element in the layout model**: Each *ConcreteInteractor* in the UI model is connected to exactly one *Node* in the layout model via a mapping of type $UI2Layout$.

3. **Not more than one background mapping exists**: No *GraphicalPropertySet* in the UI model is connected to more than one property in the domain model via a mapping of type $Domain2UI\_Background$.

4. **Not more than one text mapping exists**: No *TextLabel* in the UI model is connected to more than one property in the domain model via a mapping of type $Domain2UI\_Text$.

The conditions of the UI model concern its connection to the other models via mappings. *Navigation* elements in the user interface inherit from *Commands* and represent elements whose purpose is a navigation between user interface elements. Accordingly, an element of this type needs to be connected to the domain model to achieve the navigation (Condition 1). The connection to the layout model via mapping type $UI2Layout$ determines where in the layout a UI element is shown. Without a mapping of this type the element cannot be shown even when it is active. Accordingly, each UI element needs to be connected to such a mapping (Condition 2). The mappings into the domain model can be used to manipulate certain properties of a UI element like its colour or its text based on values of the domain model. If multiple mappings exist they can conflict with respect to the values they imply for the UI element. Thus, Conditions 3 and 4 restrict the two mapping types of the example to exist once for each UI element.

The following condition exists for the layout model:

1. **All leaves are connected to exactly one UI Element**: All elements of type $Leaf$ in the layout model are connected to exactly one $ConcreteInteractor$ in the UI model via a mapping of type $UI2Layout$.

Elements of type $Leaf$ in the layout model represent a user interface element that is inserted in the layout at the place of this leaf. Accordingly, each leaf must be connected to a user interface element. Additionally, it is not possible to insert two elements for one leaf. Thus, Condition 1 requires all leaves to be connected to exactly one UI element.

For the mapping model the following conditions exist:

1. **Mapping type is contained**: If a $Mapping$ is contained in a mapping model via the relation *mappings* its type also is contained in the same mapping model via relation *mappingTypes*.

2. **Source and trigger of a mapping link are consistent**: The *trigger* of a mapping link is a feature of the $EClass$ that is defined as the *source* of this link.

3. **Target and action of a mapping link are consistent**: The $EOperation$ that is defined as *action* of a mapping link is an operation of the $EClass$ that is defined as the *target* of the link.

4. **Mapping Links are consistent**: All $MappingLinks$ in a $MappingType$ connect the same two types, meaning the same two types are either used as *source* and *target* or *target* and *source*.

5. **Source and target type of mapping is consistent**: The *source* and *target* objects connected by a mapping are instances of the *source* and *target* classes referenced by the $MappingLinks$ of the $MappingType$ of the mapping.

The conditions on the mapping model assure the consistency of mappings. Condition 1 ensures that the type referenced by a mapping is actually contained in the model and available. Furthermore, Conditions 2 and 3 ensure that for each mapping link the feature, which the link listens to, is available in the source class and that the action, which the

condition triggers, is available in the target class. In the SOE project mappings connect two objects. Links can be spanned in any direction between these objects. Accordingly, the links of a mapping type have to connect the same two types, but might differ in which of them is the source and which is the target. Condition 4 ensures that all links of a mapping connect the same two types. A mapping is inconsistent if it relates objects that cannot be used as intended by its mapping links. Condition 5 assures this by requiring the mapped objects to conform to the correct types.

5.5. Summary

In this chapter we described the case study that will serve as a running example and evaluation of the Trollmann approach. The case study is taken from the project SOE. It represents an adaptive user interface for heating control. The project SOE contains four models: A dialog model, a UI model, a layout model and a domain model. We exemplified these models on the user interface of a plan window. This window will be used as running example in the next chapter. In addition, we presented several adaptations that optimize this window to different situations and several structural conditions that have to be fulfilled by the models in this project.

In the next chapter we present the Trollmann approach to the detection of adaptation conflicts. The case study is used to illustrate this approach.

6. The Trollmann Approach

This chapter presents the Trollmann approach for detecting adaptation conflicts at run time. A model-based framework has to fulfil a set of assumptions for this approach to be applicable. Section 6.1 articulates these assumptions. Subsequently, we give a conceptual overview of the Trollmann approach is in Section 6.2. Section 6.3 presents a running example, an excerpt of the case study discussed in Section 5, to illustrate the Trollmann approach effectively.

The remaining sections describe the Trollmann approach in detail, including the formalism for representing models, model relations, adaptations and consistency requirements (cf. Section 6.4), the algorithms for conflict detection (cf. Section 6.5) and an explanation on how these algorithms can be used to provide information for conflict resolution (cf. Section 6.6). We conclude the chapter with a summary.

6.1. Prerequisites

The Trollmann approach relies on run time models to analyse adaptation conflicts. The analysis methods are based on a set of formalisms that represent models, adaptations and consistency requirements. Thus, the approach makes a set of assumptions about the modelling framework. It can be applied in any framework that fulfils them. In this section we list and describe these assumptions.

Assumption 1. *For each aspect that should be analysed with respect to adaptation conflicts there is at least one run time model that represents this aspect.*

The analysis methods of this approach are based on models. Accordingly, all aspects that need to be analysed for adaptation conflicts have to be expressed as models for this method to be applicable. Since the analysis method aims to be applied at run time these models have to be available while the software system is running.

Assumption 2. *The analysed models are synchronized with the software system via a causal connection.*

The analysis is based on models and assumes that the analysed models correctly reflect the software system. Thus, Assumption 2 is that the models are in causal connection to their system under and study. They are models@run.time (cf. Section 2.1.6). Retaining the consistency between evolving models and the evolving SUS can be a complex task and has been identified as one of the key challenges of run time assurance [10]. However, since the synchronization of model and SUS is not a focus of this thesis we assume that both are synchronized correctly.

Assumption 3. *All consistency requirements that need to be analysed are expressed as structural conditions on models.*

The analysis methods also require an explicit representation of consistency requirements based on the analysed models. Accordingly, Assumption 3 is that all requirements that need to be analysed are expressed explicitly based on the model structure.

Assumption 4. *All adaptations that need to be analysed are described as reconfiguration rules that can be executed to change the models accordingly.*

Similarly, it is only possible to analyse adaptations that are represented on the basis of models. Thus, Assumption 4 is that all adaptations that need to be taken into account in the analysis are expressed on models. We assume that a reconfiguration rule exists for each adaptation. A reconfiguration rule is a specification of the adaptation that can be executed to trigger the adaptation. This explicit representation enables us to analyse the changes made by the adaptation without executing it.

Assumption 5. *The models, adaptations and consistency requirements are either represented in the formalism specified in Section 6.4 or can be converted into this formalism.*

The previous assumptions require models, consistency requirements and adaptations to be represented explicitly in the run time models but do not assume any specific formalism. For the analysis it is optimal to use the formalism specified in Section 6.4 directly. However, the analysis method is also applicable in frameworks that use other formalisms, as long as a mapping to our formalism is possible.

Based on these assumptions we describe the conceptual overview of the approach in the next section.

6.2. Conceptual Overview

This section provides a conceptual overview of the approach. The approach relies on a run time framework that meets the assumptions made in the previous section.

As stated in Assumption 5 we assume that the languages used for describing models, consistency requirements and adaptations are at least translatable into our formalism. Some frameworks already utilize models represented in other formalisms. If these formalisms are sufficiently close to be represented by our formalism, the approach can be applied. In this case both formalisms can be seen as two syntaxes of the same model, similar to the concept of abstract and a concrete syntax. This distinction is normally made in programming languages where the concrete syntax is the syntax the programmer works on and the abstract syntax is a version of the same program that is formalized to be processable by the software systems. Our formalism can be interpreted as an alternate syntax for models that enables the analysis of adaptation conflicts. In [78] we proposed to make this distinction and to formalize consistency of models based on the abstract syntax, regardless of the actual modelling language. Our formalism is an extension of the ideas presented in this publication.

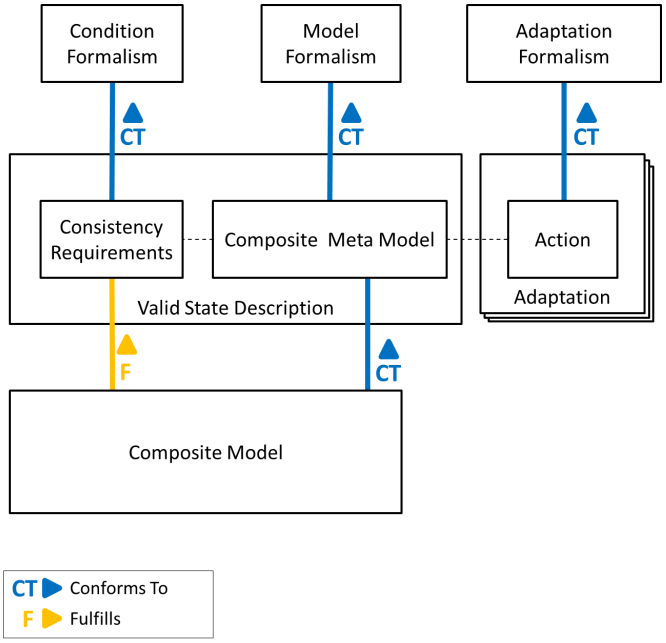


Figure 6.1.: The formalisms in the Trollmann approach and its relation to run time models.

As a basis for analysis we define three formalisms. These formalisms and their relations to the run time models are depicted in Figure 6.1. The three formalisms are called the *Model Formalism*, the *Condition Formalism* and the *Adaptation Formalism*. The model formalism is the basis for the description of models. The condition formalism can be used to formulate consistency requirements. The adaptation formalism is a formalism for the description of adaptations.

The *Model Formalism* is a meta-meta model for the run time models. It can be used to describe the specifics of several modelling languages (e.g., petri nets or UML models). This is done in the *Composite Meta Model*. The word composite indicates that this is not necessarily a meta model for a single model. The composite meta model can describe a set of modelling languages and relations between them. The specific languages and relations used in one modelling framework can be specified in the composite meta model. The model formalism itself is independent of the actual modelling languages used.

The *Composite Model* contains a set of run time models, each conforming to one language described in the composite meta model. The models in the composite model are related with each other. Again, the relations conform to the relation types described in the composite meta model. The description in the composite meta-model is assumed to be complete. It is assumed that all models that conform to the composite meta model can be interpreted by the run time framework.

The *Condition Formalism* can be used to describe *Consistency Requirements*. These requirements constrain the structure and relations of the composite model and thus are

formulated based on the composite meta model. They use elements from this meta model. This is indicated by the dashed line between both boxes in Figure 6.1. The conditions could be further enhanced with values indicating their importance. For example, it might be possible to distinguish between *must-have* conditions that need to be fulfilled and *nice-to have* conditions that should be fulfilled but will not lead to a system malfunction if they are not. Additionally, the origin of the conditions could be important. For example, conditions required by the run time framework might be treated differently than conditions defined by the designer. This additional information plays a big role during conflict resolution. Depending on the importance and origin of the condition different resolution strategies can apply. However, all conditions are treated the same by conflict detection as adaptation-consistency conflicts need to be detected for all conditions. Therefore, the formalism abstracts from details such as importance and origin of consistency conditions.

The combination of the composite meta model and the consistency requirements forms a *Valid State Description*. This description specifies when a model is considered valid from the frameworks point of view and when not. It takes into account the interpretability of the model as well as additional consistency requirements. Every modelling framework can have its own valid state description tailored to the specifics of this framework. This valid state description might even be specific to a software application if it contains conditions by the applications developer.

The *Adaptation Formalism* is a formalism for specifying actions that determine reconfigurations of the model and thus adaptations of the software system. An adaptation consists of an *Action*. The action describes how the composite model is changed when the adaptation is executed. Each action is a description of a reconfiguration of models that conform to the composite meta model and thus has a reference to the composite meta model (as indicated by the dashed line). The actions are specified using the adaptation formalism. The adaptation formalism makes no assumptions about when an adaptation is triggered. This can vary depending on the adaptation strategy.

Figure 6.2 depicts the information that is used as basis for the analysis. The composite model C is the current state of the software system. A set of adaptations $Adaptation_i$ is supposed to be applied to the composite model. All adaptation actions A_i are supposed to be applied, resulting in the new state C'. The composite meta model, expressed in the model formalism, and the consistency requirements, expressed in the consistency formalism, specify when C' is considered valid. The analysis methods in the Trollmann approach can be used to answer whether there is an unambiguous C' as a result of the application of all adaptation actions (adaptation-adaptation conflicts) and whether C' is conform to the composite state meta model and fulfil the consistency requirements (adaptation-consistency conflicts). In case of conflicts the analysis methods provide information that can be used by a conflict resolution mechanism. The analysis uses the current composite model C, the adaptation actions A_i, the composite meta model and the consistency requirements.

In the next section we describe our running example. Subsequently, we describe the model, adaptation and consistency condition (cd. Section 6.4) and the analysis methods (cf. Section 6.5).

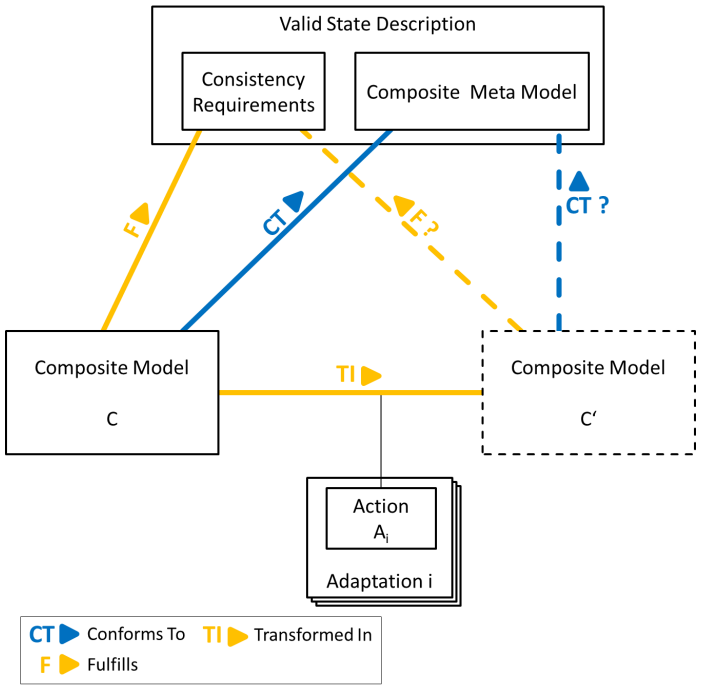


Figure 6.2.: A transformation of a composite model.

6.3. Running Example

We use a running example to illustrate our formalisms and algorithms. The running example is meant as a toy example which is complex enough to exemplify the approach and simple enough to provide good illustration for our definitions. The models of the running example are loosely based on the models of the project SOE described in Section 5, but are strongly simplified. These models are used to implement a simple adaptive software application for making tea.

The running example focuses on the UI model, the dialog model and their relation. Section 6.3.1 contains the simplified version of these two models and the user interface of the tea application. In Section 6.3.2 we describe consistency requirements in the scope of this software system. In the last section we showcase two adaptations.

6.3.1. UI and Dialog Model

The running example consists of a UI model and a dialog model. Both represent simplified versions of the models in the case study. The UI model abstracts from the separation of the UI model and layout model in the project SOE and contains elements from both. It contains a set of concrete graphical UI elements that, starting from a window, describe a tree of graphical layouts and UI elements that represents the content layout of the window. This UI model encompasses the concrete UI Elements contained in the SOE UI model and the layout-tree described in the SOE layout model. The dialog model

determines the order in which the windows are displayed to the user. It also defines the interactions that trigger a progression in the dialog. This model is similar to the dialog model in the project SOE. The domain model is not reflected in the running example.

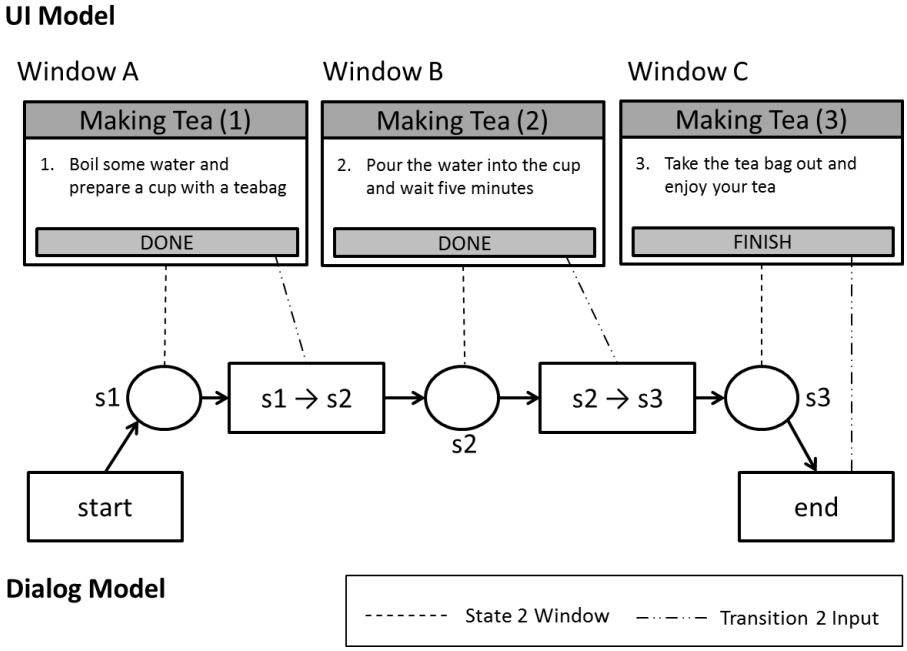


Figure 6.3.: Models of the running example.

As a simple example application Figure 6.3 depicts an application for making tea. This software system provides the user with stepwise information on how to make a cup of tea. Each step is contained in its own window. These windows are contained in the UI model in the upper part of Figure 6.3. Overall there are three steps and for each step there is a window (*Window A* to *C*) that contains a description of the step and a button. Pressing the button indicates that the current step has been completed. After the user clicks the button the current window is removed from the visible user interface and the window for the next step is shown. This is managed by the dialog model.

The dialog model of the tea application contains 3 places, representing the three windows of the software system. Whenever a place is marked, the according window is shown. This is indicated by the relation of type *State 2 Window*. This relation is a simplified version of the mapping type *Place2UI* from the case study. The transition *start* places a token on place *s1*. It is called once, when the tea application is started. All other transitions are triggered by pressing a button in the user interface. This is represented by the relation *Transition 2 Input*. This relation is a simplified version of the mapping type *UI2Transition* from the case study. Each transition except for *start* and *end* move the token from one place to another, thus closing one window and opening the next one. The transition *start* does not close a window and the transition *end* does not open a window.

In the tea-application both models are initialized as they are depicted in Figure 6.3. When the application is started the transition *start* is called. This transition places a token on the place *s1*. Accordingly, the associated *Window A* is shown to the user and presents the first step of making tea. When the user is done with this step she clicks on the button labelled *DONE*. This triggers the transition $s1 \rightarrow s2$. While firing, this transition moves the token from *s1* to the place *s2*. Accordingly, *Window A* is removed from the visible user interface and *Window B* is added. Similarly, after the user finishes the next step and clicks the button labelled *DONE* the token moves to place *s3* and *Window 2* is shown. After finishing the last step the user can click on *FINISH* to finish the application by triggering the transition *end*. This transition removes the token from *s3*, which leaves the dialog model tokenless.

The next section describes the consistency requirements these models have to fulfil to be interpreted correctly at run time.

6.3.2. Consistency Requirements

In this section we introduce consistency conditions for the tea application.

In the examples the following set of conditions is used:

i) *Only one place is marked at a time.* The run time framework is only able to show one window at a time to the user. This restriction can occur on mobile devices with limited screen size. The number of shown windows is directly derived from the number of places that contain a token. Therefore, the number of places marked at any time is restricted to one.

ii) *Each window has content.* If a window has no content it cannot be displayed to the user. Therefore, each window is required to contain a user interface.

iii) *Every place is associated to exactly one window.* A place that is not associated to a window represents a state of the software system that does not have any user interface. Combined with the requirement for only one marked place this indicates a state of the software system that has no user interface at all. Since all transitions, except for the starting transition, are triggered by an interaction with a user interface element, a state without user interface elements represents a deadlock. Thus, every state has to be associated to a window. It cannot be associated to more than one window because this would require the software system to display multiple windows at the same time.

iv) *There is exactly one start transition.* The start transition is fired when the software system is started. Before firing this transition the marking is not existent. Thus, the transition initializes the dialog model. If no start transition exists the software system cannot be started. Firing two start transitions can lead to two visible windows, which is forbidden. Thus, exactly one start transition should exist.

v) *Every transition that is not a start transition has to be associated to a UI element.* In the running example transitions can only be triggered by interaction with UI

elements. The only exception is the start transition. To prevent transitions that cannot be triggered all transitions with the exception of the start transition are required to be connected to at least one UI element.

vi) *When a transition is activated its triggering UI element has to be visible.* A transition is activated when the places on its incoming arcs contain enough token to fire it. A transition associated to a UI element is triggered when the UI element is interacted with. However, if the UI element is not in the window associated to the transitions source place the UI element is not shown when the transition is activated and the transition can never be fired. Therefore, the run time framework requires triggering UI elements to be contained in a window that is associated to the source place of a transition.

Some of these consistency conditions concern single models and some concern the relations between models. Conditions i) and iv) restrict the dialog model while condition ii) only concerns the UI model. Conditions iii), v) and vi) concern both models. These conditions also restrict their relation to each other.

More requirements can be enumerated. For example requirements can be used to assure that the dialog model will always contain only one token in all of its future states. However, for the running example the above mentioned requirements are sufficient. These requirements have been defined from the perspective of the modelling and run time framework. One other potential source of requirements is the designer. While the requirements of the run time framework deal with interpretability of the models the designer might require additional conditions that ensure that the software system correctly fulfills its purpose.

The purpose of the sample application is to make tea. For this reason the designer wants to assure the correct presentation of all steps to the user, even when the application itself is adapted. The following requirements have to be fulfilled:

vii) *All three steps are contained in the user interface of a shown window.* The process of making tea takes three steps. The designer wants to assure that all of these steps are contained in the visible user interface of the software system. This is assured by a condition that requires the corresponding labels to be contained in a window that is associated to a place.

viii) *The order of steps is correct.* The steps are supposed to be executed in a certain order. The designer of the software system wants to assure that the steps are presented to the user in the correct order. This is necessary because some adaptations can change the order of windows, e.g., by uniting multiple windows. The designer wants to assure that step 2 is either shown in the same window as step 1 or in the window after step 1 and step 3 is shown in the same window as step 2 or in the window after step 2.

The reason why the designer might want to add additional requirements is the fact that the software system is adaptive. Depending on the implementation of the framework the set of adaptations may not be fully under the designers control as there can

be adaptations on framework-side or even adaptations defined at run time. In addition, these requirements can be used as a safety net in case the adaptations or their combination have unexpected side-effects on the software systems workflow. Both conditions of the designer restrict both models.

The next section describes adaptations for the tea application.

6.3.3. Adaptations

This section describes adaptations for the tea application. The adaptations in this example are described from the perspective of the developer of the software system. In other cases the run time framework may also provide adaptations.

For representing the changes encompassed in the adaptations a minimalized before-after notation is used. The view of the user interface before the adaptation is shown on the left hand side of an arrow. The interface after the adaptation is shown on the right hand side. The images only contain elements that are important for the adaptation and are required or changed by the adaptation.

The following two adaptations are used in the running example:

Adaptation 1: Back Button

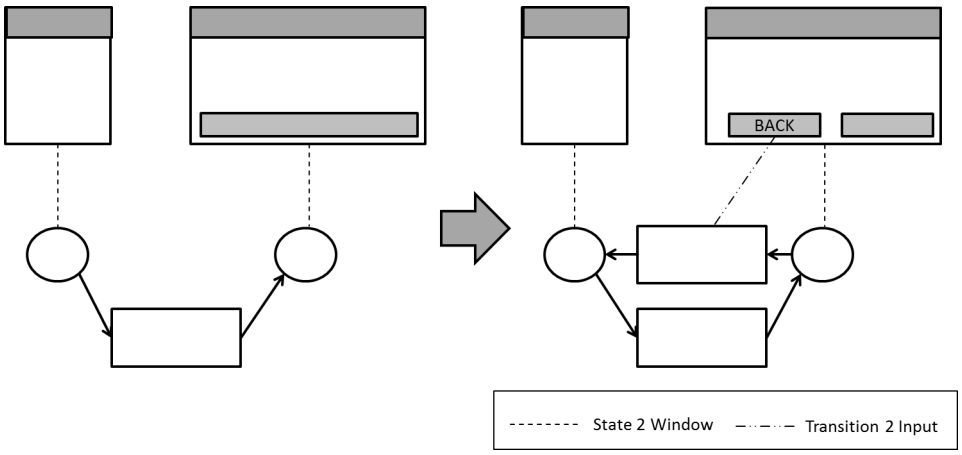


Figure 6.4.: Adaptation 1 of the Running Example: adding a back button.

The first adaptation is depicted in Figure 6.4. This adaptation adds a back button to one window. When this back button is pressed the previous window is shown. As precondition (left hand side) the adaptation requires two windows that are associated to two places, connected by a transition. The second window contains a button. The unspecified elements (for example the titles of the windows, the content of the first window or the label of the button) are not important for the adaptation and are thus not drawn in the figure. The adaptation changes the size of the button in the second window and adds a new button, labelled *BACK* on the left of this button. This button

triggers a newly added transition that leads from the second place back to the first place. When this button is clicked the current window is removed from the visible user interface and the previous window is shown.

The adaptation can be applied to any two windows that are consecutive in the dialog flow if the second window contains a button. In the example in Figure 6.3 the adaptation is applicable in two ways. The back button may be added between *Window B* and *Window A* or between *Window C* and *Window B*. Thus, by applying the adaptation twice the user can be enabled to navigate back and forth between all windows.

Adaptation 2: Uniting Windows

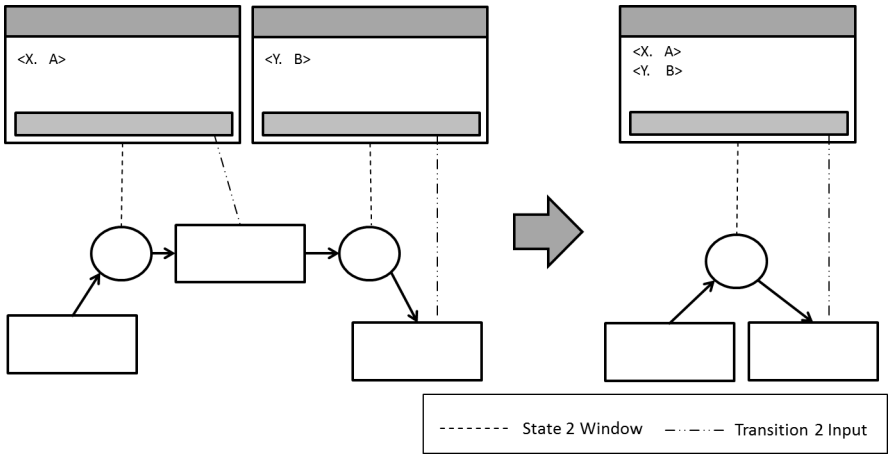


Figure 6.5.: Adaptation 2 of the running example: uniting two windows.

The purpose of Adaptation 2 is to unite two windows and show their respective steps in the tea-making process underneath each other. The left hand side requires two windows, which each have one button and one label of arbitrary content (the format $\langle X.A \rangle$ is used to denote arbitrary content). In addition, the rule assumes that the places for both windows are connected by a transition and that there is a transition leading to the first place and a transition leading from the second place (both may or may not have connections to other places that aren't shown in the figure). During the adaptation the windows are united and the label of the second window is added under the label of the first window. Both places as well as the transition between them are removed and a new place, associated to the new window, is added. The left and right transitions are preserved and connected to the new place.

Again, the adaptation is formulated general enough to be applied in two ways. Either steps 1 and 2 or steps 2 and 3 may be united by applying the adaptation.

6.4. Formalism

This section describes the formalisms that are the foundation for adaptation conflict detection. Based on an analysis of existing formalisms we decided that graph transformation for the representation of adaptations and nested conditions for the representation of consistency requirements are a suitable basis for the detection of adaptation conflicts. Both formalisms are well formalized and a large body of analysis results already exists that can be applied. Especially the notion of parallel dependencies (cf. Section 2.2.4) lends itself to be applied for the detection of adaptation-adaptation conflicts.

These two formalisms and the existing theoretical results can be applied to any $\mathcal{M}$-adhesive category. As argued in Section 4.2.1 attributed typed graphs are a suitable foundation for the representation of models of arbitrary modelling language, but are not able to represent multiple models and their relation without uniting them. Since the existing extensions of attributed typed graphs are also limited either in their power of expression or are not $\mathcal{M}$-adhesive we define a new formalism, called graph diagrams, which is based on attributed typed graphs and is able to represent multiple models and model relations. This model formalism is described in Section 6.4.1.

Section 6.4.2 describes nested conditions as condition formalism and their application to graph diagrams. The combination of graph transformation as adaptation formalism with graph diagrams is described in Section 6.4.3. We exemplify all formalism by using the running example from Section 6.3.

6.4.1. Model Formalism

This section describes the model formalism. The definition of a suitable model, condition and adaptation formalism is Contribution 1 of this thesis. From this contribution we can derive the following requirements for the model formalism:

- *Generality:* The model formalism should be able to express models of arbitrary modelling languages.
- *Multiple Models:* The model formalism should be able to express multiple models of different modelling languages.
- *Model Relations:* The model formalism should be able to represent relations between models.

We describe our model formalism in a stepwise fashion. As a basis the model formalism for elementary models without relations is described. This formalism is based on attributed typed graphs as representation for elementary models. Subsequently, we define graph diagrams as an extension of attributed typed graphs for representing multiple models and model relations. Finally, we extend the model formalism for the representation of composite models by using graph diagrams.

Formalism for Elementary Models

In this section we describe the model formalism for elementary models on the basis of attributed typed graphs. Elementary models are single models. Relations to other models are not represented in this formalism. This section contains the definition of an elementary model and elementary meta model. These definitions are extended to composite models and composite meta models in the following sections.

Elementary models are attributed typed graphs from the category $AGraphs_{ATG}$. However, as stated in Section 2.2.5 this category is not finitary and does not contain an $\mathcal{M}$-initial object. To derive a category that fulfils both criteria the category is used in a restricted version, called *algebra-restricted*. The restriction is twofold. On the one hand, the category is restricted to contain only a finite number of nodes, edges and attributes. This restricts the category to ATGs that have a finite structure. In addition, the data that can be used for attributes is fixed to the data contained in a specific algebra. The restricted categories of attributed graphs and attributed typed graphs are defined in the following Definition:

Definition 4 (Algebra Restricted Attributed Graphs and Attributed Typed Graphs)**.** *The category* $\boldsymbol{AGraphs}^D$ *of algebra restricted attributed graphs over an algebra* D *is a subcategory of the category* $\boldsymbol{AGraphs}$ *that contains only objects* $AG = ((V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j\in\{G,NA,EA\}}), D)$ *for which the sets* V_G*,* E_G*,* E_{NA} *and* E_{EA} *are finite and all morphisms* $f = ((f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}}), f_D)$ *between these objects for which* f_D *and* f_{V_D} *are identities.*

Given an algebra D *for signature* $DSIG$*, a* D*-type graph is an attributed type graph over the final* $DSIG$ *algebra.*

Given an algebra D *and a* D*-type graph* ATG *the category* $\boldsymbol{AGraphs}^D_{ATG}$ *of algebra restricted attributed typed graphs over* D *is a subcategory of the category* $\boldsymbol{AGraphs}_{ATG}$ *that contains only attributed typed graphs* (AG, t) *for which* AG *is from* $\boldsymbol{AGraphs}^D$ *and only morphisms from* $\boldsymbol{AGraphs}^D$*.*

Although both restrictions limit the expressivity of attributed typed graphs they seem reasonable in the scope of most modelling frameworks. The restriction to finite attributed graphs means that all models have a finite amount of nodes, edges or attributes. Since software models need to be expressible within the limited storage of a computing system anyway this restriction is reasonable. The restriction to one algebra means that the data types used for expressing attribute values are fixed. Within one model it is a reasonable assumption that these data types do not change at run time.

The implementation of the elementary model and elementary meta model is depicted in Figure 6.6. An elementary model is represented as an algebra-restricted attributed typed graph. Definition 5 defines the elementary model.

Definition 5 (Elementary Model)**.** *Given an algebra* D *and a* D*-type graph* ATG*, an elementary model is an algebra-restricted attributed typed graph from* $\boldsymbol{AGraphs}^D_{ATG}$*.*

The elementary meta model describes the modelling language that can be used to define elementary models. Thus, the elementary meta model has to describe a set of

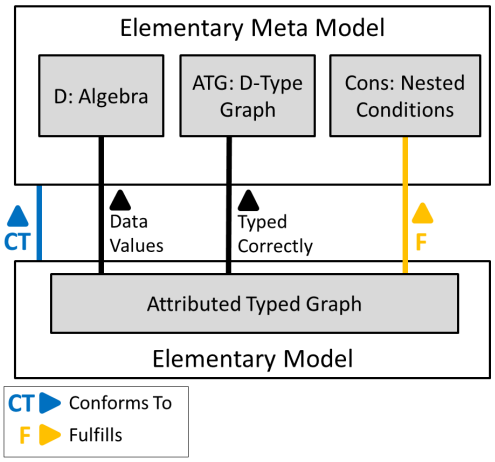


Figure 6.6.: The model formalism for elementary models.

attributed typed graphs. This is done by three components: an algebra, a type graph and a set of nested conditions. The algebra describes the data elements that can be used to annotate attributes in the model. The type graph defines the types of nodes, edges and attributes that can be used. The structure of the modelling language can be further restricted by using nested conditions (see Section 2.2.3). These can be used to add further structural requirements that have to hold in all models. The elementary meta model is described in Definition 6.

Definition 6 (Elementary Meta Model)**.** *An elementary meta model consists of an algebra D, a D-type graph ATG and a set of nested conditions $cons$ in $\boldsymbol{AGraphs}^{D}_{ATG}$. An elementary model is conform to this meta model if it is an object from $\boldsymbol{AGraphs}^{D}_{ATG}$ and fulfills all constraints in $cons$.*

This definition of elementary models and meta models is independent of modelling languages. Models of a variety of modelling languages can be expressed as attributed typed graphs, with a specific set of types, attributes and nested conditions. We discussed the generality of attributed typed graphs for representing arbitrary modelling languages as part of the related work in Section 4.2.

The running example contains two elementary models: the UI model and the dialog model. In the remainder of this section we describe the elementary meta model and elementary model of both models in the elementary model formalism.

The elementary meta model of the UI model is depicted in Figure 6.7. The type graph can be seen in the top part of the figure. It consists of types for the elements *Window*, *UI Element*, *Layout Vector*, *Label* and *Button*. The type *UI Element* is a parent node for all other UI elements and declares the general attributes *width* and *height* that each UI element has. *Labels* and *Buttons* are UI elements that contain a value which describes the text they present to the user. A *Layout Vector* is an element that contains a set of *children* which it layouts next to each other either in horizontal or vertical direction

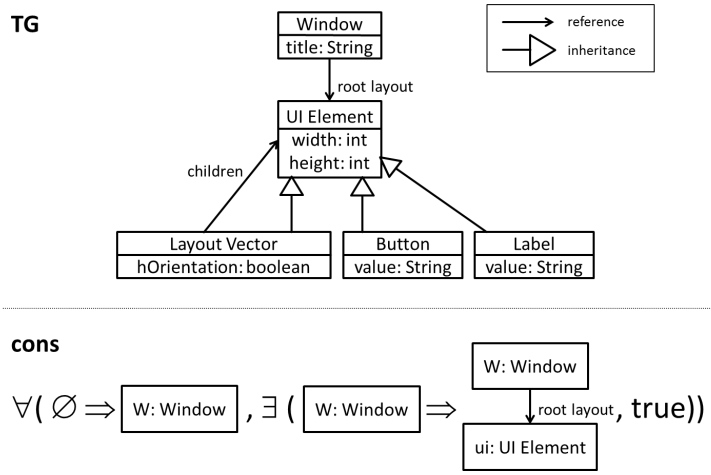


Figure 6.7.: The elementary meta model of the UI model in the running example.

(depending on the value of the attribute *hOrientation*). The node type *window* contains one attribute *title* and a reference to its *root layout*.

The elementary meta model contains one nested condition. This condition is responsible for assuring that each window has a content (Condition 2 in Section 6.3.2). The outer condition consists of an all-quantifier over all windows that can be found. An inner existence quantifier states that for each of these windows a UI element, connected by an edge of type *root layout* has to exist.

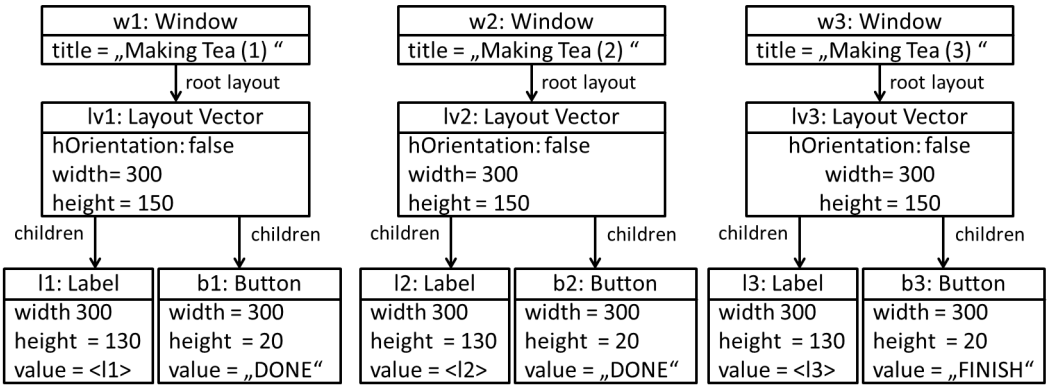


Figure 6.8.: The elementary model of the UI model in the running example.

A conforming elementary model is depicted in Figure 6.8. This model is a representation of the user interface of the running example. It consists of three windows of similar structure. Each window has a layout vector of vertical orientation as root layout. This layout vector contains one label for displaying the respective step of the tea-making process to the user (due to its length the content of the label is abbreviated by $\langle l1 \rangle$ to $\langle l3 \rangle$)

and one button for showing the next step or finishing the software system respectively. This elementary model is correctly typed (indicated by the types annotated in its nodes and edges) and fulfils the condition. Thus, it conforms to the elementary meta model of the UI model in Figure 6.7.

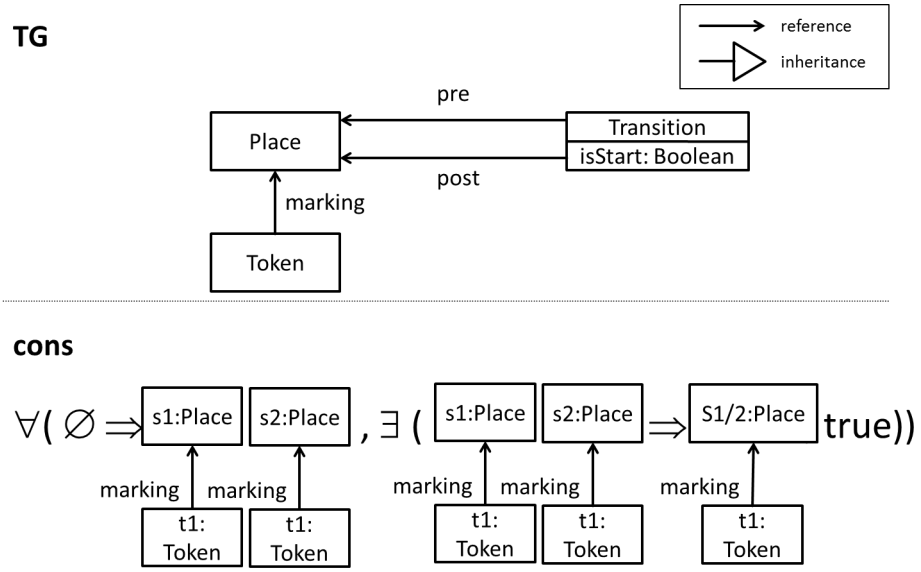


Figure 6.9.: The elementary meta model of the dialog model in the running example.

The elementary meta model of the dialog model is depicted in Figure 6.9. The type graph *TG* contains three node types: *Place*, *Transition* and *Token*. The connections between places and transitions are represented by the edge types *pre*, representing an incoming edge, and *post*, representing an outgoing edge. Tokens are associated to the places they are placed on by edges of the type *marking*. Start transitions are marked by the attribute *isStart*.

The figure also exemplifies one constraint of the dialog model. This constraint corresponds to Condition 1 in Section 6.3.2. It states that only one place can be marked with a token at a time. The outer condition consists of an all-quantified constraint that requires two marked places (represented by a place and a token connected by an edge of type *marking*). The inner condition states that these two places have to be the same. The condition is fulfilled whenever there are no two different marked places.

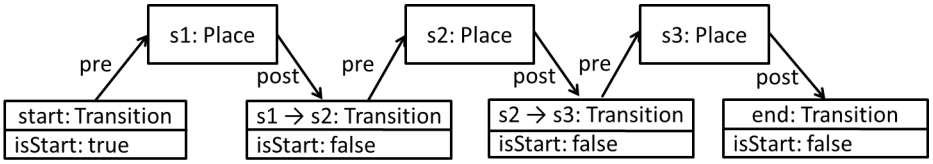


Figure 6.10.: The elementary model of the dialog model in the running example.

A conforming elementary model for the dialog model is depicted in Figure 6.10. It contains the three places $s1$ to $s3$ and the transitions between them. The transition *start* is marked as a start transition by the value of the attribute *isStart* All other transitions are regular transitions and need to be connected to the UI model. They are triggered by interaction with UI elements.

In the next section we describe how the formalism of attributed typed graphs can be extended to graph diagrams. This enables the representation of the relation between the UI and dialog model.

Graph Diagrams

This section describes the formalism of graph diagrams. This formalism builds on the power of expression of attributed typed graphs and enables the representation of multiple models. A graph diagram consists of a set of attributed typed graphs, representing models, connected by ATG morphisms, representing relations. In this section the formal definition of this formalism is given within the framework of category theory. This definition is derived from the category of attributed graphs via categorial constructions.

Definition 7 (Category $\mathbf{GraphDiagrams}_S$)**.** *Given a small category S, the category $\boldsymbol{GraphDiagrams}_S$ of graph diagrams over S is the functor category of all functors from S to **AGraphs**. S is called the scheme of the diagram.*

The definition of the category of (untyped) graph diagrams is given in Definition 7. The category is defined in relation to another category S via functor category construction. S is called the scheme of the diagram. It defines the diagram's structure. The objects in S define which attributed graphs have to exist in the diagram and the morphisms in S define which attributed graphs are linked by morphisms. The category $\mathbf{GraphDiagrams}_S$ contains all graph diagrams of this structure. The following fact describes the objects and morphisms of this category.

Fact 1 (Graph Diagrams and Graph Diagram Morphisms)**.** *A **graph diagram** with scheme S is a functor $D = (O, M) : S \to AGraphs$ where $O : Objs_S \to Objs_{AGraphs}$ maps objects from S to attributed graphs and $M : Morphs_S \to Morphs_{AGraphs}$ maps morphisms from S to attributed graph morphisms.*

*A **graph diagram morphism** m between two diagrams $D_1 = (O_1, M_1)$ and $D_2 = (O_2, M_2)$ with scheme S is a natural transformation, consisting of a family of morphisms in **AGraphs**. For each $o \in Objs_S$ there is a morphism $m(o) : O_1(o) \to O_2(o) \in m$. For each morphism $e : o_1 \to o_2 \in Morphs_S$ the following statement holds: $m(o_2) \circ M_1(e) = M_2(e) \circ m(o_1)$.*

*A graph diagram $D = (O, M)$ is **empty** if $O(o)$ is empty for all $o \in Objs_S$. An attributed graph is empty if the components V_G, E_G, E_{NA} and E_{EA} are empty.*

A graph diagram is a functor between the category S and the category **AGraphs**. This functor maps the objects and morphisms in S to objects and morphisms in **AGraphs**. It describes which object in the graph structure is represented by which attributed graph.

A graph diagram is empty if it contains only empty attributed graphs. An empty diagram is not defined as a diagram with no nodes or edges but as a diagram where all nodes are empty graphs and all edges are empty graph diagram morphisms. If the graph diagram represents a set of related models the empty diagram represents the case where all of these models are empty.

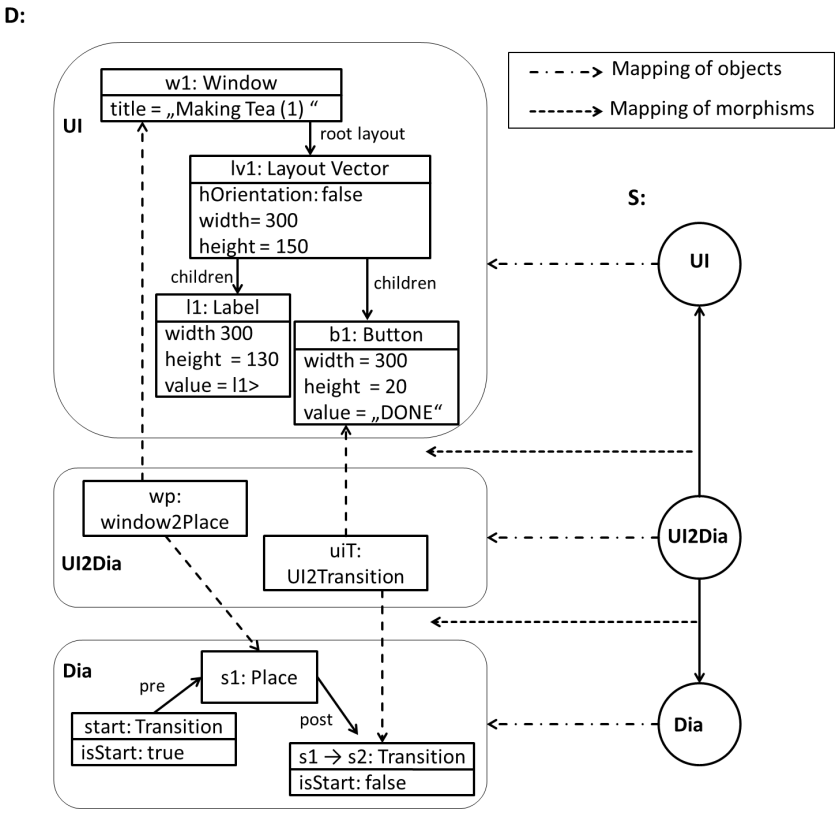


Figure 6.11.: Example: A graph diagram.

An example for a graph diagram is depicted in Figure 6.11. This example is taken from the running example. The scheme of the diagram is depicted on the right hand side. This category consists of three objects, representing three models. The object *UI* represents the UI model, the object *Dia* represents the dialog model. Attributed graph morphisms need to completely map all elements in one model to elements in another model. The partial relation of the elements in the UI and dialog model cannot be represented by a morphism. We express this relation by introducing an the additional object *UI2Dia*. This object summarizes the related elements and is mapped into the UI and the dialog model by two morphisms.

The scheme contains two morphisms, each mapping $UI2Dia$ to one of the other two models. The scheme contains three additional morphisms, not shown in the figure. These are the identity morphisms on UI, Dia and $UI2Dia$. The existence of these morphisms

is required since the scheme is a category. However, since these morphisms are trivial (they map elements to themselves) we omit them from the illustrations to save space.

As specified in the scheme, the graph diagram D, depicted on the left hand side of Figure 6.11, contains three models. It contains one UI model UI and one dialog model Dia. UI is a part of the UI model represented in Figure 6.8 and represents the window $w1$ for the first step of making tea. Dia is a part of the dialog model represented in Figure 6.10 and represents the according state $s1$ in which the window $w1$ is shown, as well as the start transition $start$ and the transition $s1 \rightarrow s2$ that is triggered from $b1$. The third model $UI2Dia$ contains representations of the related elements in UI and Dia. It contains one element wp, denoting that the window is shown when the place is marked, and one element uiT, denoting that the transition is fired when the button is clicked. The relation is established by mapping these elements to the related elements in UI and Dia via the two morphisms.

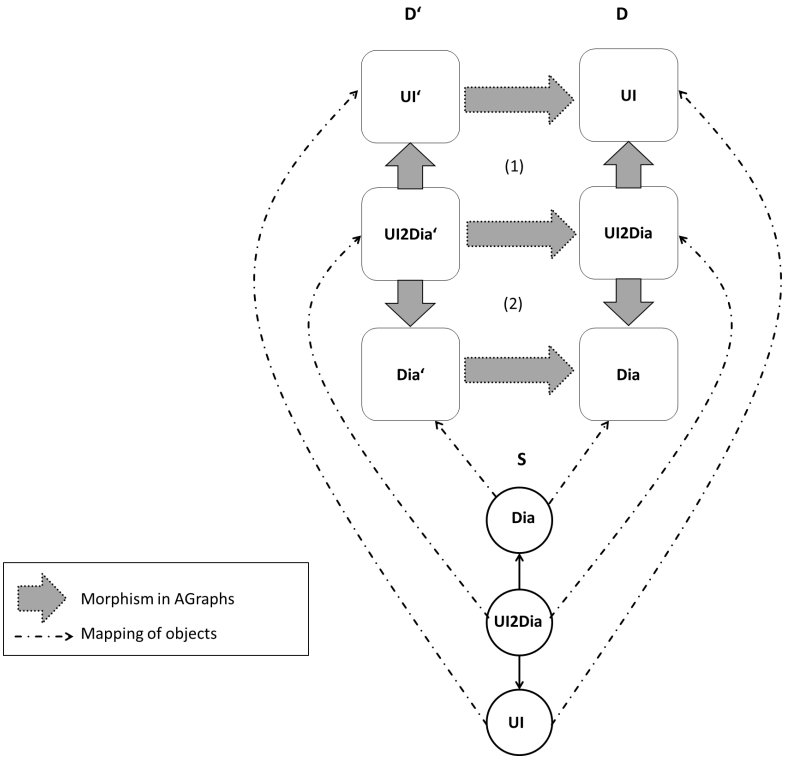


Figure 6.12.: Morphisms of graph diagrams are natural transformations.

As stated in Fact 1 a morphism in $GraphDiagram_S$ is a natural transformation. A natural transformation is a family of morphisms, containing one morphism for each node of the scheme. A schematic view of a graph diagram morphism for the running example is depicted in Figure 6.12. The morphism maps from a graph diagram D' to a graph diagram D. Both have the scheme of the running example. The natural transformation contains three morphisms. They need to fulfil two properties:

- *Compatibility with the scheme:* The natural transformation contains morphisms that map between attributed graphs that correspond to the same object in the scheme. In the example in Figure 6.12 the morphism can map UI' in D' to UI in D because they both are mapped from the node UI in S.
- *Compatibility with morphisms in the diagram:* The morphisms in the natural transformation need to commute with the morphisms in the diagram that represent the same morphism in the scheme. In Figure 6.12 the upper and lower rectangles of attributed graph morphisms (1) and (2) need to commute.

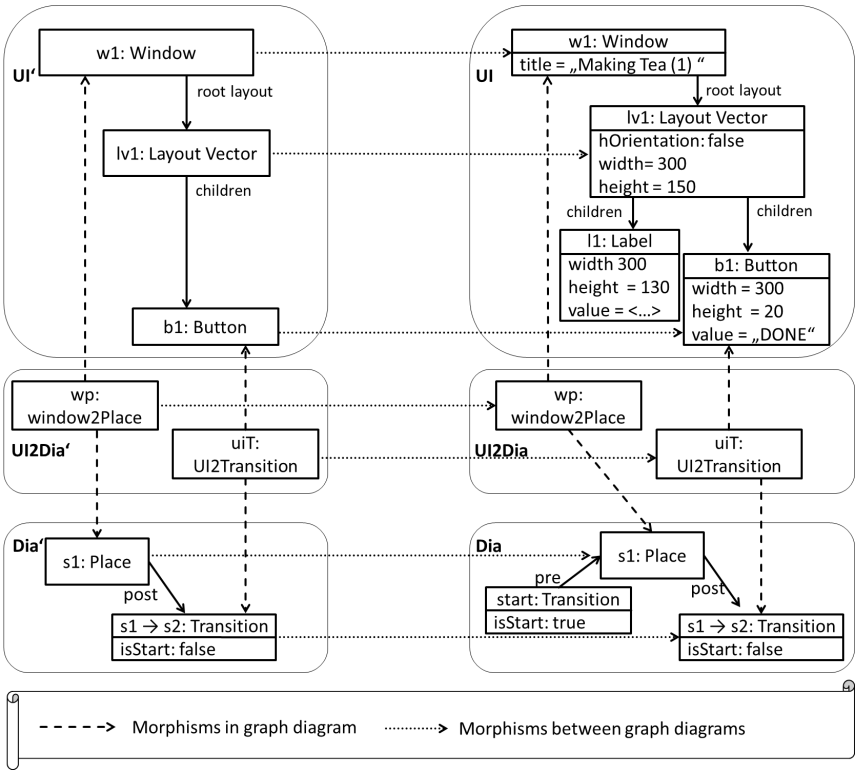


Figure 6.13.: Example: A graph diagram morphism.

An example for a graph diagram morphism is depicted in Figure 6.13. This morphism maps a graph diagram D' to the graph diagram D from Figure 6.11. D' contains a window with a layout vector which contains a button in the UI model and a place connected to one transition in the dialog model. Both the window and place and the button and transition are related via an element in $UI2Dia'$. The figure illustrates the morphisms in the graph diagram and between graph diagrams by different kinds of arrows. However, the reader should be aware that both are attributed graph morphisms. The morphisms between graph diagrams map nodes in the source diagram to nodes in the target diagram. These mappings have to be compatible with the morphisms in the diagram. E.g., the mapping of uiT in $UI2Dia'$ to uiT in $UI2Dia$ has to be compatible

with its mapping to the button and the transition it relates. In both diagrams the respective node uiT is mapped to the nodes $b1$ and $s1 \rightarrow s2$. Since these are also mapped by the morphism the morphism is well-defined.

The typing concept for graph diagrams is similar to the typing concept in attributed typed graphs. The typing is based on a dedicated type diagram and a morphism that defines the typing. The category of typed graph diagrams is defined as follows:

Definition 8 (Category $TypedGraphDiagrams_{S,TG}$)**.** *Given a small category S, an object $TG = (O_{TG}, M_{TG})$ in $\mathbf{GraphDiagrams}_S$ is a **type graph diagram** if for all objects o from S the object $O_{TG}(o)$ is an attributed type graph as defined in Definition 37.*

Given a type graph diagram TG from $\mathbf{GraphDiagrams}_S$, the category of typed graph diagrams over TG is the slice category $(GraphDiagrams_S \downarrow TG)$. The category is denoted by $\mathbf{TypedGraphDiagrams}_{TG}$.

The typing mechanism is based on the categorial construction of slice category. The objects in the slice category of a category with respect to an object TG consist of all morphisms $A \rightarrow TG$. The morphisms define the typing of the elements in A with respect to TG. The following fact describes typed graph diagrams and morphisms.

Fact 2 (Typed Graph Diagrams and Morphisms)**.** *A **typed graph diagram** $(D, type)$ over a type graph diagram TG consists of a graph diagram D and a graph diagram morphism $type : D \rightarrow TG$.*

*A **typed graph diagram morphism** $m : TD_1 \rightarrow TD_2$ between two typed graph diagrams $TD_1 = (D_1, type_1)$ and $TD_2 = (D_2, type_2)$ is a graph diagram morphism between D_1 and D_2 with $type_1 = type_2 \circ m$.*

*A typed graph diagram $(D, type)$ is **empty** if D is empty.*

A type graph diagram defines node, edge and morphism types. Node and edge types are defined by the graphs in this diagram. This typing is analogue to the typing in attributed typed graphs. Each graph in the type graph diagram defines the types for one model exclusively. The morphisms in the type graph diagram define morphism types. They describe which node and edge types can be mapped to each other via morphisms.

A typed graph diagram is described in Fact 2. It is typed via a morphism into the type graph diagram. This morphism assigns types to each node, edge and attribute for each model in the typed graph diagram. Morphisms between typed graph diagrams are graph diagram morphisms that are compatible with the typing. The notions of empty diagrams can be lifted from graph diagrams to typed graph diagrams.

The type graph diagram for the running example is depicted on the right hand side of Figure 6.14. For the UI and dialog model the type graph diagram contains the respective attributed type graphs from Figures 6.7 and 6.9. The type graph for the relation $UI2Dia$ contains two node types. The node type $Window2Place$ relates a place to the window that is shown when it is marked. The node type $UI2Transition$ relates a UI element to the transition it fires. In the type graph they are mapped to the respective types they connect. This assures that an element of the type $Window2Place$ can only be mapped to a window and a place and an element of the type $UI2Transition$ can only be mapped to a UI element and a transition.

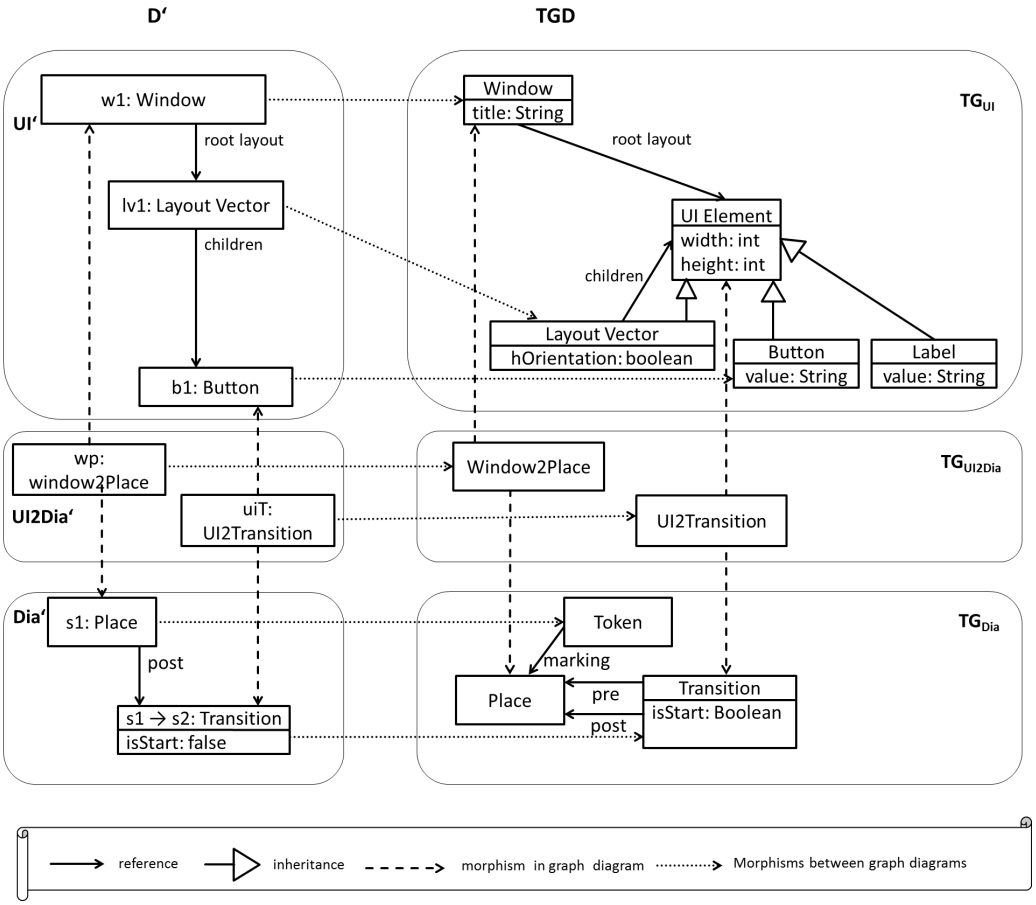


Figure 6.14.: Example: A typed graph diagram.

The examples in this and the previous section use inheritance edges in the type graph although the category of typed graph diagrams is defined based on attributed typed graphs without inheritance edges. As described in Section 2.2.5 attributed typed graphs with inheritance edges can be flattened to plain attributed typed graphs. Accordingly, graph diagrams that contain attributed typed graphs with inheritance edges as nodes can also be flattened to typed graph diagrams without inheritance edges.

The left hand side of Figure 6.14 contains the graph D' from the examples and its typing morphism $D' \to TGD$. To save space in future figures the typing morphism is annotated in the node names. For example the node $w1 : Window$ has the name $w1$ and the type $Window$. The same is done for edges. However, in most cases explicit edge names are omitted and only the type is used. The previous examples in this section already contain this annotation.

The morphisms in $TypedGraphDiagrams_{TG}$ are graph diagram morphisms. In addition to the consistency to the scheme and morphisms in the diagram they fulfil one additional property:

- *compatibility to the typing morphisms:* The morphisms between typed graph diagrams need to preserve the typing. The morphism needs to commute with the typing morphisms of its source and target.

The example for a graph diagram morphism in Figure 6.13 already has types annotated. Thus, it is also an example for a typed graph diagram morphism. It preserves the types of its nodes, edges and relations.

The category of typed graph diagrams forms the basis for the model formalism. The category is $\mathcal{M}$-adhesive as the functor and slice category constructions preserve the $\mathcal{M}$-adhesive properties. The class $\mathcal{M}$ consists of natural transformations that contain injective attributed typed graph morphisms with isomorphic data components for each node in the scheme. However, similar to attributed typed graphs, typed graph diagrams are not finitary and do not contain an $\mathcal{M}$-initial object. As described in Section 2.2.4 these two properties are required to apply existing theoretical results.

In Section 6.4.1 the category $\mathbf{AGraphs}^{D}_{ATG}$ has been defined as a restriction of attributed graphs to enable these two properties for attributed typed graphs. We can restrict Graph Diagrams in a similar manner by restricting the attributed typed graphs in each diagram to these algebra-restricted attributed typed graphs. In addition, the scheme is restricted to have a finite amount of objects and morphisms. This is defined in the following Definition:

Definition 9 (Algebra Restricted Graph Diagrams and Typed Graph Diagrams)**.** *Given a small category S with a finite number of objects and morphisms and an algebra D, the category $\boldsymbol{GraphDiagrams}^{D}_{S}$ is a subcategory of the category $\boldsymbol{GraphDiagrams}_{S}$ whose objects are diagrams of objects and morphisms from $\boldsymbol{AGraphs}^{D}$ and whose morphisms are natural transformations over morphisms from $\boldsymbol{AGraphs}^{D}$.*

Given an algebra D and a small category S with finite number of objects and morphisms a D-type graph diagram is a type graph diagram over S in which all objects are D-type graphs.

Given an algebra D, a small category S with finite number of objects and morphisms, and a D-type graph diagram TG over S, the category $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$ is a subcategory of $\boldsymbol{TypedGraphDiagrams}_{TG}$ with objects $(D, type)$, where D is an object from $\boldsymbol{GraphDiagrams}^{D}_{S}$, and morphisms from $\boldsymbol{GraphDiagrams}^{D}_{S}$.

These restrictions lead to a subcategory of graph diagrams whose models fulfil the conditions described in Section 6.4.1. All models are restricted to the same algebra, meaning the same underlying data type is assumed. We can choose an algebra that contains all data sorts that are needed for the contained models. Thus, the restriction of the algebra is not a restriction in expressiveness. In addition, the scheme is restricted to a finite number of objects and morphisms. Accordingly, the category is restricted to frameworks that contain a finite number of models and relations. Since computing systems are restricted in the amount of memory they have available anyway we do not consider this restriction to be critical.

The resulting category is a finitary $\mathcal{M}$-adhesive category with $\mathcal{M}$-initial object. This is stated in the following theorem:

Theorem 1. *Given*

- *D an algebra*
- *S a small category with finite sets of objects and morphisms.*
- *TG a D-type graph diagram over S*

$(\boldsymbol{TypedGraphDiagrams}^{D}_{TG}, \mathcal{M})$ *is a finitary* $\mathcal{M}$*-adhesive category that contains an* $\mathcal{M}$*-initial object.* $\mathcal{M}$ *is the class of all monomorphisms in* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$.

Proof for this theorem can be found in Appendix B.1.
In the next section we describe how graph diagrams can be used as model formalism.

Formalism for Composite Models

In this section the model formalism for composite models is defined based on graph diagrams. The composite model and composite meta model extend the elementary model and meta model to represent multiple models and their relation.

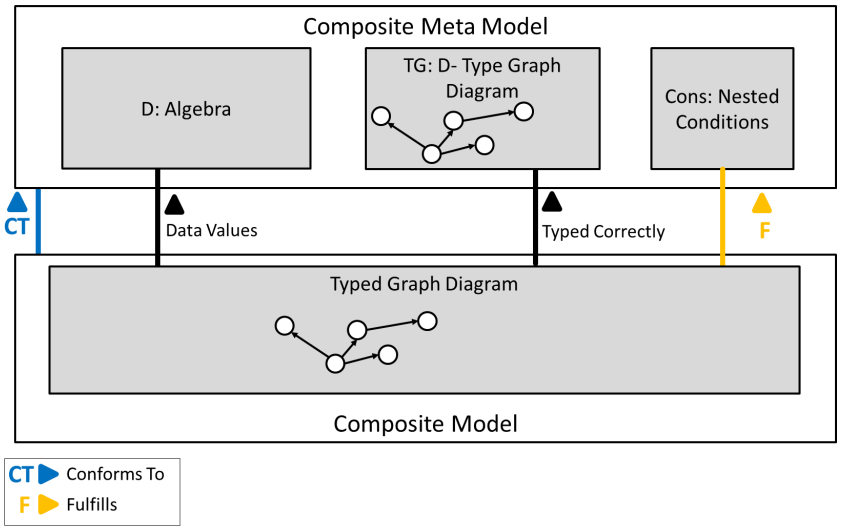


Figure 6.15.: The model formalism for composite models.

Figure 6.15 gives an overview about the definition of the composite model and composite meta-model using typed graph diagrams. The composite model is a typed graph diagram. Definition 10 defines the composite model.

Definition 10 (Composite Model)**.** *Given an algebra D and a D-type graph diagram TG, a* ***composite model*** *is an object from* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$

The composite meta model (see Definition 11) is defined analogous to the elementary meta model. It consists of an algebra, a type graph diagram and a set of nested

conditions. The algebra contains the data that can be used as attribute values in the models. The type graph diagram TG contains the type graphs of all elementary models that are present at run time. The morphisms in the type graph diagram define types for relations between models. The nested conditions in the composite meta model are formulated over graph diagrams. Since graph diagrams are an $\mathcal{M}$-adhesive category this construct is possible. The definition of the composite meta model is as follows:

Definition 11 (Composite Meta Model)**.** *A* ***composite meta model*** *is a tuple* $cmm = (D, TG, cons)$ *containing:*

- D *an algebra*
- TG *a* D*-type graph diagram*
- *cons a set of nested conditions with objects from* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$

A composite model is conform to this composite meta-model if it is an object in $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$ *and fulfils all conditions in cons.*

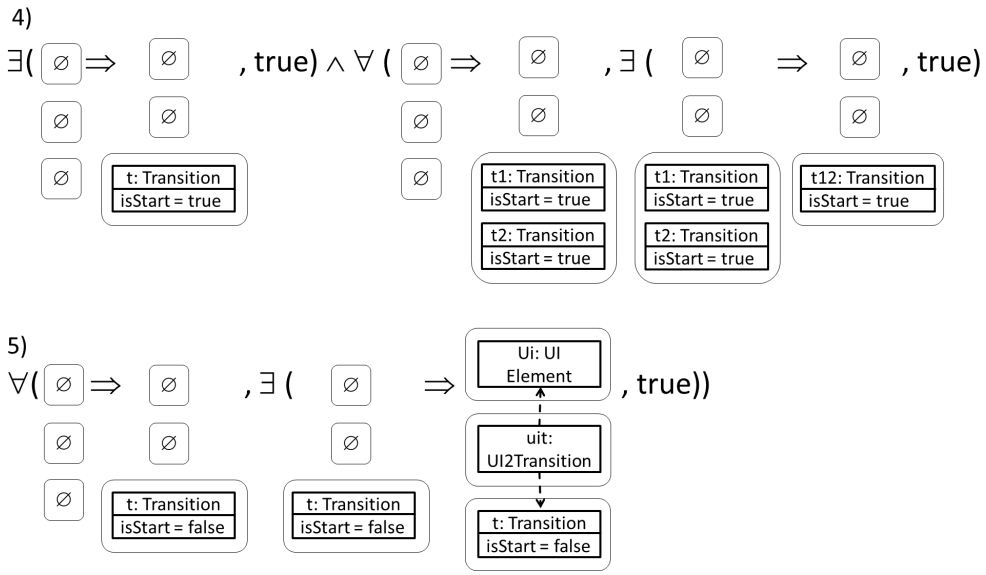


Figure 6.16.: Example: Two nested conditions on graph diagrams.

Part of the example for the composite meta model has already been given. Section 6.4.1 discusses the typing concept for graph diagrams and gives the type graph for the running example. This type graph can be found in Figure 6.14. Figure 6.16 shows conditions four and five from Section 6.3.2.

Condition four requires the existence of one and only one start transition. This condition is formulated in the dialog model. For this reason all other models in the diagrams contained in this condition are empty. Condition four is a conjunction of two parts.

The first part is an existence quantifier that requires the existence of a start transition. The second part states that every two start transitions that can be found are in fact the same transition. The second part is stated in two nesting levels. The outer all-quantified condition finds all places where two start transitions can be matched (potentially to the same transition). The inner condition states that all of these occurrences need to be matched to the same transition. This condition states that there can never be more than one start transition.

Condition five represents a condition on both models. It states that every transition that is not a start transition is connected to a UI element that triggers it. The nested condition states that for every transition with $isStart = false$ (outer all-quantified constraint) there exists a UI element that is mapped to the transition via a node typed UI2Transition (inner existence constraint). This constraint spans multiple models. It utilizes the mapping of transitions to UI elements specified in the type graph in its inner existence constraints. A condition may also span more than two models and relations.

The use of graph diagrams for composite models is compatible with the view that single models are represented as attributed typed graphs. In fact, the definitions of elementary model and elementary meta model are contained in the definition of composite models and meta models. An attributed typed graph can be interpreted as a special case of a typed graph diagram that only contains one element and no (non-identical) morphisms.

It is not always practical to use the morphisms in the diagram directly to represent model relations. Morphisms in attributed graphs are left-total. They map every element of the source model to an element of the target model. However, in most cases it is not necessary to relate all elements of a model to another model. To enable a partial relation an additional model can be used. This model represents the relation and is mapped into the related models. This model can also be related to more than two models at a time, which is another limitation of graph diagram morphisms. This explicit handling of relations is also used in triple graphs (see for example [112]) to represent the relation between two models. The running example is also structured this way to represent the relation between the UI and dialog model.

Since our formalism is able to represent arbitrary diagram structures we do not enforce this representation of relations explicitly. The theory is applicable whether or not relations are represented as dedicated models.

In the next section we describe how additional structural conditions can be expressed using the condition formalism.

6.4.2. Condition Formalism

In this section we define the condition formalism. Multiple models can coexist at run time and also need to be restricted jointly and with respect to their relation with each other. Accordingly, the following two types of consistency requirements should be expressible using the condition formalism:

- *single-model conditions :* it should be possible to express conditions that are specific to a single model.

- *multi-model conditions :* it should be possible to express conditions that restrict multiple models and their relation to each other.

The difference between the conditions from the condition formalism and the conditions in the composite meta-model is that the conditions in the composite meta-model are considered part of the meta-model and thus have to be fulfilled for the model to be considered a valid member of the modelling language and thus interpretable at run time. The conditions expressed in the condition formalism can be used to specify additional consistency requirements. From a formal point of view both types of conditions can be expressed using the same formalism. This also enables the same analysis method to be applied to both types of conditions. Thus, the condition formalism also consists of nested conditions on the basis of typed graph diagrams.

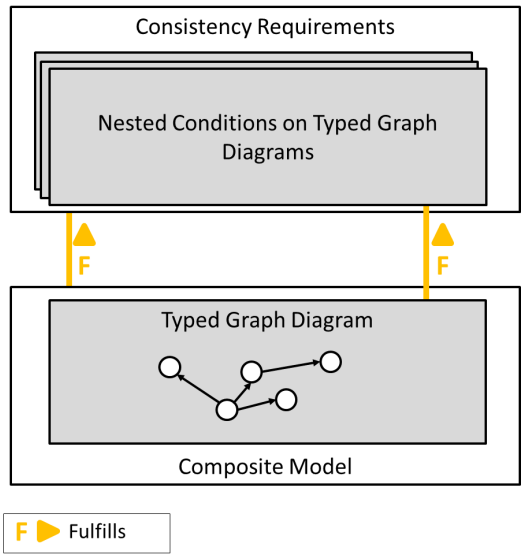


Figure 6.17.: The condition formalism.

The relevant parts of the condition formalism and their formalization are indicated in Figure 6.17. As defined in the previous section a composite model is a typed graph diagram. Accordingly, the consistency requirements use nested conditions over typed graph diagrams. Definition 12 defines consistency requirements.

Definition 12 (Consistency Requirement)**.** *A* ***consistency requirement*** *for a composite meta model* $cmm = (D, TG, cons)$ *is a nested condition over objects from the category* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$

The typed graph diagrams in a consistency requirement can partially consist of empty models. This enables the formulation of specific requirements that only concern one model or a specific subset of models. Thus, we can formulate single-model conditions by using graph diagrams where all other models are empty in the condition. Multi-model

conditions have at least two non-empty models. Relations can be restricted by using morphisms in the condition. Thus, both requirements are fulfilled.

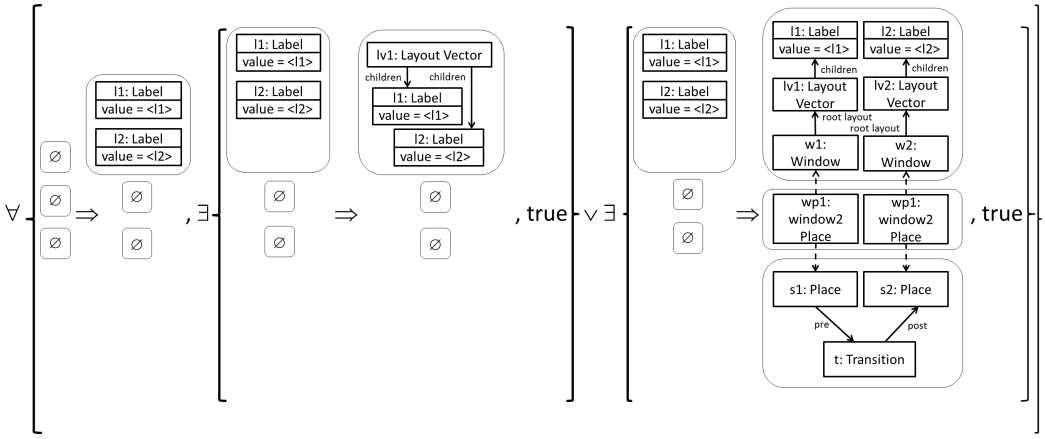


Figure 6.18.: Example: A consistency condition.

The running example contains two examples for developer requirements that are local to the application for tea making. Figure 6.18 depicts a part of the eighth condition formulated as nested condition. This condition requires each step of the tea-making process to be displayed at the same time or immediately after its preceding step. The nested condition consists of two nesting levels. The outer all-quantified condition states that for the two labels with the content of step 1 ($< l1 >$) and step 2 ($< l2 >$) the inner condition needs to hold. The inner condition is a disjunction of two existence conditions. The first condition states that both labels are contained in the same layout vector and thus are shown at the same time. The second condition states that the labels are contained in different windows that are shown in consecutive order. This is expressed by stating that each label is contained in a layout vector, which in turn is contained in a window and requiring that these two windows are related to places in the dialog model that are connected via a transition. The full implementation of condition eight contains the same structure for steps 2 and 3 of the tea making process.

The next section details the adaptation formalism.

6.4.3. Adaptation Formalism

In this section we define the adaptation formalism. The running software system can contain multiple models. It should be possible to adapt each model individually. It should also be possible to adapt multiple models and their relation to each other at the same time. This is formulated in the following requirements:

- *single-model adaptations:* It should be possible to implement adaptations that only concern single models.
- *multi-model adaptations:* It should be possible to implement adaptations that change multiple models at the same time.

- *adaptation of model relations:* It should be possible to target and adapt the relations of models.

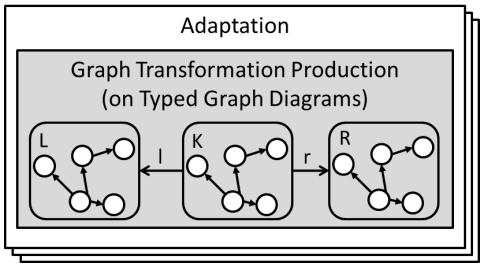


Figure 6.19.: The adaptation formalism.

The extension of the general scheme for an adaptation is shown in Figure 6.19. In the general approach each adaptation contains an action. In Figure 6.19 the action is a graph transformation production. This is defined in Definition 13.

Definition 13 (Adaptation)**.** *An adaptation based on a composite meta model* $cmm = (D, TG, cons)$ *is a tuple* $a = (c, p)$ *where*

- $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *is a graph transformation production in the category* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$

The action of an adaptation is a description of the changes caused by the adaptation. This is defined as a graph transformation production (see Section 2.2.2). The production is formulated based on the category of graph diagrams that are typed over the type diagram defined in the composite meta model. Thus, the production is able to reconfigure the composite models, which are also typed over this type diagram.

A match morphism is required to apply a graph transformation production to a typed graph diagram. Since multiple matches can exist for one graph transformation productions there are potentially multiple alternatives to choose from when the adaptation is applied. For example, the production could be applied for all possible matches or one specific match. The Trollmann approach is independent of the mechanism used for selecting adaptations and matching the graph transformation production. It operates on selected adaptations and their matched production applications. Thus, the matching strategy is left open. It can be chosen and implemented freely by the developer of the software framework.

The selection of adaptation and matches leads to a set of graph transformation productions applications (that are already matched) that are to be applied. This set of production applications is the basis for the analysis of adaptation conflicts described in the next sections.

The graph transformation production in the adaptation rule can target arbitrary state models as long as they are correctly typed. Thus, we can formulate graph transformation productions that only contain graphs diagrams with one non-empty model in L, K and

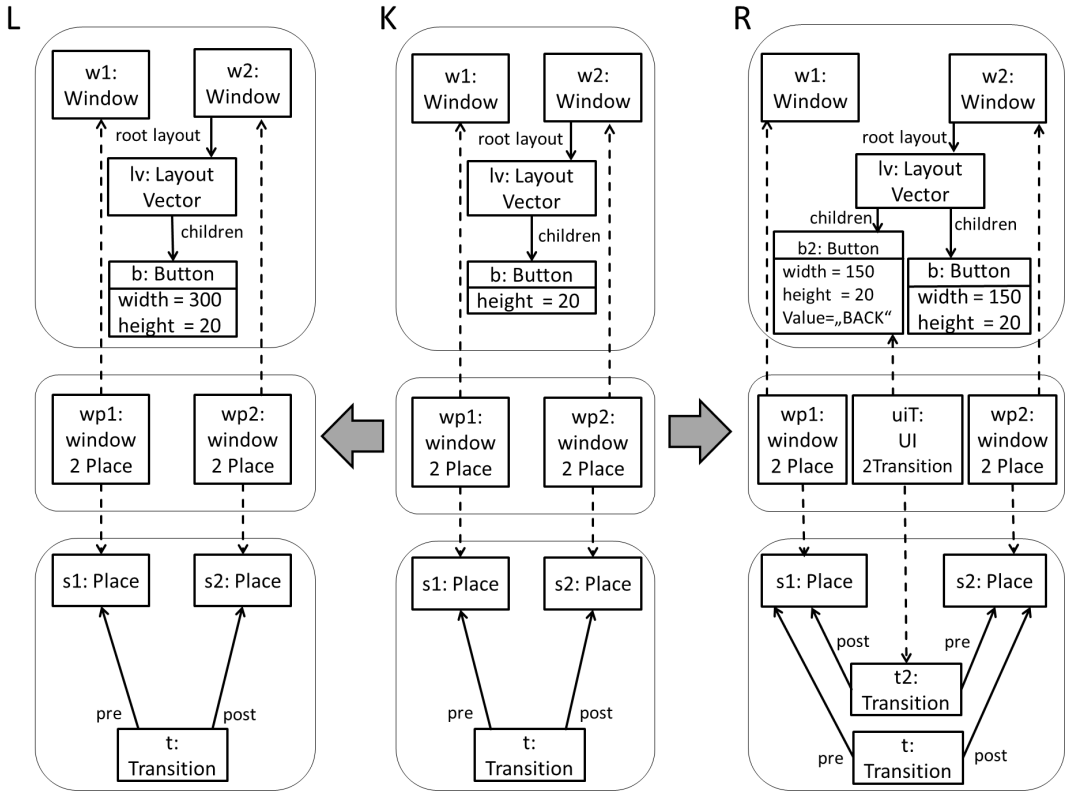


Figure 6.20.: Example: A production for adding a back button.

R. Such productions can be used to target and change single elementary models. This fulfils the requirement for single-model adaptations. Similarly, the requirement for multi-model adaptations can be fulfilled if L, K and R contain more than one non-empty model. The transformation can also change model relations by changing the elements that are mapped by morphisms.

The running example contains two adaptations. The first one adds a back button to one window to enable the user to skip to the previous step. The according graph transformation production is depicted in Figure 6.20. The left hand side, interface and right hand side of the rule are graph diagrams that are typed over the type graph diagram from the running example. The left hand side of the transformation rule requires two windows that are associated to two places that are connected via a transition. The rule makes use of the relation between the UI and dialog model to express this. All morphisms between the three graphs of the production are inclusions, meaning objects are mapped to objects of the same name, e.g., the place $s1$ in the interface is mapped to the place $s1$ in the left and right hand side. In the first step ($K \rightarrow L$) the attribute width of button b is removed. The attribute is added again in the second step ($K \rightarrow R$) with half the value to free space for the new button $b2$, which is also added in this step. In addition, a transition $t2$ is added to the dialog model and connected to the new button

by the element uiT in the relation node. This transition enables the user to switch back to the previous window.

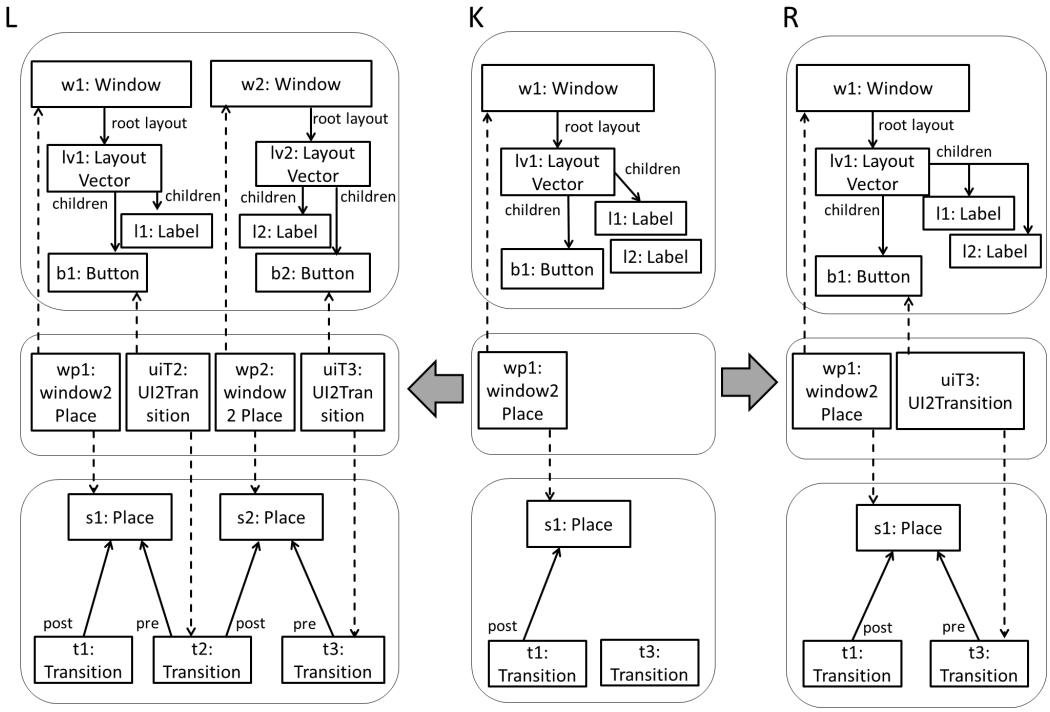


Figure 6.21.: Example: A production for uniting two windows.

The production for the second adaptation in the running example is depicted in Figure 6.21. The purpose of this adaptation is to unite two windows and show both their content in one window. Accordingly, the left hand side of the production contains two windows with each a label and a button that are related to two places. The places are connected with a transition $t2$ and there is a transition $t1$ leading to the first place and a transition $t3$ continuing from the last place. In the first step the window $w2$ is removed with all of its content except for the label $l2$. The place $s2$ is removed along with the window as well as the transition $t2$. In a second step the label $l2$ is added to the layout of the first window and the connection between the place $s1$ and the transition $t2$ is re-established via a pre arc.

In the next section we develop algorithms for the detection of adaptation conflicts based on the formalism introduced in this section.

6.5. Conflict Detection

This section describes the Trollmann approach to adaptation conflict detection. The detection algorithms are based on the formalisms for models, adaptations and consistency conditions defined in Section 6.4.

This section starts out by relating the conflict types from the problem description to situations in the formalism that need to be detected in Section 6.5.1. The results from this section indicate that two kinds of situations need to be detected: Dependencies between graph transformation productions, which are handled in Section 6.5.2, and the non-fulfilment of nested conditions, which is handled in Section 6.5.3. Parallel conflicts are easier to detect than sequential conflicts as the conflicting rules are activated at the same time and supposed to be applied to the same model. For this reason our initial conflict detection is based on the parallel case. In Section 6.5.4 we describe how the algorithms can be extended for the detection of sequential conflicts.

6.5.1. Basis for Conflict Detection

Section 3.1 describes adaptation-adaptation and adaptation-consistency conflicts. These conflicts are formulated on the basis of the software systems and their adaptations. This section relates these general notions of conflicts to our modelling approach and formalism. As a first step we summarise the information available at run time. This information is the basis for conflict detection algorithms. Subsequently, we extract situations that represent adaptation conflicts.

At run time there are two general types of information available for conflict detection. One type is situation-independent information. This information is independent of the current state. The following situation-independent information is available for analysis in our formal framework:

- *D:* The algebra from the composite meta model. It contains data that can be used for attributes in the diagram.
- *TG:* The D-type graph diagram from the composite meta-model. This diagram describes the structure of the models and specifies which types can be used in the models and relations.
- *Cons:* The nested conditions that need to be fulfilled. This set consists of the nested conditions $Cons_{type}$ from the composite meta-model and the consistency requirements $Cons_{consistency}$. All nested conditions are constructed over objects from the category $\mathbf{TypedGraphDiagrams}^{D}_{TG}$.
- *Ads:* All available adaptations. The graph transformation productions of these adaptations are over objects from $\mathbf{TypedGraphDiagrams}^{D}_{TG}$.

The second type of information describes the current state of the software system. The following situation-dependent information is available:

- *G:* A graph diagram from $\mathbf{TypedGraphDiagrams}^{D}_{TG}$ that represents the current state of the models in the software system.
- *Prods:* A set of production applications to G that have been determined from *Ads*. The adaptations and matches have been determined according to the selection and matching strategy of the software system.

The analysis makes use of Assumptions 1 and 2 (cf. Section 6.1) which state that all aspects that need to be analysed are represented as models and are synchronised with the running software system. This enables the detection of adaptation-adaptation and adaptation-consistency conflicts on the basis of these models.

The current state of the models G and the activated production applications $Prods$ are relevant to detect adaptation-adaptation conflicts. This setup is free of adaptation-adaptation conflicts if all production applications in $Prods$ can be applied in any order and all orders lead to the same result. As stated by the Local Church-Rosser Theorem (cf. Section 2.2.4) this is possible if all production applications in $Prods$ are parallel independent. Accordingly, an adaptation-adaptation conflict occurs whenever at least two production applications in $Prods$ are parallel dependent. A detection of parallel dependencies detects all adaptation-adaptation conflicts if all relevant adaptations are reflected in $Prods$ (Assumption 4 holds).

The current situation is free of adaptation-consistency conflicts if the result of the application of $Prods$ to G and all intermediate states are valid models according to the composite meta-model. A valid model is correctly typed over TG and fulfils all conditions in $Cons_{type}$. In addition, the model needs to fulfil all of the consistency requirements contained in $Cons_{consistency}$.

Graph transformation productions preserve typing. Since all graph transformation productions in $Prods$ are formulated over objects from $\mathbf{TypedGraphDiagrams}^{D}_{TG}$ they preserve conformance to the type graph diagram TG. In addition, from the point of view of conflict detection it does not matter whether a nested condition stems from $Cons_{type}$ or from $Cons_{consistency}$. This information is only relevant to conflict resolution in case the two sets are treated differently. Accordingly, the detection of adaptation-consistency conflicts requires detecting whether the end-state of the adaptation and all immediate states fulfil all conditions in $Cons$. As stated in Assumption 3 the consistency conditions $Cons$ are assumed to be complete. If this assumption holds we can find all adaptation-consistency conflicts by analysing these conditions.

Subsuming, the following two types of situations need to be detected:

- **Dependent Production Applications:** Adaptation-adaptation conflicts are detected by finding dependencies between graph transformation production applications in $Prods$.
- **Violated Nested Conditions:** Adaptation-consistency conflicts are detected by finding subsets of $Prods$ that violate nested conditions in $Cons$ when applied.

The detection of of dependencies between production applications is described in Section 6.5.2. We handle the detection of violations of conditions in Section 6.5.3.

6.5.2. Detection of Dependencies between Graph Transformation Productions

This section describes how dependencies between graph transformation productions can be detected and how additional information about the dependencies can be extracted.

The analysis is based on the graph diagram G, representing the current state of the models, and the set of production application $Prods$ that are supposed be applied.

According to the Local Church Rosser Theorem (see Section 2.2.4) parallel and sequential independence are equivalent. This means, if all production applications in $Prods$ are parallel independent they can be applied in arbitrary order yielding the same end-result. In this section we aim to detect parallel dependencies.

To limit the size of the figures we use a simplified version of the running example. The examples are restricted to one of the elementary models, the UI model. Thus, the examples are from the category of attributed typed graphs. However, the reader should keep in mind, that the theory is also applicable to typed graph diagrams since the general theory is formulated on the level of $\mathcal{M}$-adhesive categories. In fact, the example is treated as a typed graph diagram with one node.

According to the definition of parallel independence (cf. Definition 31), two morphisms $a_1 : L_1 \to C_2$ and $a_2 : L_2 \to C_1$ need to exist for two production applications $p_1 = ((L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1), m_1)$ and $p_2 = ((L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2), m_2)$ to be parallel independent. The morphisms map the left hand side of one production application into the result of the first application step of the other production application. They need to commute with the existing morphisms from the application of the production application. The existence of these morphisms denotes that no elements are removed that are needed for matching the other production application.

All production applications in $Prods$ need to be pairwise parallel independent. If this is the case they can be applied in arbitrary order yielding the same result. Accordingly, for each pair of production applications in $Prods$, these two morphisms have to exist.

Figure 6.22 shows the UI component of the two graph transformation productions from the running example applied to the UI model. The figure also shows the morphism a_1. All morphisms in the image are inclusions, meaning the source and targets of the mappings have the same name. The morphism a_1 exists and maps both windows in L_1 to the respective windows in C_2. This morphism commutes with the morphisms m_1 and g_1 and is thus one of the morphisms required for parallel independence. However, the morphism $a_2 : L_2 \to C_1$ does not exist. Window $w2$ in L_2 cannot be mapped to any node in C_1 such that $m_2 = g_1 \circ a_2$. Since this morphism does not exist the production applications from the example are not parallel independent.

The detection of dependencies at run time requires an automated mechanism for deriving whether these morphisms a_1 and a_2 exist. Furthermore, the morphisms are required to generate the new match, after one production application has been applied. Accordingly, an automated mechanism for constructing these morphisms is required.

One mechanism that could be used to detect these conflicts is critical pair analysis, as described in the related work in Section 4.3. A critical pair is an example of a parallel dependent situation. This example can be checked to detect whether this conflict occurs in the current situation. The advantage of critical pair analysis is that the main computational effort lies in the computation of critical pairs, while checking the critical pairs is relatively cheap. However, part of our motivation is the fact that the set of adaptations can be dynamic at run time. For critical pair analysis, this means that the

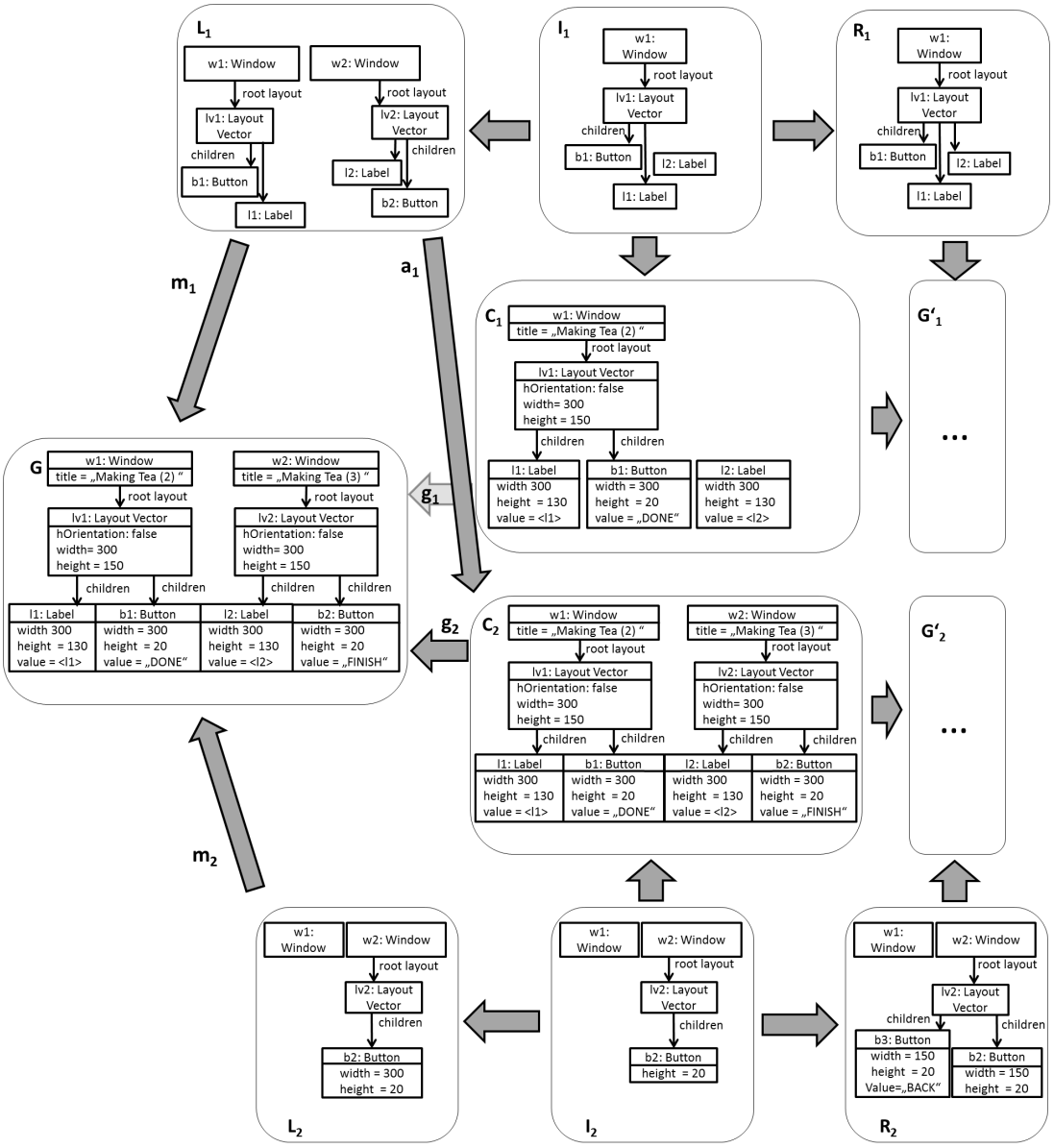


Figure 6.22.: Example: Parallel dependence.

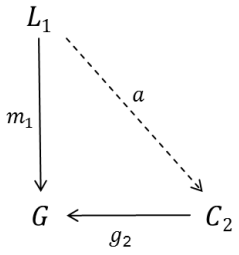
calculation of critical pairs has to be executed at run time whenever the set of adaptations changes. This re-calculation has to take into account all adaptations in *Ads*. Depending on the time available when such a change occurs and the frequency of the changes this can pose a problem. Accordingly, we follow a different path for the detection of parallel conflicts that is based on checking the existence of morphisms a_1 and a_2 directly based on the productions in *Prods*. However, critical pair analysis is also a viable option, if the recalculation can be handled by the running system. The interested reader can refer to [38] for more information on critical pair analysis.

The following two definitions describe the conditions for the existence of a_1 and a_2 and a function to construct these morphisms if they exist. Since the construction of a morphism is specific to the category this morphism is constructed in, these definitions operate on the level of specific categories. The first definition handles the condition and construction based on attributed graphs. These conditions are then extended to typed graph diagrams. The conditions for attributed graphs are the following:

Definition 14 (Construction of a Commuting Morphism in Attributed Graphs)**.** *Given*

- *an algebra* D
- $L_1 = ((V_{G,L_1}, V_D, E_{G,L_1}, E_{NA,L_1}, E_{EA,L_1}, (source_j^{L_1}, target_j^{L_1})_{j\in\{G,NA,EA\}}), D)$ *an attributed graph from* $\boldsymbol{AGraphs}^D$.
- $C_2 = ((V_{G,C_2}, V_D, E_{G,C_2}, E_{NA,C_2}, E_{EA,C_2}, (source_j^{C_2}, target_j^{C_2})_{j\in\{G,NA,EA\}}), D)$ *an attributed graph from* $\boldsymbol{AGraphs}^D$.
- G *an attributed graph from* $\boldsymbol{AGraphs}^D$.
- $m_1 = ((f_{V_G,m_1}, id_{V_D}, f_{E_G,m_1}, f_{E_{NA},m_1}, f_{E_{EA},m_1}), id_D) : L_1 \to G$*: an attributed graph morphism from* $\boldsymbol{AGraphs}^D$.
- $g_2 = ((f_{V_G,g_2}, id_{V_D}, f_{E_G,g_2}, f_{E_{NA},g_2}, f_{E_{EA},g_2}), id_D) : C_2 \to G$*: an attributed graph monomorphism from* $\boldsymbol{AGraphs}^D$.

that are related as illustrated in the following figure:



a morphism $a = ((f_{V_G}, id_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}}), id_D) : L_1 \to C_2$ *with* $m_1 = g_2 \circ a$ *exists if the following conditions hold:*

1. $f^{-1}_{V_G,g_2} \circ f_{V_G,m_1}$ *is fully defined, meaning for all nodes* v_1 *in* V_{G,L_1} *there is a node* v_2 *in* V_{G,C_2} *such that* $f_{V_G,g_2}(v_2) = f_{V_G,m_1}(v_1)$
2. $f^{-1}_{E_G,g_2} \circ f_{E_G,m_1}$ *is fully defined, meaning for all edges* v_1 *in* E_{G,L_1} *there is an edge* v_2 *in* E_{G,C_2} *such that* $f_{E_G,g_2}(v_2) = f_{E_G,m_1}(v_1)$
3. $f^{-1}_{E_{NA},g_2} \circ f_{E_{NA},m_1}$ *is fully defined, meaning for all node attribute edges* v_1 *in* E_{NA,L_1} *there is a node attribute edges* v_2 *in* E_{NA,C_2} *such that* $f_{E_{NA},g_2}(v_2) = f_{E_{NA},m_1}(v_1)$
4. $f^{-1}_{E_{EA},g_2} \circ f_{E_{EA},m_1}$ *is fully defined, meaning for all edge attribute edges* v_1 *in* E_{EA,L_1} *there is an edge attribute edge* v_2 *in* E_{EA,C_2} *such that* $f_{E_{EA},g_2}(v_2) = f_{E_{EA},m_1}(v_1)$

it can be constructed component-wise as follows:

- $f_{V_G} = f^{-1}_{V_G,g_2} \circ f_{V_G,m_1}$.
- $f_{E_G} = f^{-1}_{E_G,g_2} \circ f_{E_G,m_1}$.
- $f_{E_{NA}} = f^{-1}_{E_{NA},g_2} \circ f_{E_{NA},m_1}$.
- $f_{E_{EA}} = f^{-1}_{E_{EA},g_2} \circ f_{E_{EA},m_1}$.

These conditions are correct and complete. The according morphisms exist if and only if these conditions are fulfilled and the construction of a morphism is possible whenever the conditions are fulfilled. This is stated in the following theorem:

Theorem 2. *Given*

- *an algebra* D
- L_1, C_2 *and* G *three attributed graphs from* $\boldsymbol{AGraphs}^D$.
- $m_1 : L_1 \to G$*: an attributed graph morphism from* $\boldsymbol{AGraphs}^D$.
- $g_2 : C_2 \to G$*: an attributed graph monomorphism from* $\boldsymbol{AGraphs}^D$.

A morphism $a : L_1 \to C_2$ *with* $m_1 = g_2 \circ a$ *exists and can be constructed as described in Definition 14 if and only if all conditions in this definition are fulfilled.*

The theorem is proven in Appendix B.2. The conditions and construction for attributed graphs are used as a basis for the definition on the level of graph diagrams. The according definition is as follows:

Definition 15 (Construction of a commuting Morphism in Graph Diagrams)**.** *Given*

- D *an algebra*
- TG *a* D*-type graph diagram with scheme* $S = (O, M)$
- L_1, C_2 *and* G*, three typed graph diagrams from* $\boldsymbol{TypedGraphDiagrams}^D_{TG}$

- $m_1 : L_1 \to G$ *a graph diagram morphism from* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$
- $g_2 : C_2 \to G$ *a graph diagram monomorphism from* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$

a morphism $a : L_1 \to C_2$ *with* $m_1 = g_2 \circ a$ *exists iff the following condition holds:*

- *for all* $o \in O$*: a morphism* a_o *with* $m_1(o) = g_2(o) \circ a_o$ *exists in the category AGraphs according to Definition 14.*

if this morphism exists it can be constructed component-wise as follows:

- *For all* $o \in O$*,* $a(o)$ *is constructed from* $m_1(o)$ *and* $g_2(o)$ *as described in Definition 14.*

The conditions for graph diagrams are also correct and complete and the construction of the morphism is always possible if the conditions are fulfilled. This is stated in the following theorem:

Theorem 3. *Given*

- D *an algebra*
- TG *a* D*-type graph diagram*
- L_1, C_2 *and* G *three typed graph diagrams from* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$
- $m_1 : L_1 \to G$*: a morphism from* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$
- $g_2 : C_2 \to G$*: a monomorphism from* $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$

A morphism $a : L_1 \to C_2$ *with* $m_1 = g_2 \circ a$ *exists if and only if the condition in Definition 15 are fulfilled. If this morphism exists it can be constructed as in this definition.*

Proof for this theorem can be found in Appendix B.3.

The construction of the morphism in graph diagrams relies on the component-wise construction of morphisms for each diagram node. The morphism for each diagram node is constructed by mapping the elements in L_1 to the elements in C_2 that are mapped to the same elements in G by m_1 and g_2. The existence of these common elements is also the condition for the existence of the morphism in attributed graphs. An element in G that is mapped by L_1 but not mapped from C_2 is deleted during the application of one production although it is required to apply the second production. The condition in graph diagrams requires this condition to be satisfied for all attributed typed graphs in the diagram.

As already stated, a commuting morphism a_2 between L_2 and C_1 does not exist in the running example. This can also be derived by the conditions. Due to the reduction of the running example to a diagram with one node and no (non-identical) morphisms it is sufficient to check the condition on attributed graphs for the UI model according to Definition 15. The non-existence of a_2 can be detected based on the node w_2 in L_2. This node is mapped to w_2 in G by the morphism m_2. However, no node in C_1 is mapped to

this node by g_1. Thus, the condition for nodes in attributed graph is not fulfilled. The same can be detected on the nodes lv_2 and b_2, the edges between w_2 and lv_2 between lv_2 and b_2 and the attributes of b_2. For a_1 the subconditions hold for all nodes and edges of L_1 and thus the morphism exists.

Although the existence of the two morphisms for each combination of two production applications from $Prods$ is required for parallel independence, the actual morphisms are not required for conflict detection. Situations in which such a morphism does not exist represent a dependency for which more information needs to be collected. The conditions for the existence of a morphism are used for this purpose. They point out reasons for the conflict. On the level of graph diagrams the condition has to hold for each node in the scheme. Thus, if one of these sub-conditions fails there is a problem in this specific node and we can point out the model that contains the conflict. The sub-condition in this node relies on the conditions for attributed graphs. This condition consists of four sub-conditions. These conditions represent the following types of conflicts:

- **node conflict** If condition 1 is not satisfied there is a node conflict in the component V_G, meaning the conflict is related to a node of the graph structure.
- **edge conflict** if condition 2 is not satisfied there is an edge conflict in the component E_G, meaning the conflict is related to an edge of the graph structure.
- **node attribute conflict** if condition 3 is not satisfied there is a node attribute conflict in the component E_{NA}, meaning the conflict is related to a node attribute.
- **edge attribute conflict** if condition 4 is not satisfied there is an edge attribute conflict in the component E_{EA}, meaning the conflict is related to an edge attribute.

The elements for which these conditions are not fulfilled are the elements that are in conflict. Thus, the conditions enable the analysis to detect the scope of the conflict by detecting the problematic models, their components and the specific elements that the conflict is based on. The conflict is caused by the fact that these elements are removed or changed by the production application that is first applied while they are required by the one that is applied second. Accordingly, the deletion of these elements in the first production application is the elementary adaptation that caused the conflict together with the fact that the second production application requires them. This is used in Section 6.6 to answer the questions to provide knowledge about conflict resolution.

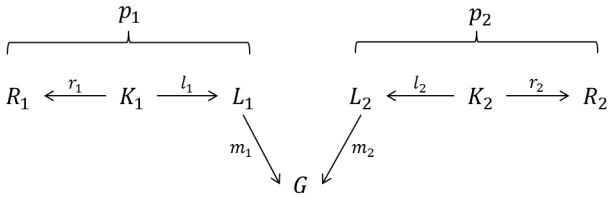
The conditions described above have two weak points. The first is, that they require the objects C_1 and C_2 to detect conflicts. Accordingly, all production applications need to be partially applied to find potential conflicts. We use a different set of conditions to remedy this problem. These conditions are equivalent to the conditions already described but are solely based on the current model D and the production applications. The second weak point is an inefficiency that could be optimised. If the parallel dependency analysis is executed separately for each pair of production applications several operations are calculated multiple times. Our second optimisation uses dynamic programming to calculate overlapping operations only once. The remainder of this section describes these two optimisations.

The first optimisation regards the necessity to partially apply the productions. To avoid this calculation we introduce an alternate condition that makes use of the fact that the production application already contains information about deleted and required elements. This enables us to extract sets of elements that are deleted and required during the application of a production application and compare them. This approach is not possible in general category theory since it makes use of information from specific categories. A set-based condition for attributed graphs is defined in Definition 16. This condition is extended to typed graph diagrams in Definition 17.

Definition 16 (Set-Based Condition in Attributed Graphs)**.** *Given:*

- *an algebra* D
- *attributed graphs* $L_1, K_1, R_1, L_2, K_2, R_2$ *and* G *with* $x = ((V_{G,x}, V_D, E_{G,x}, E_{NA,x}, E_{EA,x}, (source_j^x, target_x^x)_{j\in\{G,NA,EA\}}), D)$ *for* $x \in \{L_1, K_1, R_1, L_2K_2, R_2, G\}$ *from* $\boldsymbol{AGraphs}^D$
- *morphisms* l_1, r_1, m_1, l_2, r_2 *and* m_2 *with* $x = ((f_{V_G,x}, id_{V_D}, f_{E_G,x}, f_{E_{NA},x}, f_{E_{EA},x}), id_D)$ *for* $x \in \{l_1, r_1, m_1, l_2, r_2, m_2\}$ *from* $\boldsymbol{AGraphs}^D$
- $p_1 = ((L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1), m_1)$ *a graph transformation production application to* G
- $p_2 = ((L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2), m_2)$ *a graph transformation production application to* G

as illustrated in the following image:



The following set-based conditions need to hold:

1. $f_{V_G,m_2}(V_{G,L_2}) \cap (f_{V_G,m_1}(V_{G,L_1}) \backslash f_{V_G,m_1}(f_{V_G,l_1}(V_{G,K_1}))) = \emptyset$
2. $f_{E_G,m_2}(E_{G,L_2}) \cap (f_{E_G,m_1}(E_{G,L_1}) \backslash f_{E_G,m_1}(f_{E_G,l_1}(E_{G,K_1}))) = \emptyset$
3. $f_{E_{NA},m_2}(E_{NA,L_2}) \cap (f_{E_{NA},m_1}(E_{NA,L_1}) \backslash f_{E_{NA},m_1}(f_{E_{NA},l_1}(E_{NA,K_1}))) = \emptyset$
4. $f_{E_{EA},m_2}(E_{EA,L_2}) \cap (f_{E_{EA},m_1}(E_{EA,L_1}) \backslash f_{E_{EA},m_1}(f_{E_{EA},l_1}(E_{EA,K_1}))) = \emptyset$

The set-based condition for attributed graphs also consists of four sub-conditions, one for each set V_G, E_G, E_{NA} and E_{EA}. The condition on each set ensures the corresponding condition in Definition 14. It checks whether the required and deleted elements of both production applications are disjunctive by analysing the morphisms l_1, m_1 and m_2. The deleted elements are the elements that are mapped by m_1 but not mapped by $m_1 \circ l_1$. The required elements are the elements mapped by m_2. These conditions are equivalent to the conditions in Definition 14. The following theorem states that they are also correct:

Theorem 4. *Given:*

- *an algebra D*
- *an attributed graph G from* $\boldsymbol{AGraphs}^D$
- *two graph transformation productions* $((L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1), m_1)$ *and* $((L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2), m_2)$ *to G in* $\boldsymbol{AGraphs}^D$

with application as illustrated in Figure 6.23.

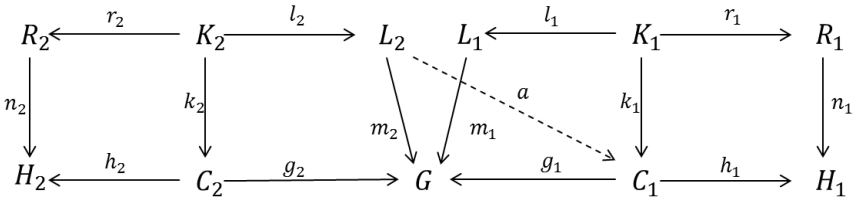


Figure 6.23.: The morphisms required for parallel independence.

The morphism a with $m_2 = g_1 \circ a$ *exists if and only if all conditions in Definition 16 are fulfilled.*

Proof for this theorem is given in Appendix B.4.

Again, we can derive the elements that are the cause of the conflict from the conditions. Each condition requires a specific set of elements to be empty. If this set is not empty the condition is not fulfilled. This set contains the elements that are required to execute the second production but are not preserved by the first production. Accordingly, it contains the elements that caused the conflict.

The set-based condition in graph diagrams is based on this condition in attributed graphs. It is defined as follows:

Definition 17 (Set-Based Condition in Graph Diagrams)**.** *Given:*

- *An algebra D*
- *a D-type graph diagram TG over scheme* $S = (O, M)$
- *typed graph diagrams* $L_1, K_1, R_1, L_2, K_2, R_2$ *and G with* $x = ((O_x, M_x), type_x)$ *with* $x \in \{L_1, K_1, R_1, L_2, K_2, R_2, G\}$ *from* $\boldsymbol{TypedGraphDiagrams}^D_{TG}$
- *typed graph diagram morphisms* l_1, r_1, m_1, l_2, r_2 *and* m_2 *in the category* $\boldsymbol{TypedGraphDiagrams}^D_{TG}$
- $p_1 = ((L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1), m_1)$ *a graph transformation production application to G*
- $p_2 = ((L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2), m_2)$ *a graph transformation production application to G*

The following condition needs to hold:

1. *For all $o \in O$ the condition in Definition 16 needs to hold for the two production applications $p_{1,o} = ((O_{L_1}(o) \overset{l_1(o)}{\leftarrow} O_{K_1}(o) \overset{r_1(o)}{\rightarrow} O_{R_1}(o)), m_1(o))$ and $p_{2,o} = ((O_{L_2}(o) \overset{l_2(o)}{\leftarrow} O_{K_2}(o) \overset{r_2(o)}{\rightarrow} O_{R_2}(o)),\ m_2(o))$*

The definition in graph-diagrams requires the condition for attributed graphs to hold for each of the nodes in the scheme. For each of these nodes two production applications in attributed graphs can be constructed from the respective attributed graphs in the production application in graph diagrams. These production applications can be checked for conflicts in attributed graph using Definition 16. We can provide knowledge about the models that cause the conflict by reporting which of the sub-conditions fails. The following theorem states that the condition is correct:

Theorem 5. *Given:*

- *An algebra D*
- *a type graph diagram TG*
- *two graph transformation productions $p1 = ((L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1), m_1)$ and $p2 = ((L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2), m_2)$ to G in $\boldsymbol{TypedGraphDiagrams}^{D}_{TG}$*

The morphism a exists if and only if $p1$ and $p2$ fulfil the condition in Definition 17.

The theorem is proven in Appendix B.5.

We now apply these conditions to the running example. Again, the diagram with one element can be checked by testing the condition on attributed graphs on the UI model. The following equation system shows that it is not possible to apply the production for adding the back button after the production for merging the windows because of the additional nodes in V_G:

$f_{V_G,m_2}(V_{G,L_2}) \cap f_{V_G,m_1}(V_{G,L_1}) \backslash f_{V_G,m_1}(f_{V_G,l_1}(V_{G,K_1}))$
$= f_{V_G,m_2}(\{w1, w2, lv2, b2\}) \cap f_{V_G,m_1}(\{w1, lv1, l1, b1, w2, lv2, l2, b2\})$
$\backslash f_{V_G,m_1}(\{w1, lv1, l1, b1, l2\}))$
$= \{w1, w2, lv2, b2\} \cap \{w2, lv2, b2\}$
$= \{w2, lv2, b2\}$
$\neq \emptyset$

This condition gives the same result as the previous one. It shows that the nodes $w2$, $lv2$ and $b2$ are removed by one production but required by the other one. The corresponding checks for edges and node attribute also fail and show that conflicts based on the edges and node attributes exist.

These conditions enable the detection of dependencies between two production applications at a time. This check requires testing the conditions for any pair of two production applications. If there are n production applications this requires $n * (n-1)$ checks for different combinations. In each check a set of required and a set of removed elements for each of the two production applications is calculated. Since these two sets

```
01 List<ConflictDescription> CalculateConflictsVG(List<ProductionApplication> apps) {
02     List<ConflictDescription> toReturn;
03
04     Map<VG,List<ProductionApplication>> deletedVG;
05
06     // fill deletedVG from deleted elements
07     for(app: apps) {
08         List<VG> deletedElems = app.getDeletedVG();
09         for(deleted: deletedElems) {
10             deletedVG.get(deleted).insert(app)
11         }
12     }
13
14     // check each production for conflicts and add them to toReturn
15     for(app:apps) {
16         List<VG> requiredElems = app.getRequiredVG();
17         for(required:requiredElems) {
18             List<ProductApplication> conflictingApplications = deletedVG.get(required);
19             for(conflicting: conflictingApplications) {
20                 if(conflicting == app) continue;
21                 toReturn.add(new ConflictDescription(app, conflicting,required);
22             }
23         }
24
25     }
26
27     return toReturn;
28 }
```

Figure 6.24.: The algorithm for detecting conflicts for the component V_G.

do not depend on the respective other production application their repeated calculation can be avoided by using dynamic programming. A pseudo code Version of the optimized algorithm for the component V_G in an attributed graph is given in Figure 6.24. Given a set of production applications in attributed graphs, this algorithm calculates all conflicts between them. It calculates a set of conflict descriptions, which consist of two conflicting production applications and the element in V_G that causes the respective conflict.

The algorithm consists of two phases. In the first phase (Lines 4 - 12) the map $deletedVG$ is filled with a mapping from each node to the set of all production applications that delete it. The deleted nodes are retrieved by using the method $getDeletedVG$, which returns the set $f_{V_G,m_1}(V_{G,L_1}) \backslash f_{V_G,m_1}(f_{V_G,l_1}(V_{G,K_1}))$ for a production application. During the second phase (Lines 14 - 25) each production application is visited again and for each of its required elements in VG and for all productions deleting it (derived from the previously filled map $deletedVG$) a new conflict is generated in Line 21. A production application cannot be in conflict with itself. If a production is applied twice this is reflected in two separate production applications. For this reason Line 20 excludes the current production application from the analysis. The set of required elements in V_G is calculated based on the production as $f_{V_G,m_2}(V_{G,L_2})$.

The algorithms for components E_G, E_{NA} and E_{EA} can be implemented analogous. For a dependency check based on graph diagrams, these algorithms are called for each node in the diagram. The result of this algorithm is a set of conflict descriptions. Each description points out one element that is the cause of a conflict and the two production applications that cause the conflict.

The algorithms performance should be better than the pairwise dependency check as the sets of deleted elements and required elements are only calculated once per production application instead of once for every combination that involves this production application. However, the performance of the overall algorithm depends on the performance of the underlying map that is used to store and retrieve intermediate results. A performance test of this algorithm is executed based on its implementation in Section 7.4.

In the next section we describe the detection of adaptation-consistency conflicts based on the fulfilment of nested conditions.

6.5.3. Detection of Violated Conditions

In this section we describe the analysis of fulfilment of nested conditions. The analysis is based on the current state of the models G, the set of production applications $Prods$ that are supposed to be applied and the nested conditions $Cons$.

This section assumes that all production applications in $Prods$ are parallel independent. This can be achieved by using a dependency analysis as described in Section 6.5.2 to resolve any dependencies. To answer the questions of the problem statement in Section 3.4, we need to answer the following questions:

1. **Does a condition conflict exist?** It should be possible to detect whether the result of the application of all production applications in $Prods$ fulfils all constraints in $Cons$ or not.

2. **If a condition conflict exists, which constraint is violated?** If there is a conflict it should be possible to determine which of the constraints in $Cons$ are violated and which are fulfilled.

3. **Given a conflicting combination of production applications, what is the reason for the conflict?** For a non-fulfilled condition it should be possible to detect the production applications from $Prods$ that cause the conflict. Additionally, it should be possible to point out which elementary adaptations in which production applications cause the conflict.

4. **Which subsets of production applications do not lead to a conflict?** It should be possible to determine which subsets of production applications from $Prods$ lead to a conflict-free result.

There are several potential options to detect whether the result of the application of $Prods$ fulfils all conditions in $Cons$. One way is to apply all of these production applications and then check each nested condition on the result. This approach requires applying the graph transformation productions and adapting the model although it is

not clear whether the result will be consistent. As an alternative, the parallel production application for the production applications in *Prods* can be constructed as described in Definition 2. The nested conditions in *Cons* can then be converted into left application conditions for the parallel production application as described in Section 2.2.4. If the match in the parallel production application fulfils the generated application condition the resulting graph fulfils the original nested condition.

These two methods provide information concerning the existence of a conflict (Question 1) and enable the derivation of which constraint is violated (Question 2) for the end result of the adaptation. However, they do not derive information to answer Questions 3 and 4. In this section we describe an alternate method for extracting reasons for the fulfilment of nested conditions (Question 3). These reasons can be evaluated with any subset of *Prods* to find out whether this subset fulfils the nested condition. This can be used to answer Question 4.

Nested conditions are defined in Definition 28. The basic building blocks of these conditions are *true*, $\neg$, $\wedge$ and $\exists$. All other conditions are shortcuts for combinations of these conditions. From these conditions only the existence quantifier is influenced by the structure of the graph diagram. Accordingly, this operator is the only one that is influenced by changes in the model caused by production applications in *Prods*. The evaluation of the operator $\exists(a, c)$ with morphism $a : P \to C$ for a morphism $p : P \to G$ depends on the existence of a morphism $q : C \to G$ with $q \circ a = p$ (as stated in the satisfyability of nested conditions in Definition 28). As a basis for analysis of fulfilment of existence quantifiers we analyse the influence of the production applications on the existence of the morphism q.

If the current state of the models fulfils a condition all such morphisms q in the structure of the condition can be tracked to find out whether they are preserved by a production application by checking whether any targeted elements are deleted. However, this only provides an incomplete picture. Even if one of these morphisms does not exist after the application of all *Prods*, a different morphism may still exist. Alternatively, a different disjunctive branch of the condition could be fulfilled. The adaptation may also cause the existence of an additional morphism that prevent the fulfilment of a condition although all previous morphisms are preserved.

This indicates that the analysis needs to take into account morphisms that do not exist in the current state of the models. Question 4 requires information about all subsets of *Prods*. Production applications can delete elements. Accordingly, there may be morphisms that exist when applying a subset of *Prods* for which no corresponding morphisms can be found when applying all production applications in *Prods*. Thus, it is not sufficient to reason about morphisms that exist when applying all production applications in *Prods* to answer Question 4. The analysis needs to take into account all morphisms that exist in the application of any subset of *Prods*. This requires a way to find these morphisms. Applying all subsets of *Prods* and searching for morphisms is unpractical as the number of subsets grows exponential with the size of *Prods*. Instead we utilise an object, called the maximum graph, which can be used to find these morphisms. Definition 18 defines the maximum graph.

Definition 18 (Maximum Production Application, Deletion Production Application, Maximum Graph)**.** *Given a production application* $pa_i = ((L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i), m_i)$*, the* ***maximum production application*** *for* pa_i *is the production application* $pa_{i,max} = ((K_i \overset{id_{K_i}}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i), m_i \circ l_i)$ *and the* ***deletion production application*** *for* pa_i *is the production application* $pa_{i,del} = ((L_i \overset{l_i}{\leftarrow} K_i \overset{id_{K_i}}{\rightarrow} K_i), m_i)$*.*

Given an object G *and a set of production applications* $Prods$ *to this object, the* ***maximum graph*** *is the result of* $apply(G, Prods_{max})$ *where* $Prods_{max} = \{pa_{i,max} | pa_i \in Prods\}$*.*

As a shortcut notation $max(G, Prods)$ *denotes the maximum graph of an object* G *and a set of production applications* $Prods$*.*

To create the maximum graph a production application is split into a maximum and a deletion production application, which only contain the added / deleted elements respectively. The maximum graph is a result of the application of all maximum production applications for production applications in $Prods$. The construction of a maximum graph is always possible given an object G and a set of production applications $Prods$. This is stated in the following theorem:

Theorem 6. *Given an object* G *and a set of production applications* $Prods$ *to this object, the construction of the maximum graph as defined in Definition 18 is always possible.*

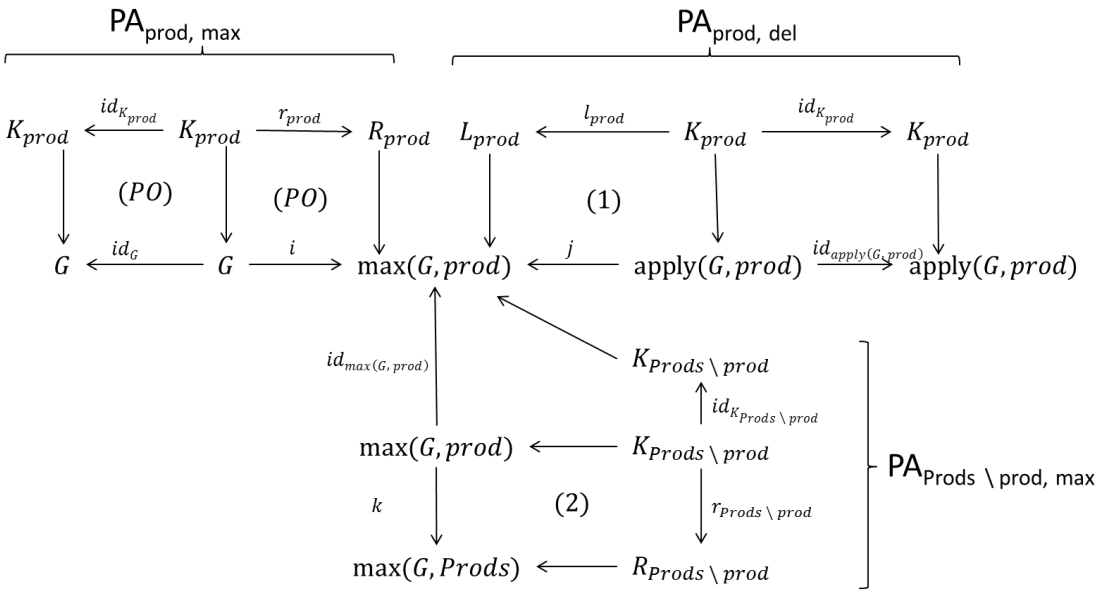
Proof for this theorem is given in Appendix B.6.

The maximum production applications add the same elements as to the original production applications but do not remove any elements. Similarly, the deletion production applications only contain the deleting part of a production application. The maximum graph is generated by using only maximum production applications. Thus, the maximum graph, generated from a model G and a set of production applications $Prods$, contains all elements from G and all elements that are added in any production application from $Prods$. Thus, the maximum graph always contains the result of the application of any subset of $Prods$. In category theory this is described by the existence of a monomorphism. The following theorem states the existence of these monomorphisms:

Theorem 7 (inclusion morphisms)**.** *Given a set of parallel independent production applications* $Prods$ *in a category* $\mathbf{TypedGraphDiagrams}^{D}_{TG}$ *to an object* G*, for each subset* $prod \subset Prods$ *there are monomorphisms:*

- $incl_{prod} : max(G, prod) \rightarrow max(G, Prods) = k$
- $map'_{prod} : apply(G, prod) \rightarrow max(G, prod) = j$
- $map_{prod} : apply(G, prod) \rightarrow max(G, Prods) = incl_{prod} \circ map'_{prod}$

where the morphisms j*, and* k *stem from the application of the production application* $PA_{prod,del}$ *and the production application* $PA_{Prods \setminus prod,max}$ *to* $max(G, prod)$ *as shown in the following figure:*



$PA_{prod,max}$ and $PA_{prod,del}$ are the maximum and deletion production application for the parallel production application PA_{prod} for all production applications in prod. The production application $PA_{Prods\setminus prod,max}$ is the maximum production application for the parallel production application $PA_{Prods\setminus prod}$ for all production applications in $Prods\setminus prod$.

This theorem is proven in Appendix B.7.

The maximum graph is an upper bound estimation of potential morphisms. For any set of production applications $prod \subset Prods$ and all morphisms q that target $apply(G, prod)$ there is a corresponding morphism q' that maps elements like q in $max(G, Prods)$. q' can be defined as $map_{prod} \circ q$. Thus, we can find all potential morphisms by searching for morphisms into the maximum graph. However, the maximum graph does not take into account the deletion of elements. Thus, it can deliver false positives of morphisms that do not exist in any subset of $Prods$ because targeted elements have been deleted. The analysis needs to reflect this fact.

The two production applications in the running example are not parallel independent. Accordingly, this section uses a different setup for illustrating the main results. This setup is shown in Figure 6.25. The example is based on the UI model and uses condition ii) of the running example, which requires each window to have at least one root layout. This condition is shown on the top of the figure. Two productions are used as example. The production P_1 moves a layout vector from one window to another. The production P_2 adds a new layout vector as content of a window. Both productions are matched into a model G that contains two windows. The match morphisms of the production applications are inclusions, meaning they match elements to elements of the same name. The figure illustrates the maximum graph $max(G, \{P_1, P_2\})$. This object contains all elements from G as well as the edge $r1$ added by P_1 and the edge $r3$ and the node $lv3$ added by P_2. The elements that are added or removed by a production application are

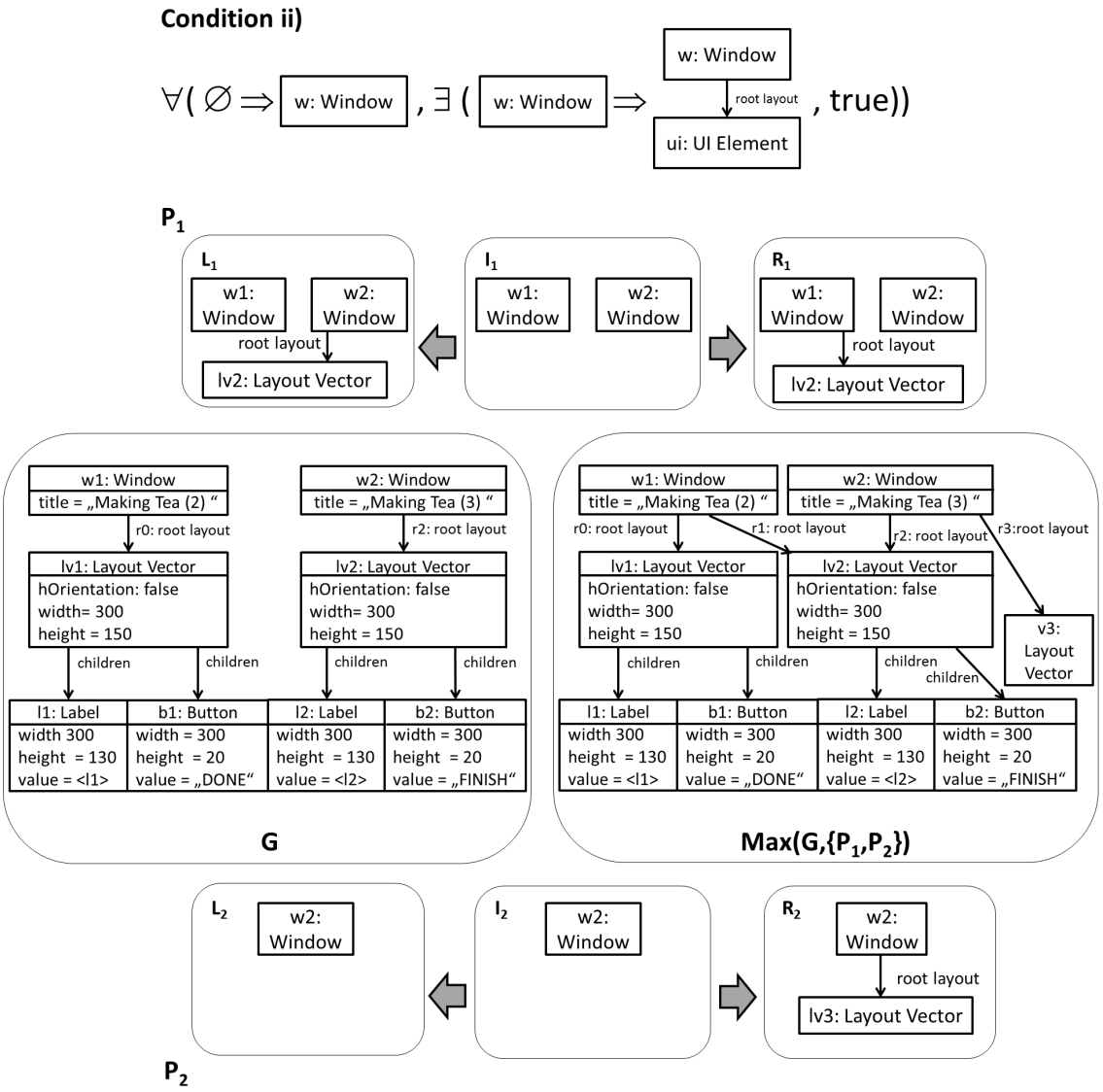


Figure 6.25.: The model, condition, rules and maximum graph for illustrating adaptation-consistency conflicts.

annotated in the figure. The edge $r2$ is removed by P_1 but is present in the maximum graph since this object does not reflect deletion.

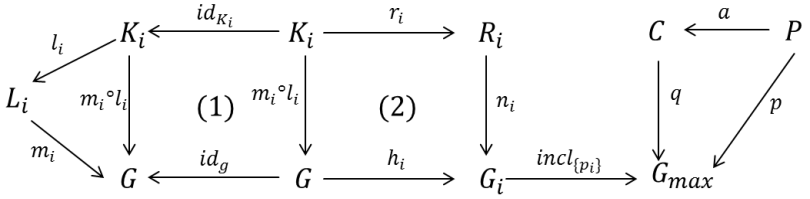


Figure 6.26.: Example: The application of a maximum production application and its embedding into the maximum graph.

Morphisms that have been found based on the maximum graph may only exist in a limited number of subsets of *Prods*. Accordingly, we need to identify in which subsets such a morphism exists. The following definition contains a collection of functions that can be used for this purpose.

Definition 19 (PossibleMorphisms, Averse Productions, Required Productions and Combinations)**.** *Given a Graph Diagram G, morphisms $a : P \to C$ and $p : P \to G_{max}$ and a set Prods of n production applications $p_i = ((L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i), m_i)$ with $i \in \{1, ..., n\}$. Furthermore, let G_{max} be the maximum graph for G and Prods with maximum production applications $p_{max,i}$ for p_i and inclusion $incl_{\{p_i\}} : max(G, \{p_i\}) \to G_{max}$ from the maximum graph of p_i into G_{max} that exists according to Theorem 7. The relevant objects and morphisms are illustrated in Figure 6.26. For every morphism $q : C \to G_{max}$ with $p = q \circ a$ the following functions are defined:*

- ***overlapAddElements**(q,p$_i$) = $incl_{\{p_i\}}(n_i(R_i)) \backslash incl_{\{p_i\}}(n_i(r_i(K_i))) \cap q(C)$ the set of all elements in G_{max} that are targeted by q and produced by p_i.*
- ***overlapRemoveElements**(q,p$_i$)= $incl_{\{p_i\}}(h_i(id_g^{-1}(m_i(L_i)))) \backslash incl_{\{p_i\}}(h_i(id_G^{-1}(m_i(l_i(K_i))))) \cap q(C)$ the set of all elements in G_{max} that are targeted by q and removed by p_i.*

These functions can be used to calculate the set of required and averse productions for a morphism as follows:

- ***required**(q,Prods) = $\{p_i, p_i \in Prods \wedge overlapAddElements(q, p_i) \neq \emptyset\}$ the set of production applications from Prods that are required for the existence of q.*
- ***averse**(q,Prods) = $\{p_i, p_i \in Prods \wedge overlapRemoveElements(q, p_i) \neq \emptyset\}$ the set of production applications from Prods that forbid the existence of q.*

Using these sets the set of morphisms that can exist in any subset of Prods can be derived with

- ***possibleMorphisms(****Prods,p,a****)*** $= \{q : C \to G_{max} | p = q \circ a \wedge required(q, Prods) \cap averse(q, Prods) = \emptyset\}$.

The functions in Definition 19 enable an analysis with respect to the existence of one morphism and the sets of production applications that are required or averse to the existence of this morphism. For each morphism we can calculate a set of averse production applications by using the function *averse*. The application of any of these production applications leads to the non-existence of the morphism. This set contains all production applications that remove any element that is mapped by the morphism. The removal of this element is the reason for the non-existence. The set of these elements is calculated using the function *overlapRemoveElements*. It contains subsets of the nodes and edges as well as attribute edges for each node of the diagram.

In addition, we can calculate the set of production applications that are required for the existence of a morphism. If any of these production applications is not applied the morphism does not exist. The function *required* can be used for this purpose. This set of production applications is calculated by finding all elements that are added by the production application and targeted by the morphism. These elements do not exist if this production is not applied and thus cannot be targeted by the morphism. This set is calculated by the function *overlapAddElements*. As with *overlapRemoveElements* this function returns a subset of the elements for each diagram.

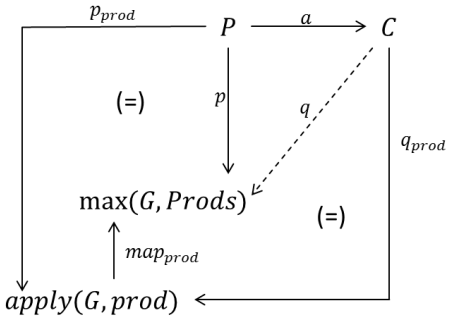
In some cases a production application p_i is both required and averse. This is the case if the production application adds targeted elements but removes other targeted elements. The production application p_i is required for the morphism to exist but at the same time forbids it. This morphism cannot exist in any subset of $Prods$. It is a false positive which has been found because the maximum graph does not regard deletion. The function *possibleMorphisms* filters morphisms by this criteria and only contains the commuting morphisms that exist when applying at least one subset of $Prods$.

The following theorem states that these functions behave as expected:

Theorem 8. *Given:*

- *an algebra* D
- *a D- type graph diagram* TGD
- *a typed graph diagram* G *from* $\mathbf{TypedGraphDiagrams}_{TG}^{D}$
- *a set* $Prods$ *of* n *parallel independent production applications* $p_i = ((L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i), m_i)$ *with* $i \in \{1, ..., n\}$ *in* $\mathbf{TypedGraphDiagrams}_{TG}^{D}$ *to* G
- $a : P \to C$ *a morphism from* $\mathbf{TypedGraphDiagrams}_{TG}^{D}$
- $p : P \to max(G, Prods)$ *a morphism from* $\mathbf{TypedGraphDiagrams}_{TG}^{D}$
- $map_{prod} : apply(G, prod) \to max(G, Prods)$ *the inclusion of the application of a subset* $prod$ *of* $Prods$ *into the maximum graph as described in Theorem 7*

as summarised in the following figure:



The following statements hold:

1. *For any morphism* $q : C \to G_{max} \in possibleMorphisms(Prods, p, a)$ *with* $p = q \circ a$ *the function* $required(q, Prods)$ *denotes exactly the subset of production applications from* $Prods$ *that are required for the existence of an equivalent morphism.*

   Formally:

   $$\begin{aligned}
   &\forall p \in Prods. \\
   &\; p \in required(q, Prods) \\
   &\; \Leftrightarrow \forall prod \subset Prods. \\
   &\quad (\exists q_{prod} : C \to apply(G, prod).q = map_{prod} \circ q_{prod}) \\
   &\quad \to p \in prod
   \end{aligned}$$

2. *For any morphism* $q : C \to G_{max} \in possibleMorphisms(Prods, p, a)$ *with* $p = q \circ a$ *the function* $averse(q, Prods)$ *denotes exactly the subset of production applications from* $Prods$ *whose application forbids the existence of* q*.*

   Formally:

   $$\begin{aligned}
   &\forall p \in Prods. \\
   &\; p \in averse(q, Prods) \\
   &\; \Leftrightarrow \forall prod \subset Prods. \\
   &\quad p \in prod \\
   &\quad \to \neg(\exists q_{prod} : C \to apply(G, prod).q = map_{prod} \circ q_{prod})
   \end{aligned}$$

3. $possibleMorphisms(Prods, p, a)$ *contains all morphisms that are possible in any subset of* $Prods$*, meaning the following equation holds for each potential morphism* q *with* $p = q \circ a$*:*

   $$\begin{aligned}
   &q \in possibleMorphisms(Prods, p, a) \\
   &\Leftrightarrow \exists prod \subset Prods. \\
   &\quad \exists p_{prod} : P \to apply(G, prod). \\
   &\quad\; \exists q_{prod} : C \to apply(G, prod). \\
   &\quad\quad p = map_{prod} \circ p_{prod} \wedge q = map_{prod} \circ q_{prod} \wedge p_{prod} = q_{prod} \circ a
   \end{aligned}$$

The theorem is proven in Appendix B.8.

The algorithms for extracting information about reasons for conflicts utilize a specific notation for representing these reasons. This notation is called fulfilment condition. It is a condition in the sense that it can be evaluated to true or false with a subset of $Prods$ to derive whether that subset fulfils the nested condition that has been used to generate the fulfilment condition. The structure of the fulfilment condition can be analysed to derive additional information about the fulfilment of the nested condition it has been generated with. The notation is defined as follows:

Definition 20 (Fulfilment Conditions)**.** *Given a set of production applications Prods to Graph Diagram G and a constraint c the following notation is used to denote* ***fulfilment conditions****:*

- $true$ *- the constraint is always fulfilled*
- $false$ *- the constraint cannot be fulfilled*
- $Deletes(pa_i, elem)$ *- production application* $pa_i \in Prods$ *removes element elem*
- $Adds(pa_i, elem)$ *- production application* $pa_i \in Prods$ *adds an element elem*

Given a set of n fulfilment conditions s_i *with* $i \in \{1, ..., n\}$, $\wedge_{i\in\{1,...,n\}} s_i$, $\vee_{i\in\{1,...,n\}} s_i$ *and* $\neg s_i$ *are also fulfilment conditions.*

The method $eval : FulfilmentCondition \times ProductionApplication^* \rightarrow \{True, False\}$ *evaluates fulfilment conditions the following way:*

- $eval(true, Prods) = True$
- $eval(false, Prods) = False$
- $eval(Deletes(pa_i, elem), Prods)) = \begin{cases} True & \text{if } pa_i \in Prods \\ False & \text{else} \end{cases}$
- $eval(Adds(pa_i, elem), prods)) = \begin{cases} True & \text{if } pa_i \in Prods \\ False & \text{else} \end{cases}$
- $eval(\wedge_{i\in\{1,...,n\}} s_i, Prods) = \wedge eval(s_i, Prods)_{i\in\{1,...,n\}}$
- $eval(\vee_{i\in\{1,...,n\}} s_i, Prods) = \vee eval(s_i, Prods)_{i\in\{1,...,n\}}$
- $eval(\neg p, Prods) = \neg eval(p, Prods)$

The elementary conditions in this notation are $true$, $false$, $Deletes(pa_i, elem)$ and $Adds(pa_i, elem)$. The conditions $Deletes$ and $Adds$ are fulfilled if production application pa_i deletes or adds an element $elem$. In the scope of graph diagrams $elem$ can be a node, edge, attribute node or attribute edge from one of the attributed typed graphs that is contained as a diagram node. Mappings between diagrams are reflected implicitly by the deletion / creation of the mapped elements. $false$ denotes that the constraint is

not fulfillable and *true* denotes that the constraint is always fulfilled. From these elementary conditions we can construct formulas representing complex conditions using the operations $\vee$, $\wedge$ and $\neg$. A fulfilment condition can be seen as a formula in propositional logic. As in propositional logic truncation rules can be used to simplify these formulas.

The basis for the detection algorithm for fulfilment conditions for nested conditions is a detection algorithm for fulfilment conditions that describe the existence of a morphism. These conditions are used to describe the fulfilment of existence quantifiers. They are derived as follows:

Definition 21 (Conditions for the existence of a morphism)**.** *Given a set of production applications* $Prods = p_i, i \in \{1, ..., n\}$ *to a graph diagram* G *with maximum graph* G_{max} *and a morphism* $q : Q \to G_{max}$*, the existence condition for* q *is defined as follows:*

- $positiveCondition(Prods, q) = \wedge_{p_i \in required(p,Prods)}(\wedge_{elem \in overlapAddElements(q,p_i)}(Adds(p_i, elem)))$
- $negativeCondition(Prods, q) = \vee_{p_i \in averse(p,Prods)}(\vee_{elem \in overlapRemoveElements(q,p_i)}(Removes(p_i, elem)))$
- $existenceCondition(Prods, q) = positiveCondition(Prods, q) \wedge \neg negativeCondition(Prods, q)$

The existence condition for morphism is calculated as a conjunction of a positive condition and a negated negative condition. The positive condition is a conjunction of adds statements for all elements targeted by the morphism that have been added by any required production application. The negative condition is a disjunction of removes statements for all elements that are targeted by the morphism and removed by any averse production application.

The algorithm for existence conditions is illustrated on the running example in Figure 6.27. The condition shown in the figure is equivalent to the original condition ii). Its all-quantifier has been substituted with the pattern of existence quantifier and negation it abbreviates. The figure shows all morphisms that exist for one object of the inner condition. The matched object contains a window and a UI element connected by an edge of type *rootlayout*. This edge can be matched to the four edges $r0$, $r1$, $r2$ and $r3$ in the maximum graph. The existence conditions for these morphisms are annotated on the dashed arrows. A morphism to $r0$ is always possible because none of the targeted elements are added or deleted by any production application. Accordingly, the positive condition is *true*, the negative condition is $false$ and the overall existence condition resolves to *true*. The mapping to $r1$ is only possible if P_1 is applied since this edge is added by this production application. Accordingly, the positive condition is $Adds(P_1, r1)$. No targeted elements are deleted and thus the negative condition is $false$. The overall existence condition resolves to $Adds(P_1, r1)$. Similarly, the mapping to $r3$ is only possible if P_2 adds $r3$ and $lv3$ and accordingly the existence condition is $Adds(P_2, r_3) \wedge Adds(P_2, lv3)$. Since edge r_2 is removed by P_1 the mapping to this edge contains a negative condition $Removes(P_1, r2)$. The existence condition of this morphism is $\neg Removes(P_1, r2)$.

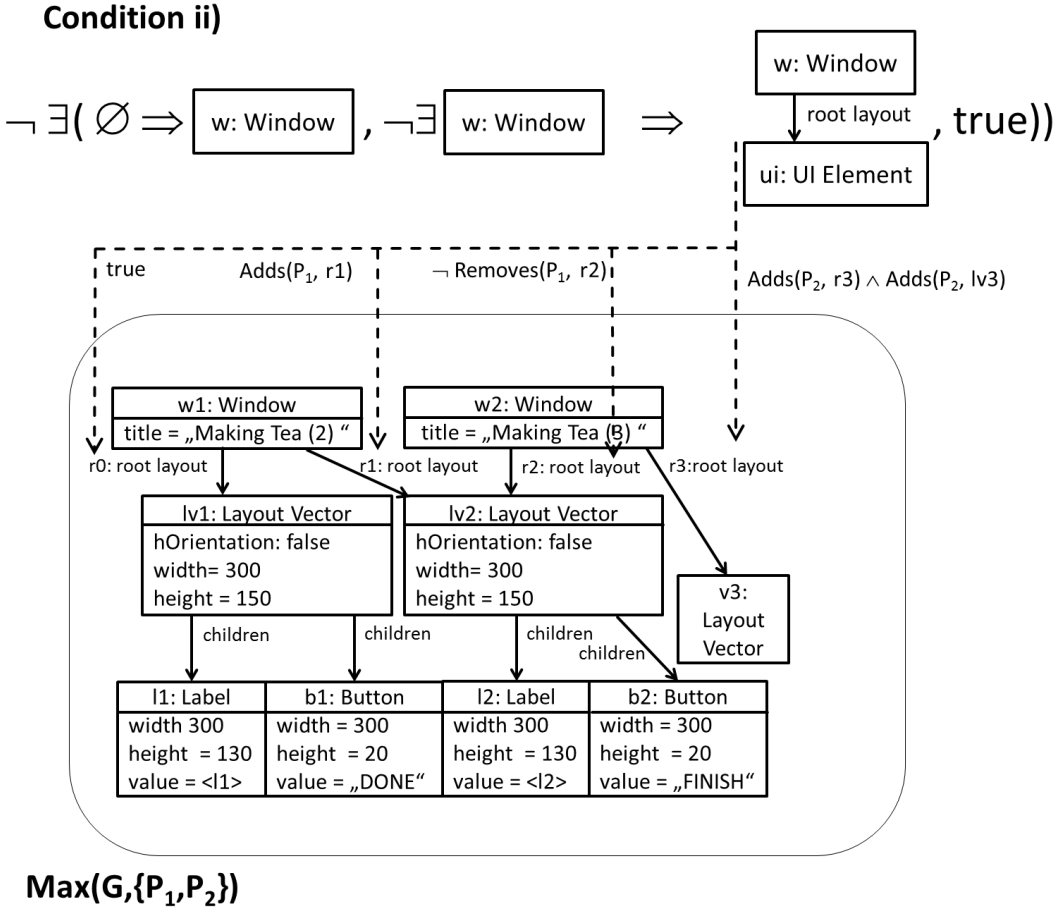


Figure 6.27.: Example: Morphisms into the maximum graph and their existence condition.

The following theorem states the correct behaviour of the derivation of existence conditions for morphisms:

Theorem 9. *Given a set of parallel independent production applications* $Prods = p_i, i \in \{1, ..., n\}$ *to a graph diagram* G *the following holds:*

- *For a given morphism* $q : Q \to max(G, Prods)$*:*

  $\forall prod \subset Prods : eval(existenceCondition(Prods, q), prod) = True \Leftrightarrow$
  $\exists q' : Q \to apply(G, prod) : q = map_{prod} \circ q'$ *where* map_{prod} *is the inclusion from* $apply(G, prod)$ *to* $max(G, Prods)$ *according to Theorem 7.*

Proof for this theorem can be found in Appendix B.9.
The algorithm for fulfilment of a nested condition is defined as follows:

Definition 22 (Derivation Algorithm *conditions* for Fulfilment of Constraints)**.** *Given a graph constraints c over an object P and a set of production applications $Prods$ to a graph diagram G, the algorithm* ***conditions*** *for generating the fulfilment conditions for the fulfilment of c with a morphism $p : P \to max(G, Prods)$ is as follows:*

- $conditions(true, p, Prods) = true$
- $conditions(\neg(j), p, Prods) = \neg condition(j, P, Prods)$
- $conditions(\wedge_{j \in J} c_j, p, Prods) = \wedge_{j \in J} conditions(c_j, p, Prods)$
- $conditions(\exists(a, c), p, Prods) = \vee_{q \in possibleMorphisms(Prods,p,a)}($ $existenceCondition(q, Prods) \wedge conditions(c, q, Prods))$

We define the algorithm component-wise over the elementary constructs that can be used to generate nested conditions. The constraint **true** is always fulfilled and thus resolved to *true*. The **negation** is resolved to a negation of the inner constraint. The **conjunction** of constraints is resolved to a conjunction of their condition. The **existence** constraint is resolved to a disjunction over all existing commuting morphisms calculated by the function *possibleMorphisms*. The existence of one of these morphisms is enough for the existence constraint to be fulfilled. For each of these morphisms the inner constraint consists of the condition for the existence of this morphism, calculated by the function *existenceCondition*, in conjunction with the condition for the fulfilment of the inner constraint, calculated by a recursive call to *conditions*.

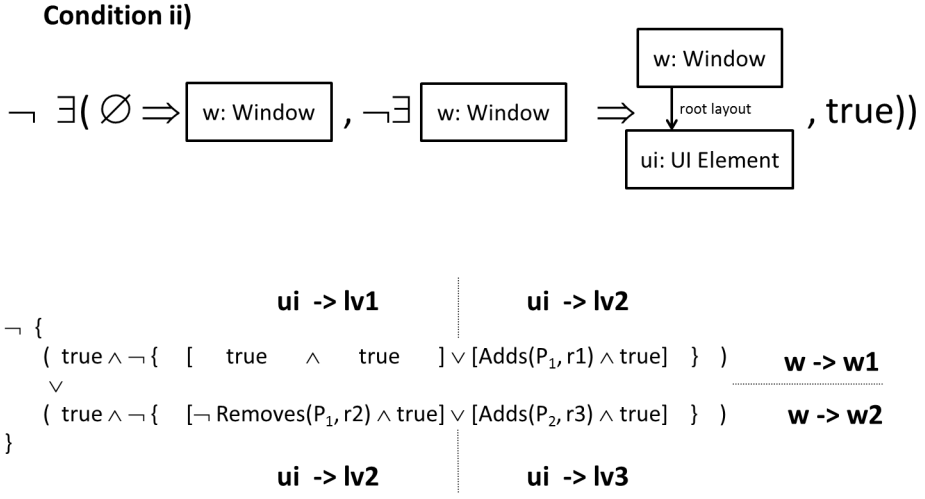


Figure 6.28.: Example: The result of the analysis for fulfilment conditions.

Figure 6.28 shows the result of the analysis of fulfilment conditions in the running example. Condition ii) has a negation as outer constraint. Accordingly, the fulfilment

condition starts with a negation. The existence quantifier for the node w of type window can be fulfilled by two morphisms, mapping w to $w1$ or $w2$. The two rows show the inner conditions for these two mappings. Both windows are not touched by any production application and thus their fulfilment condition is $true$. The inner condition starts with a negation. Accordingly, the next inner fulfilment condition also starts with a negation. The mapping of the inner existence quantifiers are the morphisms described in Figure 6.27. Accordingly, they have the existence condition described in that figure. Mapping ui in the first row to $lv1$ is always possible (the existence condition is $true$) and mapping ui to $lv1$ has the existence condition $Adds(P_1, r1)$. Similarly, the existence conditions for mapping ui to $lv2$ and $lv3$ in the second row are calculated. Since the inner condition of the inner existence condition is $true$, these existence conditions are concatenated with $\wedge true$.

The correct behaviour of this algorithm is stated in the following theorem:

Theorem 10. *Given*

- *An algebra D*
- *a type graph diagram TG*
- *a nested condition c in the category $\boldsymbol{TypedGraphDiagrams}^D_{TG}$ over object P with finitely many nesting levels*
- *a graph diagram G from $\boldsymbol{TypedGraphDiagrams}^D_{TG}$*
- *a set of parallel independent production applications $Prods$ in $\boldsymbol{TypedGraphDiagrams}^D_{TG}$ to G*
- *a morphism $p : P \to max(G, Prods)$ from category $\boldsymbol{TypedGraphDiagrams}^D_{TG}$*

the algorithm conditions is correct and complete meaning, for an arbitrary set of production applications $prod \subset Prods$, the following equation holds:

- $eval(existenceCondition(Prods, p) \wedge conditions(c, p, Prods), prod) = true \Leftrightarrow \exists p' : P \to apply(G, prod) : p = map_{prod} \circ p' \wedge p' \models c$

where $map_{prod} : apply(G, prod) \to max(G, Prods)$ is the morphism from the application of productions to the maximum graph that exists according to Theorem 7.

Appendix B.9 gives proof for this theorem.

Depending on the intended use of the fulfilment condition they can be converted into other equivalent formulas using Boolean equivalences. For instance, the formula can be converted into a conjunctive or disjunctive normal form. The target form depends on the requirements of the conflict resolution approach. The fulfilment condition in the running example can be converted to:

- $\neg Removes(P_1, r2) \vee Adds(P_2, r3)$

This condition denotes that all subsets of $\{P_1, P_2\}$ that do not contain P_1 or that contain P_2 fulfil Condition ii) when applied.

The fulfilment condition can be used to derive reasons for the fulfilment of a nested condition (Question 3) by reasoning about why the condition is evaluated to false, given a set of production applications. It can be evaluated with any set of production applications to derive whether this set fulfils the condition. This answers Question 4.

In fact, we can answer all four questions by calculating the fulfilment condition for all constraints using the set of all activated productions applications. If for a constraint the fulfilment condition can be resolved to *true* the constraint is fulfilled. If not the constraint is not fulfilled. This enables to answer Question 1 (Does a condition conflict exists?) and Question 2 (which constraint is violated?).

Although the calculation of $conditions(c, p, Prods)$ for a constraint enables to answer all other questions in addition to Question 4 it may not be the most efficient way to answer these questions. This calculation of fulfilment conditions requires:

- the generation of maximum production applications
- the application of maximum production applications that are required to generate the maximum graph
- an iteration over the constraint structure
- for each existence quantifier, the iteration over all morphisms that exist into the maximum graph
- for each morphism into the maximum graph an iteration over all added and deleted elements of this morphism
- truncation of the resulting fulfilment condition

Answering Questions 1 and 2 first as described in the beginning of this section is cheaper as it only takes into account the end result of the adaptation. At run time resources and time are critical. Therefore, this option may be preferred in cases where only the end result of the adaptation needs to be consistent. The calculation of fulfilment conditions can be delayed until a conflict is detected and it becomes necessary to answer Question 3 and 4.

The final decision on which of the algorithms to use for conflict detection is dependent on the requirements of the running system and should therefore be made by the developer of the modelling framework or the developer of the software system.

6.5.4. Sequential Conflict Detection

The preceding sections focus on adaptation conflicts between production applications that are applied in parallel. In this section we extend the approach to enable the detection of conflicts involving previously applied adaptations.

In graph transformation already applied production applications can be reverted again under certain circumstances. Accordingly, it makes sense to take previously applied

adaptations into account during conflict detection as they may be reverted to resolve a conflict. However, the reversion of production applications is not always possible. The production application can represent a change that is imposed by the environment or the user and is not allowed to be reverted. Dependencies between graph transformation productions can also complicate reversion. If sequentially dependent graph transformation productions have been applied the reversion of one production application might require the reversion of other production applications that are sequentially dependent on this one. In order to distinguish between adaptations that can be reverted and adaptations that cannot the analysis is extended with a set of reversible production applications that represent all applied adaptations that can be safely reverted. Thus, in addition to the set $Prods$ of production applications to be applied, the algorithms in Sections 6.5.2 and 6.5.3 can take into account this set $Revers$ of production applications that, if applied, reverse the effect of a previously applied production application.

This section is divided in three subsections. The first one describes one way of maintaining a set of reversible production applications. Subsequently, we discuss the integration of reversible production applications into the detection of dependencies between graph transformation productions. Finally, we describe the integration of reversible production applications into the analysis of fulfilment of conditions.

Maintenance of Reversible Production Applications

In order to deal with sequential conflicts the description of the current state of the application is enhanced with a new component. In addition to the current state D and the currently applicable production applications $Prods$, we add a set of production applications $Revers$, which can be reverted to undo previous adaptations, to the analysed information from Section 6.5.1. In this section we describe how this set of production applications can be maintained.

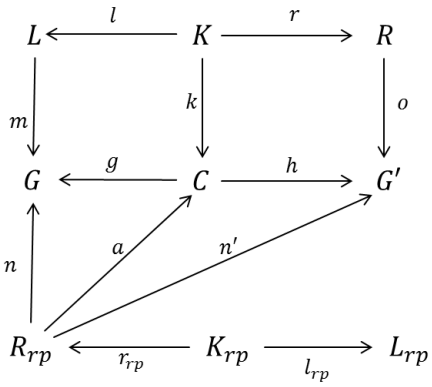
In fact, this set could be built in different ways. For the purpose of this dissertation the reversible production applications are those production applications that can be immediately reverted (i.e. no sequentially dependent production application has been applied after them). Other ways to construct such a set could be to merge all production applications that have to be reverted together into one or to partially revert production applications that cannot be completely reverted. These strategies depend on the specific run time framework. However, as long as the set of reversible production applications is available, it can be integrated into conflict analysis as described in the next two sections. Thus, we do not go into detail on other strategies and use the strategy described above.

The following definition describes how reversible production applications can be created and how the set $Revers$ can be maintained.

Definition 23 (Generation and Maintenance of Reverse Production Applications)**.** *Given a production application* $p = ((L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R), m)$ *applied to a model* G *the* ***reverse production application*** *is defined as* $rp = ((R \overset{r}{\leftarrow} K \overset{l}{\rightarrow} L), n)$*, where* n *is the comatch of the direct transformation implied by* p*.*

Given a reverse production application $rp = ((R_{rp} \overset{r_{rp}}{\leftarrow} K_{rp} \overset{l_{rp}}{\rightarrow} L_{rp}), n)$ *to a model* G

and a production application $p = ((L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R), m)$ to G, rp can be maintained after the application of p, iff the productions rp and p are parallel independent.



If rp can be maintained, the production application can be updated to a production application $rp' = ((R_{rp} \overset{r_{rp}}{\leftarrow} K_{rp} \overset{l_{rp}}{\rightarrow} L_{rp}), n')$. $n' : R_{rp} \rightarrow G'$ is derived from the application of p (as depicted in the above figure). It is defined as $n' = h \circ a$ where $a : R_{rp} \rightarrow C$ fulfils $n = g \circ a$. This morphism exists because rp and p are parallel independent.

A reverse production application contains the reversed production from the original production application (left and right side are switched) and the comatch of the original production application as match morphisms. The application of this production application reverses the effects of the original production application. Since the match of a production application targets the current model it is necessary to update the match of all reverse production applications whenever the model changes. Thus, the matches have to be updated when a new production application is applied. This is only possible when the applied production application and the updated reverse production application are parallel independent. This assures that after the application of the new production application the reverse production application can still be matched and that the current production application is still applicable after the reversion production application is applied. Parallel independence can be tested as described in Section 6.5.2 and is available as a result of the detection of parallel dependencies. If the reverse production application is still applicable it can be updated with a new match morphism and kept. If it is not applicable, it is removed from the set of reversible production applications.

One way to integrate reversible production applications into conflict detection is to apply all reverse production applications in *Revers*, reverse them again (i.e retrieve the original production application) and add them to the set of production applications *Prods* that is used to find applicability and condition conflicts. However, this approach requires the reversion of past adaptations every time a new production is to be applied. A better way to deal with conflict analysis is to integrate *Revers* into the existing analysis methods. This is described in the next two sections.

Integration of Reversible Production Applications into Dependency Analysis

The analysis of dependencies between production applications in Section 6.5.2 is based on the set $Prods$. It detects parallel dependencies between these productions. This algorithm can be executed on $P = Prods \cup Revers$ to involve $Revers$ into this analysis. This enables the dependency analysis to yield additional results that describe dependencies between production applications in $Prods$ and $Revers$. Since reverse production applications are also production applications this is possible. However, the analysis result is interpreted in a different way if reverse production applications are involved.

The conflict detection algorithm described in Figure 6.24 generates a set of conflict descriptions of the form *ConflictDescription(requiring,conflicting,element)* where *requiring* is a production application that requires element *element* and $conflicting$ is a production application that deletes *element*. Based on which sets the dependent production applications belong to we can distinguish four cases:

1. **normal dependency:** Both production applications belong to the set $Prods$. No reverse production application is involved n this case. It is the case described in Section 6.5.2.

2. **reverse dependency:** Both production applications belong to the set $Revers$. This reflects a dependency between reversible production applications and is not an immediate problem. However, it denotes that it is not possible to apply both reversions and needs to be taken into account during conflict resolution in case their reversion is required.

3. **dependent adaptation:** *application* is in $Prods$ and $conflicting$ is in $Revers$. It is not possible to apply the adaptation described in *requiring* if $conflicting$ is applied to reverse a previous change.

4. **dependent reversion:** *requiring* is in $Revers$ and conflicting is in $Prods$. It is not possible to apply the reversion once the production application $conflicting$ has been applied.

In case three and four the reversion is parallel dependent to one current adaptation. In both cases it is not possible to revert the production application any more after the current adaptation has been applied, either because the reverted production application provided a foundation for the current one or because some elements that needed to be reverted have been removed by the current adaptation. In these cases the reverse production application needs to be removed from $Revers$ if the conflict resolution decides to apply the production application.

Not all of these dependencies necessarily represent a problem. It is up to the conflict resolution mechanism to decide whether a dependency is considered faulty behaviour or not, based on the knowledge that there is a dependency.

Integration of Reversible Production Applications into Condition Analysis

The analysis method for adaptation-consistency conflicts can also be execute with set $P = Prods \cup Revers$ to take into account the reversion of production applications. Accordingly, the fulfilment condition can be evaluated with subsets from P, thus finding out whether the reversion of previously applied adaptations leads to a consistent result. While technically nothing changes, the tested combinations should be interpreted in a different way. When taking into account only $Prods$, the optimal result would be that all analysed production applications can be applied and the result fulfils the nested condition. This is not the case when analysing P as the running system should avoid unnecessary reversions. Accordingly, the optimal result would not be that all production applications in P fulfil the nested condition when applied but that all production applications in $Prods$ fulfil the condition when applied. The main difference here is for a resolution mechanism which should try to apply as many productions from $Prods$ and as little productions from $Revers$ as possible.

When interpreting a fulfilment condition with respect to the added and deleted elements the interpretation of these statements also changes when considering a change made by a reverse production application. The reverse production application rpa represents the reversion of a production application pa that has already been applied. Thus, the statements $Deletes(rpa, elem)$ and $Adds(rpa, elem)$ denote that the reversion of pa deletes or adds an element, which is a reason for the constraint being fulfilled. Since rpa is derived from an already applied production application pa in such a way that it reverses its changes, the statement $Deletes(rpa, elem)$ denotes that pa has originally added $elem$. $Adds(rpa, elem)$ can be interpreted as pa removed $elem$. Accordingly, the fact that pa was applied prevented the condition from being fulfilled for those reasons.

The next section describes how the conflict detection algorithms can be used to provide knowledge for a conflict resolution mechanism.

6.6. Provision of Knowledge for Conflict Resolution

The focus of this thesis is the detection of adaptation conflicts and the provision of knowledge about them. The main goal of the detection of conflicts is to generate information that can be used for conflict resolution. In this section we describe how the required information can be derived from the results of the detection of adaptation-adaptation and adaptation-consistency conflicts.

The basis for our analysis is the information described in Section 6.5.1 and the set of reversible production applications defined in Section 6.5.4. Accordingly, the following information is used as input to conflict detection:

- Situation-independent information:

  TGD: a type graph diagram

  $Cons$: a set of Nested Conditions

  Ads: a set of Adaptations

- Situation-dependent Information:

  G the current state of the models

  $Prods$ the currently activated production applications

  $Revers$ the reversible production applications

The analysis consists of two parts. The first part is the analysis of dependencies among graph transformation productions to detect adaptation-adaptation conflicts. The second part is the analysis for fulfilment of nested conditions. This analysis detects adaptation-consistency conflicts. For both kinds of conflicts the five questions from the problem statement (cf. Section 3.4) need to be answered. The following two sections contain information on how the required information for answering these questions can be derived from the result of the conflict detection.

6.6.1. Answering the Questions for Adaptation-Adaptation Conflicts

The answers to the five questions can be provided by the algorithm described in Section 6.5.2 and the extension for sequential conflicts described in Section 6.5.4. The result of this algorithm is a set of descriptions of parallel dependencies *deps* of the form *ConflictDescription(requiring,conflicting,element)*. This description denotes that a production application $conflicting$ deletes an element *element* which is required to apply production application $requiring$. These dependencies can be used to answer the five questions as follows:

Which conflicts do occur?

Each dependency in *deps* represents one parallel dependency and thus one adaptation-adaptation conflict. An adaptation-adaptation conflicts is detected if $deps \neq \emptyset$.

Which adaptations are in conflict?

For each $ConflictDescription(requiring, conflicting, element) \in deps$ the two production applications $conflicting$ and $requiring$ are in conflict with each other.

Which elementary adaptations lead to the conflict?

For each $ConflictDescription(requiring, conflicting, element) \in deps$ the elementary adaptations that lead to this conflict are the deletion of *element* by $conflicting$ and the fact that $requiring$ needs *element*.

Which model elements are involved in the conflict?

The model elements that are involved in a conflict are:

- $conflictingElements = \{elem | \exists ConflictDescription(\_, \_, elem) \in deps\}$

For any two production applications a and b the elements that cause the conflict between these two production applications are:

- $conflictingElements(a, b) = \{elem | ConflictDescription(a, b, elem) \in deps\}$

Which subsets of adaptations can potentially be applied?

The applicable combinations are all subsets $prod \subset Prods$ for which the following holds:

- $\forall ConflictDescription(a, b, elem) \in deps.\neg(a \in prod \wedge b \in prod)$

An application sequence seq of production applications in $Prods$ that contains each production application once can be applied if one of the following statements holds for all $ConflictDescription(a, b, elem) \in deps$:

- a is not in seq
- b is not in seq
- b is applied before a

The sequence seq can contain parallel dependent production applications, if they are applied in the right order.

6.6.2. Answering the Question for Adaptation-Consistency Conflicts

The five questions for adaptation-consistency conflicts can be answered using the algorithms from Section 6.5.3 and the extension for the detection of sequential conflicts described in Section 6.5.4. The fulfilment condition fc generated by the algorithm *conditions* is used to answer these questions. The questions can be answered as follows:

Which conflicts do occur?

Given a set of parallel independent production applications an adaptation-consistency conflict occurs if $eval(fc, Prods) = false$.

If a modelling framework requires intermediary states of an application sequence seq of production applications from $Prods$ to be consistent then an adaptation-consistency conflict also occurs if $eval(fc, prod) = true$ where $prod$ is an intermediary set of production applications that results in the partial application of seq.

Which adaptations are in conflict?

The fulfilment condition contains statements of the form *Adds* and *Deletes* that state that a production application adds or removes a certain element. The production applications that are contained in these statements are the reason for fulfilment or non-fulfilment of the nested condition. This can be used to derive information about which production applications are responsible for the non-fulfilment.

This information can be derived by building the disjunctive normal form of the fulfilment condition. In this form each disjunctive clause represents a set of solutions for which the nested condition is fulfilled. The statements *Ads* and *Deletes* in each clause are either negated or non-negated. This enables a clause-wise reasoning about the fulfilment of a condition. For example the clause

$[Adds(p1, a), Deletes(p2, b), \neg Adds(p3, c)]$

is fulfilled when $p1$ adds a and $p2$ deletes b and is not fulfilled if $p3$ is applied, because $p3$ adds the element c.

In each clause the production applications whose *Adds* and *Deletes* statements are negated are in conflict with those production applications whose *Adds* and *Deletes* statements are not negated. In the example, $p3$ is in conflict with $p1$ and $p2$ as the nested condition would be fulfilled if $p3$ is not applied when $p1$ and $p2$ are.

Which elementary adaptations lead to the conflict?

The *Adds* and *Deletes* statements from the fulfilment condition directly represent the elementary creating and deleting operations that lead to the conflict. They can be used to find out which elementary adaptations of the conflicting production applications are responsible for the non-fulfilment of the nested conditions. Again, the conflict detection can be done clause-wise in the disjunctive normal form to derive information for each combination of production applications that would fulfil the condition.

Which model elements are involved in the conflict?

The model elements can be pointed out in the maximum graph as the added elements are not reflected in G. The set of elements that are involved in a conflict are all elements e for which a statement $Adds(e,)$ or $Deletes(e,)$ is contained in fc.

Again, a case-based reasoning in the disjunctive normal form can be executed to distinguish the conflicting elements for each potential solution.

Which subsets of adaptations can potentially be applied?

The set of fulfilling subsets is defined as follows:

- $fulfilling = \{prod \subset Prods | eval(fc, prod) = true\}$

This set can be tested element-wise by evaluating fc.

The complete set $fulfilling$ can also be derived without checking each element in the power set of $Prods$. This can be done based on the disjunctive normal form of fc. In this form each clause represents a set of solutions. Production applications that are not contained in any *Adds* or *Deletes* statement in the clause are neutral to the evaluation of this clause and can be applied without effect on the nested condition.

6.7. Summary

In this chapter we described the Trollmann approach. The approach contains a formalism for representing models, model relations, adaptations and consistency requirements. Algorithms for the detection of adaptation conflicts are based on this formalism.

For the representation of related models we defined the language of graph diagrams. Graph diagrams form a diagram structure of models, represented by attributed typed graphs, and model relations, represented by attributed typed graph morphism. Graph diagrams are an $\mathcal{M}$-adhesive category and thus can be combined with graph transformation and nested conditions to represent adaptations and consistency requirements.

In our formalism adaptation-adaptation conflicts can be detected by finding dependencies between graph transformation productions. We extended the existing mechanisms for finding such dependencies to enable the analysis of more than two productions and to extract information about the reasons for dependencies.

Adaptation-consistency conflicts can be detected by detecting which subsets of graph transformation productions fulfil a nested condition when applied. We developed an analysis algorithm that answers this question for all subsets of a set of graph transformation productions. The result is a fulfilment condition that encodes which subsets of production applications fulfil the nested condition. The fulfilment condition also contains additional information about the reason for the conflict.

The algorithms for the detection of adaptation-adaptation and adaptation-consistency conflicts are defined for parallel conflicts. In order to enable a detection of sequential conflicts we added a set of reversion production applications that can be applied to revert previous adaptations. We discussed how this set can be integrated into both conflict detection algorithms.

Based on the formalism and the detection algorithm in the Trollmann approach we are able to retrieve information about adaptation conflicts and the reasons for them. In Chapter 3 the required information has been represented as questions that need to be answered about adaptation conflicts. We described how these questions can be answered for adaptation-adaptation conflicts and adaptation-consistency conflicts based on the result of our conflict detection algorithms.

The next chapter describes an implementation of our analysis methods and an application of this implementation to the running example.

7. Implementation

This chapter presents the implementation of our main algorithms using the Eclipse Modeling Framework. The implementation is an extension of the Henshin plugin [113], an Eclipse plugin able to express graph transformation and nested conditions based on EMF models. Our implementation extends the Henshin plugin with capabilities to detect adaptation-adaptation and adaptation-consistency conflicts. For this we implemented the algorithms defined in Chapter 6.

EMF and Henshin do not use graph diagrams as a model formalism. Thus, it is necessary to clarify the relation of EMF models and the formalisms used in this chapter. For this we review the basics of EMF and Henshin and their relation to the formalism of attributed graphs (cf. Section 7.1) and describe how graph diagrams can be integrated into this framework (cf. Section 7.2). Subsequently, we describe the implementation of conflict detection (cf. Section 7.3). The running example from Section 6.3 is implemented and evaluated to test the implementation (cf. Section 7.4). Finally, we conclude the chapter with a summary (cf. Section 7.5).

7.1. Relation of EMF and Henshin to Attributed Typed Graphs

The formalism of graph diagrams is based on attributed typed graphs. This section describes the Eclipse Modeling Framework [20], the Henshin plugin and their relation to attributed typed graphs. This is a foundation for the extension to graph diagrams discussed in the next section.

The EMF is a framework that allows to define models that are similar to UML class diagrams and generate code from them. It is often used as the basis for Java-based model driven development frameworks, for example for tools in USIXML [17], in the model synchronisation framework proposed by Vogel et al. [114] or in the framework for model driven visualization developed by Bull [115]. EMF meta models follow the Ecore standard. Ecore is the meta meta model of the Eclipse Modeling Framework. It is similar to the eMOF standard [116] defined by the Object Management Group. Thus, EMF models are similar to UML class diagrams.

EMF models can be represented as attributed typed graphs. The Henshin plugin [113] makes use of this fact to implement graph transformation on EMF models. Henshin enables the definition of graph transformation productions based on Ecore meta models. These productions can be used to reconfigure EMF models. For the transformation, the Henshin tool converts the ecore meta-model into a type graph and the EMF model into a corresponding attributed typed graph. Henshin uses the implementation of graph transformation of the AGG tool [117].

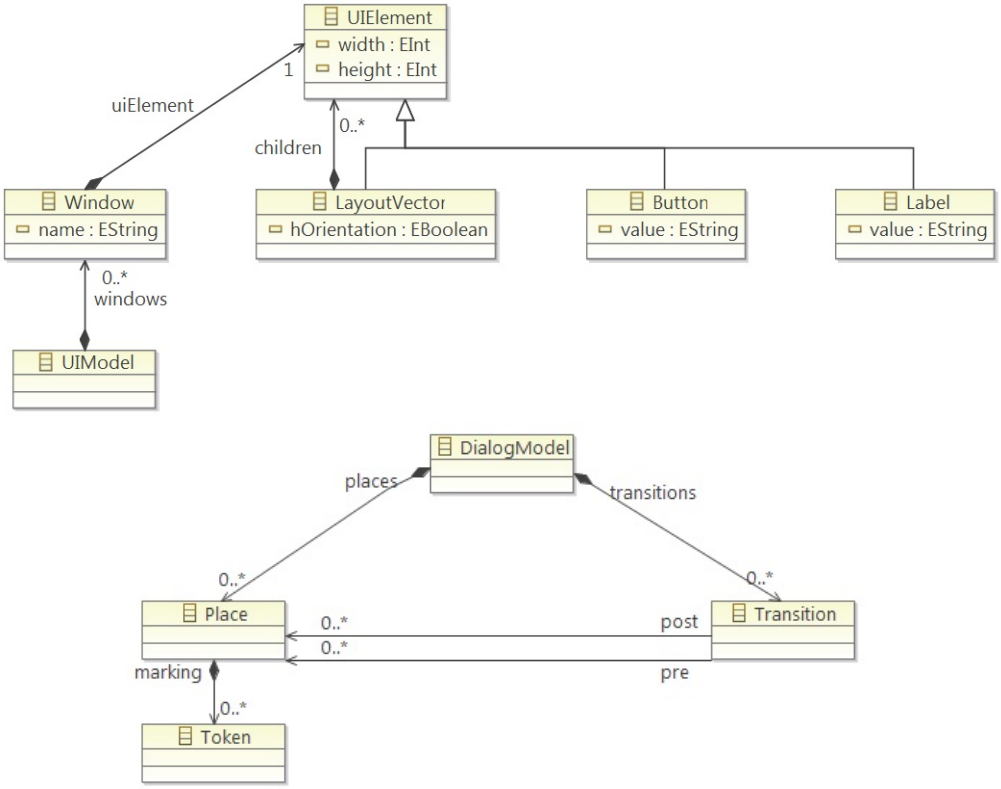


Figure 7.1.: The Ecore models for the UI model (top) and dialog model (bottom) from the running example.

Figure 7.1 shows an Ecore version of the type graphs for the two models in the running example. The UI model in the top half of the figure contains an additional Element *UIModel*. This element exists due to the necessity to have a containment hierarchy with a single containment root in EMF. All windows are contained in this element via the reference *windows*. The references *uiElement* and *children* are also containment references. The layout hierarchy of the window constitutes the containment hierarchy of the EMF model. The dialog model, shown in the bottom part of the figure, contains an element *DialogModel* which is the containment root. This root contains all places and transitions. The tokens are contained in the respective places.

Henshin offers the option to add nested application conditions to graph transformation productions. Although there is no direct support to model global nested conditions, we can formulate empty graph transformation productions with nested application conditions. This can be used to simulate global conditions. The current version of Henshin does not support all variations of conditions. In the editor it is not possible to use the elements $false$, $\forall$ and $\Rightarrow$. In addition, conjunction and disjunction are restricted to binary operators. Since nested conditions are defined based on $true$, $\wedge$, $\neg$ and $\exists$ and all other elements are abbreviations for patterns of these elements (cf. Definition 28) this

is not a restriction in power of expression. Similarly, a conjunction or disjunction with more than two elements can be simulated by converting it into a set of nested binary conjunctions or disjunctions.

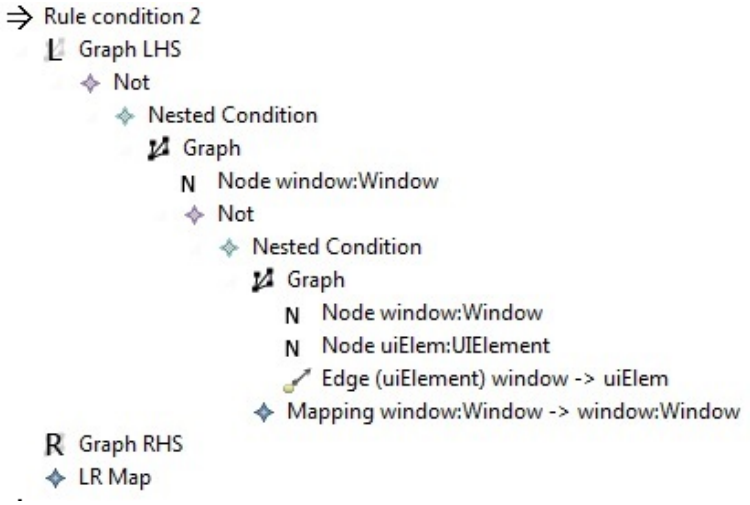


Figure 7.2.: Implementation of Condition 2 from the running example.

Figure 7.2 exemplifies a global condition in the Henshin model editor. The condition is condition 2 from the running example. The left and right hand side of the Henshin production are empty. Thus, the condition is interpreted as global condition. The condition itself requires every window to have a root element. Normally, this constraint is defined using an all-quantifier and one existence quantifier to state that for all windows a connected UI element exists. Since the all-quantifier is not available in the editor, the logical equivalent is used. The condition states that no window that is not connected to a UI element exists. The outer condition is a negation, followed by an existence quantifier that requires the existence of a window. Existence quantifiers are called *NestedCondition* in Henshin. The inner nested condition is also a negation with an existence quantifier that requires the window to be connected to a UI element.

The relation between Ecore and attributed typed graphs is explored in [118]. It is always possible to represent an EMF model as an attributed typed graph. Most of the restrictions of the Ecore model (the meta model for this model) can be represented by an attributed type graph and a set of nested conditions. The attributed type graph has the structure of the Ecore model. The additional conditions are used to describe constraints that are specified in the Ecore model but cannot be solely expressed by a type graph. An example is the annotation of multiplicities, which cannot be expressed in a type graph but can be formulated as nested conditions. However, one aspect of the Ecore model cannot be expressed this way. EMF requires its models to specify a containment hierarchy where all nodes are contained recursively in one containment root. Edges can be marked as containment edges that denote that the target element is contained in the source element of the edge. EMF requires that every node in a model

is contained in the root element and is only contained via one path. Attributed typed graphs are not able to mark containment edges. They cannot represent a containment hierarchy. Thus, although it is always possible to transform an Ecore model into a type graph, together with a set of nested conditions, and a conforming EMF model into an attributed typed graph that is typed correctly and fulfils the conditions, it is not in general possible to transform a correctly typed attributed typed graph into a correct EMF model with correct containment hierarchy.

Biermann et al. extend attributed typed graphs with containment edges to counter this problem [118]. Containment edges could be integrated into graph diagrams but would only serve to cater to one peculiarity of EMF. For this reason we decided not to add containment edges to the theory in this thesis. However, if the extension of attributed typed graphs with containment edges does preserve the $\mathcal{M}$-Adhesive properties, it is also possible to construct a functor category from them and apply the Trollmann approach.

One particular problem of containment edges is that graph transformation rules can have side effects that are not contained in the production itself. For example, an element could be deleted by EMF due to containment because a containment edge is deleted by a graph transformation production, although the object is originally preserved by the rule. This causes problems for conflict analysis as not all side effects of the transformation rule are captured by the graph transformation production and thus there can be changes which cannot be taken into account in an analysis based on this production.

To counter this problem we implemented a closure of transformation rules under containment. This closure analyses a production application for side effects and manipulates the original graph transformation production to correctly capture these side effects. Side effects occur in the following cases:

- *Deletion of containing node:* Whenever a node is deleted in EMF, the subtree of its contained children is also removed from the model. Thus, the closure needs to enhance the original production to reflect all deleted children that are not reflected in the production.
- *Deletion of containment edge:* Similar to containing nodes, the deletion of containment edges results in the deletion of all children in the containment subtree.
- *Creation of new containment edges:* If a new containment edge is added to an element that is already contained elsewhere in the model, the old containment edge is removed.
- *Addition of non-contained elements:* If an element is added but is not contained correctly, this element and all references to it are deleted due to containment.

The closure algorithm implements the following steps:

1. *Calculate falsely added elements:* calculate all elements that are added by the production but not inserted in the containment hierarchy. This also applies to the children of these elements.

2. *Remove falsely added elements:* remove the falsely added elements from the right hand side of the production application.
3. *Calculate captured deleted elements:* find all elements that are deleted by the production application. These are all elements that are referenced from its left hand side and not contained in its right hand side.
4. *Calculate uncaptured deleted elements:* calculate closure due to removed containing nodes, containment edges and multiple containment edges. Take into account that elements whose containment edge is deleted can be inserted into the model via another containment edge.
5. *Capture uncaptured deleted elements:* add all deletions that are not yet reflected in the production application. If the elements are not matched add them to the left hand side and match them correctly. If they are already matched but preserved remove them from the right hand side.

The closure is a production application that explicitly contain all changes due to containment that are derived from the original production application. The closure of a Henshin rule correctly reflects all elements that are deleted during the application. On the other hand it requires more nodes than the original production as its left hand side is enhanced with the additionally deleted elements. Thus, the closure is suited for finding the deleted nodes of the production and the original rule is suited for finding the required nodes. Thus, both versions of the rule are used for the detection of adaptation-adaptation conflicts.

The following relations of EMF models and the formalism in the Trollmann approach can be summarized on the level of attributed graphs:

- *Model:* An EMF model can be represented as an attributed graph.
- *Meta Model:* An EMF ecore model can be used to generate an attributed type graph and a set of nested conditions.
- *Adaptation Formalism:* Henshin productions on EMF models can be used to generate AGG productions in attributed graphs. Additional deletions can be integrated into the analysis by calculating the closure of the productions.
- *Consistency Requirement:* A nested condition can be represented by a Henshin production with empty left and right hand side.

The tested version of Henshin seems to have one bug regarding injective matching. Although, non-injective matching can be specified, the morphism between nesting levels in nested conditions are injective. Some of the conditions from the running example require such non-injective morphisms. These conditions are evaluated wrongly by henshin. However, since the implementation of our algorithm *conditions* does not use the evaluation of conditions, it yields correct results.

The concepts in EMF are not directly related to Graph Diagrams. The next section describes how graph diagrams can be integrated into EMF and how this integration relates to the original formalism.

7.2. Integration of Graph Diagrams

As described in the last section EMF is closely related to attributed typed graphs. However, EMF handles model relations in a way that is conceptually different from graph diagrams. EMF enables Ecore models to import and use elements from other Ecore models. This enables references to elements that conform to a different meta-model. In this approach the models are very closely linked, similar to the approach of merging multiple attributed graphs into one. This can be used to simulate graph diagrams for the purpose of the case study.

As briefly described in Section 6.4.1 relations can be handled as explicit nodes in the graph diagram. This enables the representation of partial relations between more than two models. The running example is structured this way. The implementation of the running example also uses this manner of handling relations. In the implementation each of the nodes is represented as an EMF model conforming to a separate Ecore meta model. The implementation restricts the Ecore meta models representing model nodes to be closed (without edges to other models). The Ecore meta models for relations import these models and reference them.

The Ecore meta models for the UI and dialog model from the running example have already been given in Figure 7.1. In this section we introduce a relation node for establishing the relation between these models. We simulate morphisms between the relation node and the models by references in the model. We make further assumptions to simulate these morphisms. Morphisms are left-total, meaning that every element in the source model is mapped to an element in the target model. Thus, we need a reference to one element in each related model for every node in the relation model. In addition, morphism are right-unique, meaning every element of the source graph is mapped to exactly one element of the target graph. To achieve this we annotate these references in the meta-model with multiplicity of one, requiring exactly one such reference for each element of that type.

Figure 7.3 shows the Ecore meta model of the relation between the two models in the running example. The model contains a containment root *Dialog2UIRelation* which contains everything else. This containment root is an exception from the requirement to be related to elements in the source and target model. The two relation elements *Window2Place* and *UI2Transition* are each connected to one element from the dialog and UI model with multiplicity 1 as required.

This way of representing models and relations can be extended to arbitrary schemes. Models are represented as closed-off entities and morphisms are represented by attributes in the elements of the source model.

These relation references are sufficient to express the requirements of the running example and in the proof of concept implementation in Chapter 8. However, it should be mentioned that this construction does not map all of the properties of attributed graph morphisms. A morphism also maps between the edges of the source and target object. In attributed typed graphs these are normal graph edges, node attribute edges and edge attribute edges. Since EMF models do not have edge attributes this component can be ignored. However, the mapping of graph edges and node attribute edges needs

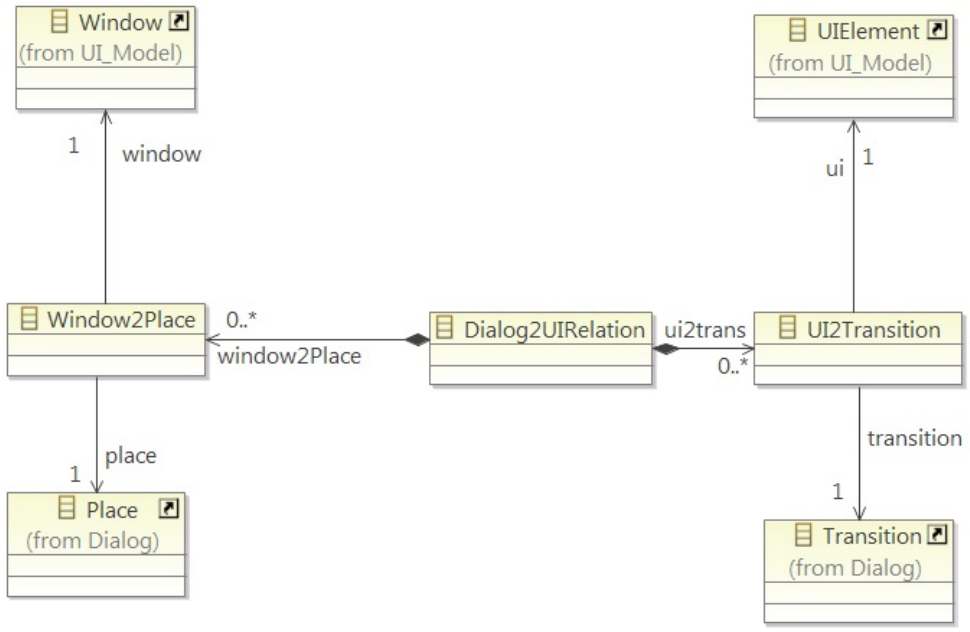


Figure 7.3.: The Ecore model for the relation Dialog 2 UI from the running example.

to be regarded. Node attribute edges are translated to attributes in EMF models. The requirement to preserve node attribute edges means that for each attribute in a node in the source EMF model there needs to be an attribute in the mapped node in the target EMF model with the same values. Both attributes do not need to have the same name. This requirement can be formulated as a nested condition on EMF models.

Mapping graph edges directly is not as easy as there can be multiple edges of the same type between the same nodes in the target EMF model and no way to distinguish which of these is the mapped one. Edges could be modelled as nodes which has one source and one target object to represent them explicitly. In this case the node mapping can apply and uniquely identify the mapped edge. Source and target compatibility for the edge can be formulated as nested conditions. As stated before these compatibility requirements are not required in the running example as the relation node does not contain and attributes or edges. However, they can be implemented if needed. It is also possible to regard the whole EMF model as one big ATG and apply the analysis based on the assumption that there is only one model in the diagram.

We use one additional Ecore model to represent the whole diagram in one Ecore file and have a convenient containment root. The role of this model is to reference all models and relations in one diagram and to serve as containment root for the whole system. The containment root of the running example is depicted in Figure 7.4.

In general the diagram is still one EMF model with several sub-models that are arranged in a certain way. Thus, the Henshin plugin can still be utilized for describing graph transformation and nested conditions on these models. Condition 2, as described in the previous section, is thus also a valid example of a condition. It concerns only

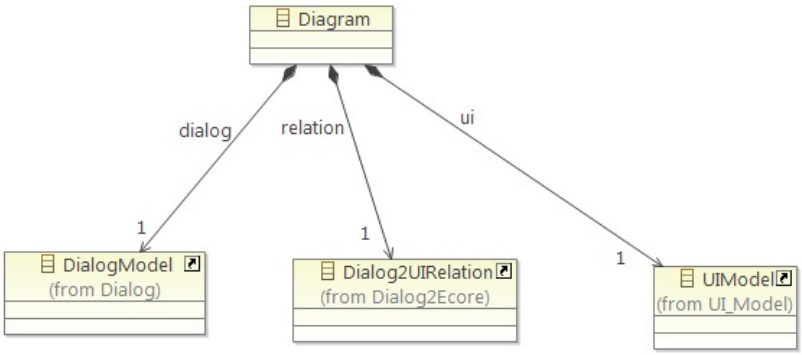


Figure 7.4.: The Ecore model containing the root of the diagram from the running example.

one model node and all elements in the condition are from the same model. Although, the whole diagram is one big EMF model anyway, the contained models and relations can be distinguished. This can be done by checking to which Ecore meta model they conform by checking which EPackage their type is contained in. This check enables the distinction between elements in different models.

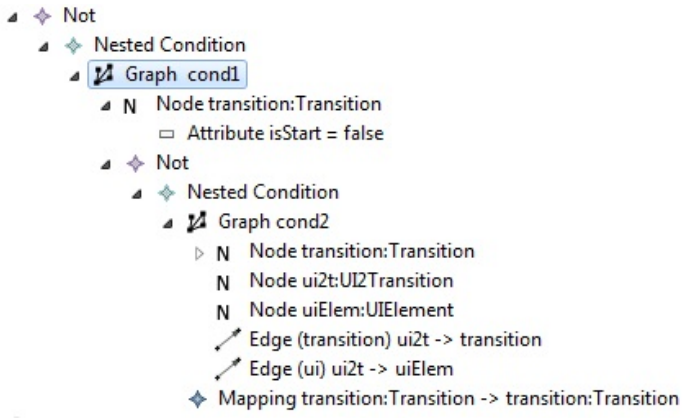


Figure 7.5.: Implementation of Condition 5 from the running example.

Figure 7.5 shows a second condition that spans multiple models. The condition is condition 5 from the running example. It requires each transition with $isStart = false$ to be connected to a UI element via a relation node of type *UI2Transition*. The condition has been implemented using negation and existence as a substitute for the all-quantified constraint. The condition spans two models and one relation.

Graph transformation productions on graph diagrams are also implemented using the Henshin plugin. Figure 7.6 shows an example for such a production. The production adds a back button to a window. The figure shows the production in the Henshin diagram editor. This editor merges the left hand side, interface and right hand side into

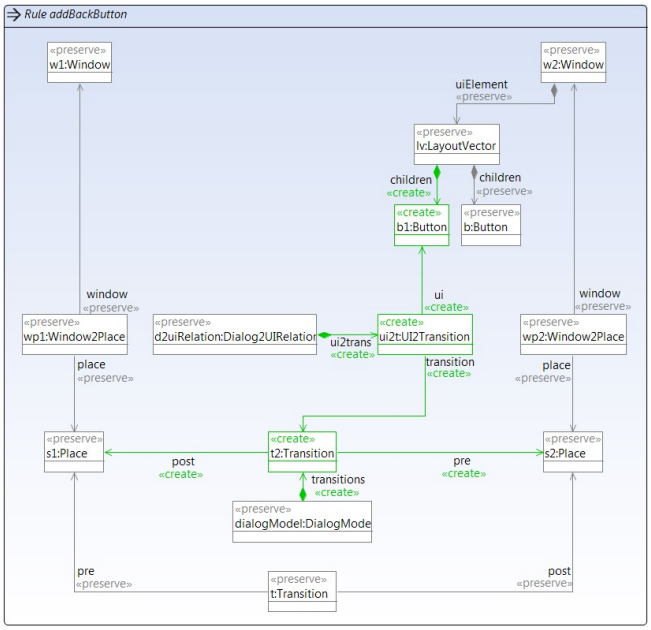


Figure 7.6.: The rule for adding a back button from the running example.

one view and indicates the location of each node by annotating them with the keywords $<<$ *preserve* $>>$,$<<$ *delete* $>>$ and $<<$ *create* $>>$. $<<$ *preserve* $>>$ denotes that an element is located in the left hand side, right hand side and interface of the production and thus is preserved by the production. The other keywords indicate that an element is either only in the left hand side ($<<$ *delete* $>>$) or the right hand side ($<<$ *create* $>>$). The borders between the models are not shown in the Henshin diagram editor. We added them manually for illustration.

The production in Figure 7.6 requires two windows $w1$ and $w2$ that are connected to two places $s1$ and $s2$ in the dialog model. These places are connected via a transition t. In the layout of window $w2$ a button is added that is associated to a transition $t2$ which is added to the dialog model and enables the return from the place $s2$ to the first place $s1$. This production is an example of a production that manipulates multiple models at the same time. It changes the dialog and UI model as well as the relation between them.

Figure 7.7 contains the second production from the running example. This production merges two windows $w1$ and $w2$. It removes the window $w2$ and the associated place. Only the label of this window is preserved. The production adds this label to the first window. Additionally, the dialog model is reconnected to bypass the removed place. The rule connects the button $b1$ from the first window to transition $t3$.

Summarizing, in this section we established the following relations between EMF models and graph diagrams:

- *Model:* Multiple EMF models that are combined in one big diagram model. References between models are solely established by morphisms, which are represented

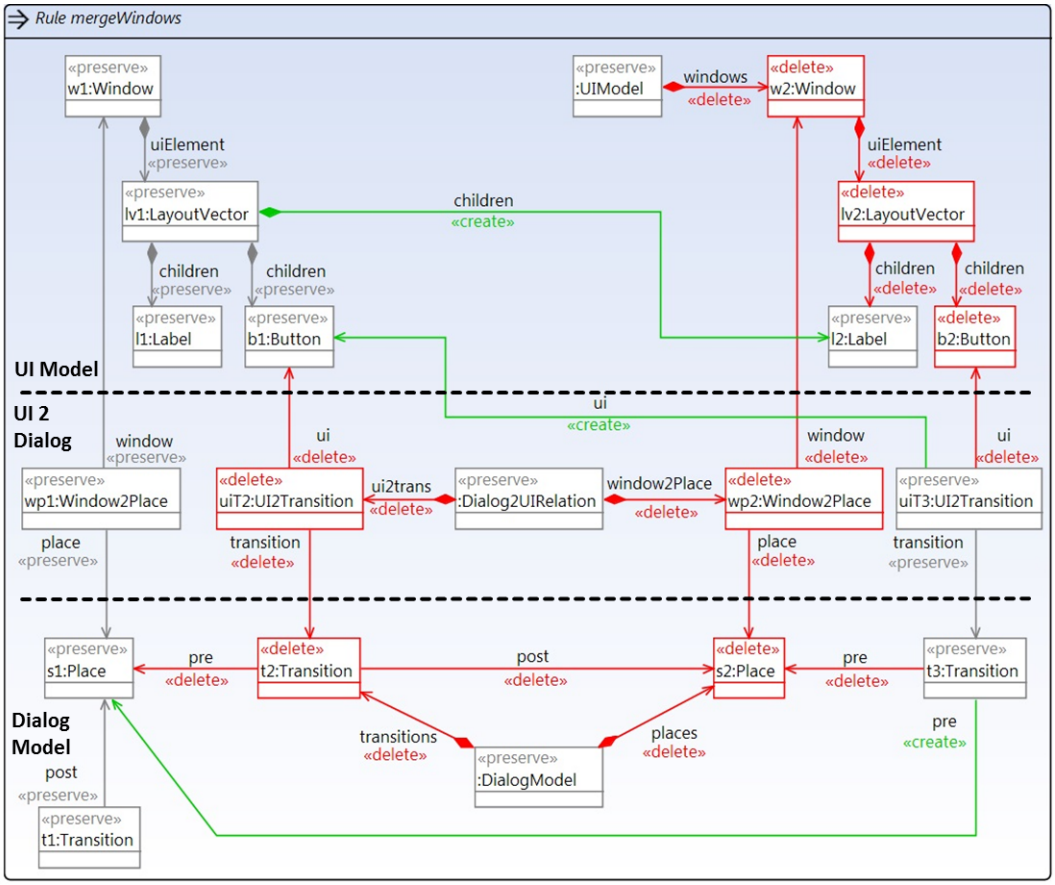


Figure 7.7.: The rule for merging two windows from the running example.

by a set of attributes and conditions.

- *Meta Model:* Ecore meta models that follow the conditions for morphisms given in this section and empty Henshin transformation rules with nested application conditions to define structural conditions.
- *Adaptation Formalism:* Henshin rules.
- *Consistency Conditions:* Empty Henshin rules with nested application conditions.

In the next section we describe our implementation.

7.3. Conflict Detection

In this section we describe the implementation of the conflict detection. As a basis for this implementation we had to incorporate some minor changes into the Henshin plugin. The relevant classes from the Henshin plugin are described in Section 7.3.1.

In the following sections we describe the main classes of our implementation. We implemented the conflict detection as an additional module that imports the Henshin models as base classes for the implementation and the Henshin interpreter to use its implementation of the execution of graph transformation productions and match calculation. The main functionality is provided to the programmer via five static classes. The class *HenshinTools.java* (cf. Section 7.3.2) contains general utility methods for loading and printing productions. The class *ConditionTools.java* (cf. Section 7.3.3) contains utility functions for loading and managing conditions. The class *maximumGraphTools.java* (cf. Section 7.3.4) provides utility functions for deriving a maximum graph. The conflict detection is implemented in the class *ConflictAnalysisTools.java* (cf. Section 7.3.5). The class *ReversionTools.java* (cf. Section 7.3.6) enables the maintenance of reversible production applications. We provide a code example of an analysis in Section 7.3.7.

7.3.1. The Henshin Plugin

Version 0.9.1 of the Henshin plugin serves as the basis for the implementation. For the purpose of this thesis we obtained access to the Java Code in the Henshin SVN repository. The implementation is based on Revision 1588. The conflict detection imports two modules from the Henshin Project: the Models (org.eclipse.emf.henshin.model) and the Interpreter (org.eclipse.emf.henshin.interpreter). The models are needed for their data structures. The Interpreter module contains implementations of graph transformation, match finding and condition checking that are used during conflict analysis.

The most important classes in the Henshin plugin are depicted in Figure 7.8. A *Rule Application* describes a graph transformation production that is matched to a graph and is the Henshin equivalent to a production application. It contains the production of type *Rule* as well as a match of type *Match*. The match maps each node in the left hand side to an *EObject* (an element of the model the production is applied to). The mappings of attributes and edges are derived from the object mapping. After the rule application is executed it also contains a result match, which describes the comatch of the production (the match from the right hand side to the result of the transformation). Each production contains two graphs of type *Graph* that denote the left and right hand side of the production. The interface between both is implicitly established by the *Mapping* of elements between left and right hand side. All nodes on the left hand side that are not mapped are deleted during the productions execution. All nodes on the right hand side that are not mapped are created during the productions execution.

Each graph may contain a condition of type *Formula*. Formulas in the left hand side are interpreted as left application conditions, formulas in the right hand side as right application conditions. The formulas in Henshin are slightly different from traditional nested conditions. They do contain existence quantifiers of class *NestedCondition*, the unary condition *Not* and binary conditions *And* and *Or*. However, they do not contain an all-quantified condition. Instead, they contain an exclusive disjunction of type *XOR*.

Before their interpretation the conditions of type *Condition* are converted to conditions of type *IFormula* by Henshin. Figure 7.9 contains this type and relevant sub-types. Corresponding to the original conditions, there are the subtypes *AndFormula*, *OrFormula*,

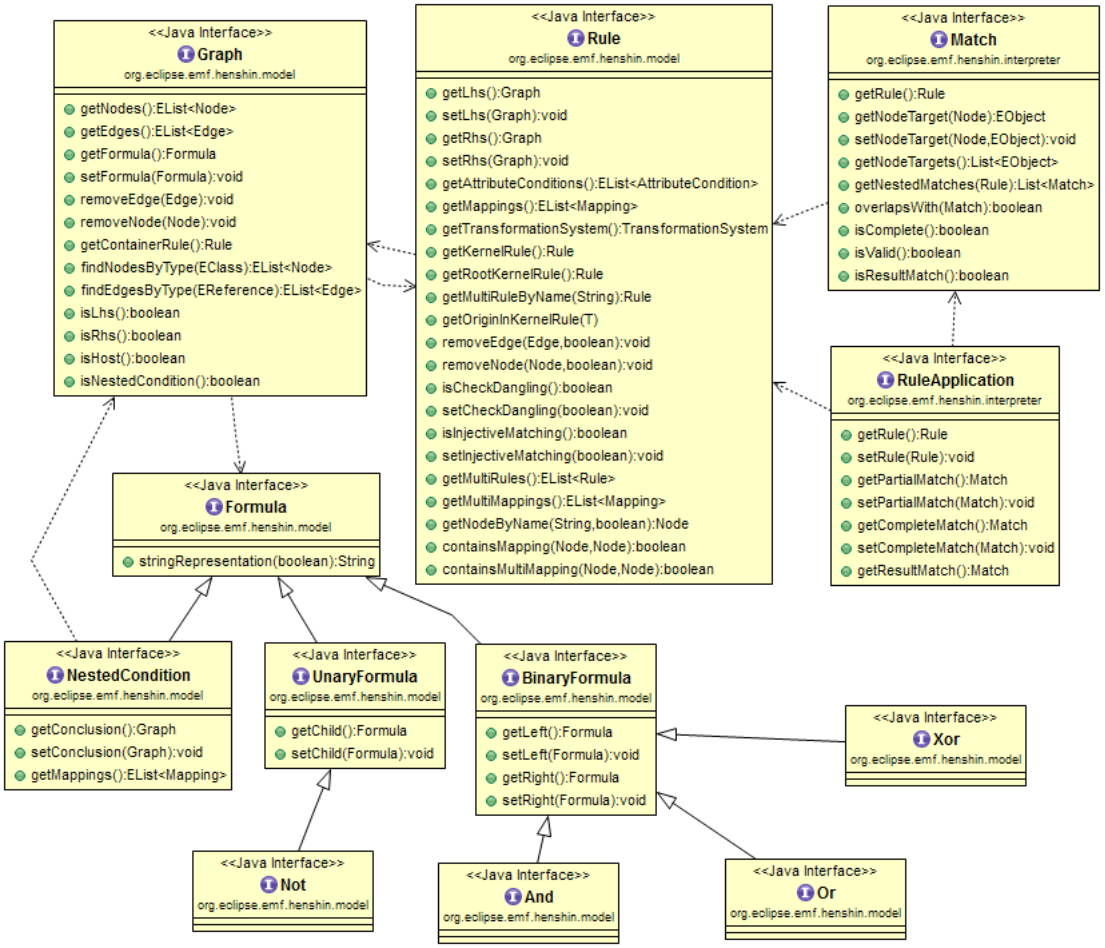


Figure 7.8.: Main classes of the Henshin plugin.

XorFormula and *NotFormula*. The existence quantifier is represented by an element of type *ApplicationCondition*. The original Henshin implementation does not contain an all-quantified condition. While formulating application conditions for productions they are simulated by the equivalence $\forall(A) = \neg\exists(\neg A)$.

Several smaller changes have been implemented in the Henshin code. These changes enable access to attributes that are hidden in the standard implementation. Their visibility has been changed to enable the conflict detection to read these attributes. We either converted them to public attributes or added getter and setter methods.

All other implementations are outside of the Henshin code in a dedicated module. The next Sections describe the main classes.

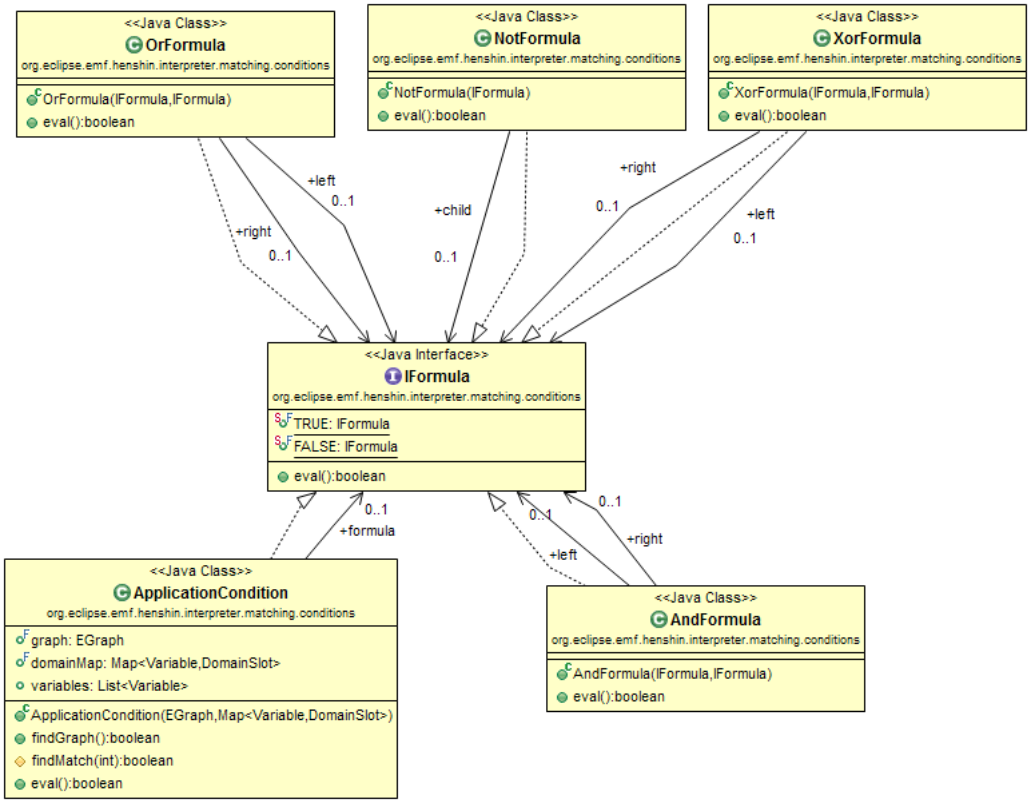


Figure 7.9.: The IFormula implementation of Henshin.

7.3.2. HenshinTools.java

The class HenshinTools is not directly related to conflict detection. However, since the Henshin plugin is designed to be used to do analysis within the graphical eclipse environment it lacks utility methods for using it programmatically at run time. The class HenshinTools implements some essential utility methods for this purpose.

A class diagram of the class HenshinTools and its methods is shown in Figure 7.10. The method *init* initialize EMF environment variables that need to be set up for the Henshin plugin to work. All other methods can be divided into two groups. The first group contains methods that simplify loading and creating objects, the second one contains methods that can be used to print a console representation of objects for debug purposes.

The HenshinTools contain methods for loading an existing model (*loadModel*) and graph transformation productions (*getRule*) from files. The method *createGraph* creates a graph from an EMF object that the productions can be applied to. The two versions of the method *createRuleAplication* create a rule application object from a production by finding a match to a graph. The method can be provided with an engine (an implementation of graph transformation). If not, it creates its own engine. This implementation selects the first match it finds. The developer is able to change the match from Java

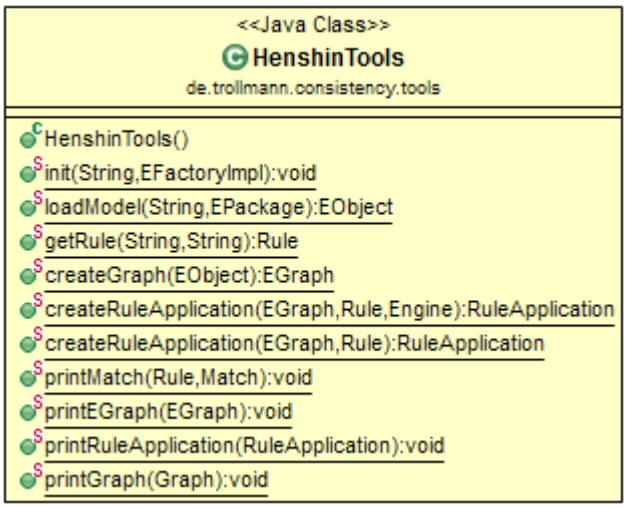


Figure 7.10.: UML representation of the class HenshinTools.

code in case a different one is required.

For debugging purposes the class HenshinTools contains methods for printing a textual representation of an EGraph (*printEGraph*), which describes the current state of the model. In addition, it enables to print objects of type Graph using the method *printGraph*. These objects describe the left and right hand side of productions. Printing a rule application and a match is also possible, using the methods *printRuleApplication* and *printMatch*.

7.3.3. ConditionTools.java

Similar to the class HenshinTools, the class ConditionTools.java contains several static utility functions. The developer can use these functions to create, debug and handle conditions. Figure 7.11 contains the class diagram of the utility class ConditionTools.

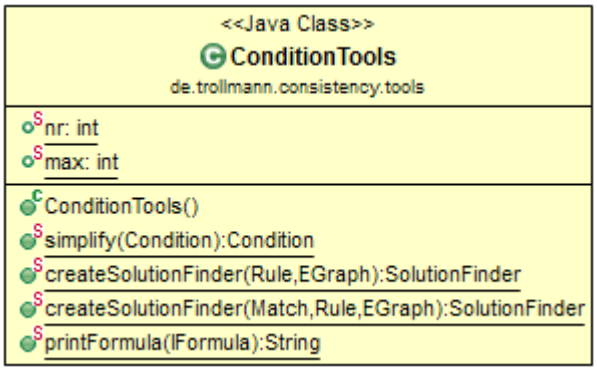


Figure 7.11.: UML representation of the class ConditionTools.

Both versions of the method *createSolutionFinder* create a class *SolutionFinder* for a nested condition that is contained in a graph transformation production. The method converts the nested condition into an *IFormula* and creates an object of type *SolutionFinder*. This object contains information for finding solutions based on an EGraph that represents the current structure of the model. The second version of the method can be provided with a partial match to restrict the scope of the condition.

The method *printFormula* prints a textual representation of a nested condition. It iterates over all inner conditions and calculates the statistics of the formula. The contained graphs can be printed via the respective methods.

The result of the algorithm for fulfilment reasons is of type *Condition*. This type represents a fulfilment condition for a nested condition and is conform to Definition 20. Usually, the direct output of the algorithms can be simplified. The convenience method *simplify* can be used for this purpose. This method operates on Boolean equivalences. Trivial and non-fulfillable conjunctions and disjunctions, for example, are resolved to true or false. The method simplifies the whole condition recursively.

The next section describes tools for calculating the maximum graph as a foundation for analysis of adaptation-consistency conflicts.

7.3.4. MaximumGraphTools.java

The maximum graph is the basis for the analysis of adaptation-consistency conflicts. It enables an estimation of potential morphisms as a basis for the generation of fulfilment conditions in Definition 22. Several classes are involved in calculating and representing the maximum graph in the implementation. They are shown in Figure 7.12.

The utility class MaximumGraphTooles provides the static method *CreateMaximumGraph* that can be used to calculate a maximum graph from a list of rule applications, a graph they are applied to and an engine that is used for transformation purposes. This method implements Definition 18. It first creates a set of maximum rule applications from the provided rule applications via the private helper method *createMaximumRuleApplication*. This helper method creates a copy of the rule. It then deletes all elements on the left hand side of this copy that are not mapped to the right hand side. Accordingly, all deletions are removed from the production. The remaining nodes are matched to the graph as in the original rule application. The private helper method *copyMatch* is used for this step.

The maximum graph is describes as an object of type *MaximumGraphInfo*. This object contains the maximum graph represented as an EGraph. Several of the analysis methods require additional information. The object *MaximumGraphInfo* keeps track of this information during the creation of the maximum graph. The original model is not changed directly. The maximum graph is created based on a copy. Accordingly, the nodes in the maximum graph are copied nodes that cannot be directly compared using the method *equals*. The maximum graph info holds a map *originalToCopiedNodeMap* that relates the nodes in the original model to the nodes in the maximum graph to represent the relation between the original and copied nodes. This map represents the inclusion morphism from the original model to the maximum graph. The maximum

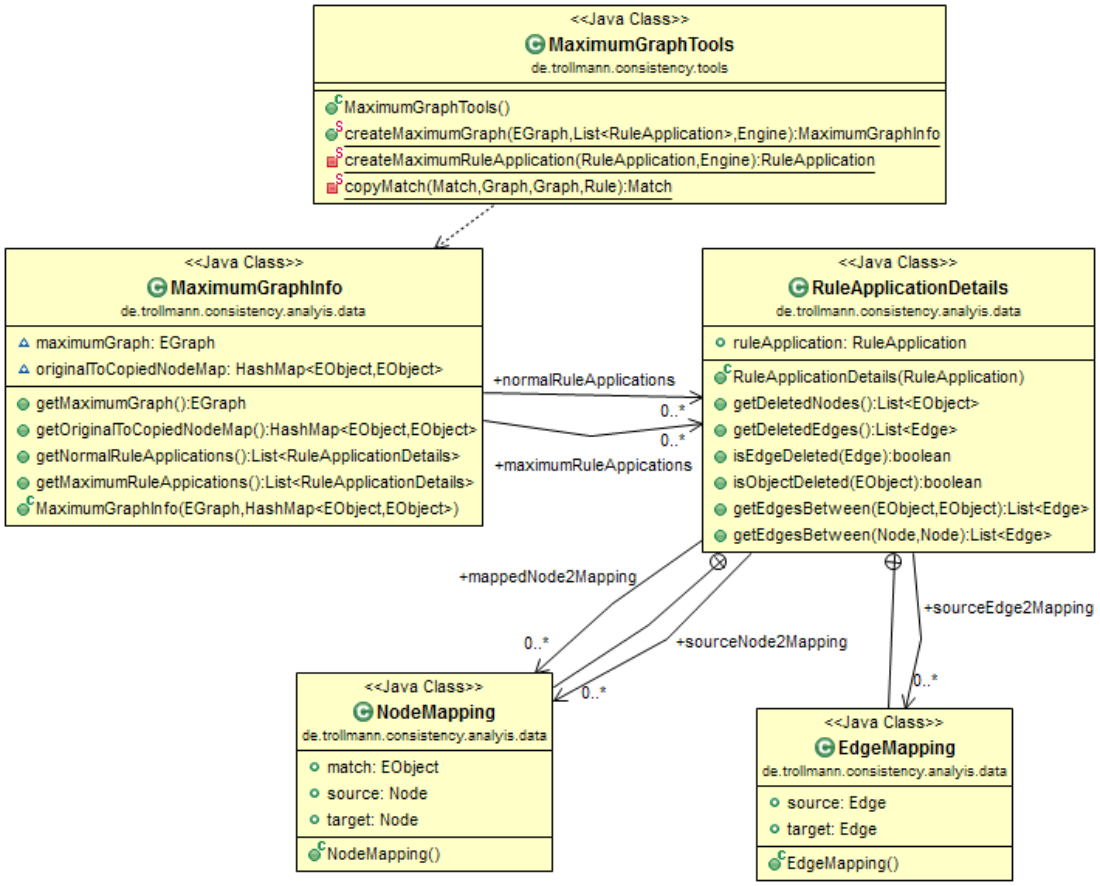


Figure 7.12.: UML representation of the class MaximumGraphTools.

graph info also holds the original rule applications and the maximum rule applications that have been created from them. They can be used to reason about the creation and deletion of objects in the maximum graph.

Each of the rule applications and maximum rule applications are wrapped into an object of class *RuleApplicationDetails* to ease reasoning about them. On Creation this object extracts objects of types *NodeMapping* and *EdgeMapping* from the left hand side and right hand side of the rule application. These objects represent the matches into the graph and between the left and right hand side to make them easily accessible. The class *RuleApplicationDetails* uses these objects to implement convenience methods *isObjectDeleted*, *isEdgeDeleted*, *getDeletedNodes* and *getDeletedEdges* which can be used to check for deleted and added elements. The deleted elements are stored for more efficient access later once they are calculated. The *RuleApplicationDetails* also provide methods *getEdgesBetween* for retrieving edges between two nodes in the left hand side of the production or in the maximum graph.

These utility functions are used in the class ConflictAnalysisTools.java to calculate condition conflicts. This tool class is described in the next section.

7.3.5. ConflictAnalysisTools.java

The class *ConflictAnalysisTools.java* contains the main analysis methods. In this section we describe the methods of this class as well as the data type that is used to describe the results. An UML class diagram of this class is shown in Figure 7.13.

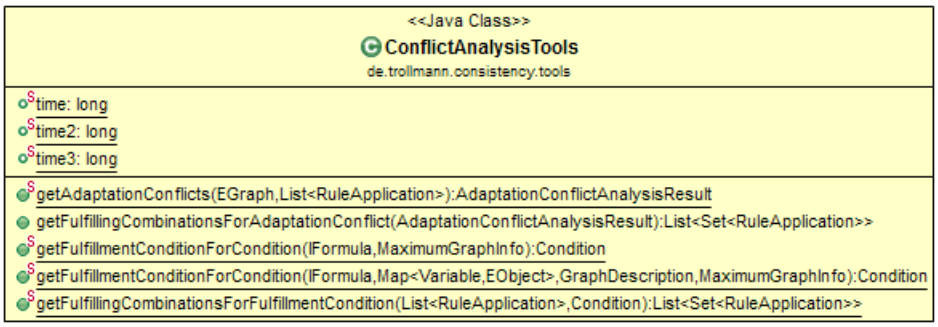


Figure 7.13.: UML representation of the class ConflictAnalysisTools.

The conflict analysis tools contain four methods. The method *getAdaptationConflicts* executes an analysis for adaptation-adaptation conflicts. This method returns an object of type *AdaptationConflictAnalysisResult*. This class is shown as UML class diagram in Figure 7.14. It contains a set of node-, edge- and attribute conflicts. Each conflict is described by an element of type *Conflict*. This object describes the two conflicting rule applications and the conflicting element from the graph. The conflicting element may be a node, an edge or an attribute. The conflicting description denotes that the rule application *first* removes the conflicting element although the rule application *second* requires it to be applied.

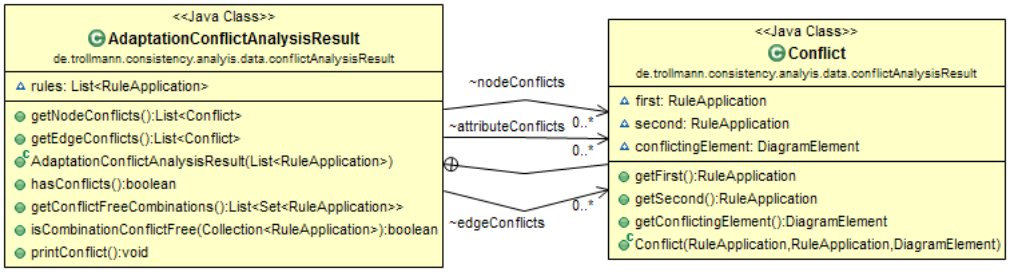


Figure 7.14.: UML representation of the result of the dependency analysis.

The method *getAdaptationConflicts* implements the algorithm presented as pseudo-code in Figure 6.24 from Section 6.5.3. The pseudo-code is implemented for nodes, edges and node attributes. EMF models do not contain edge attributes.

The method *getFulfillingCombinationsForAdaptationConflicts* in class *ConflictAnalysisTools* calculates conflict free subsets of adaptations for adaptation-adaptation conflicts. This method uses an *AdaptationConflictAnalysisResult* and its method *getCon-*

flictFreeCombinations. This method returns a list of combinations (sets of rule applications). Each conflict represents a pair of productions that cannot be applied at the same time. This method makes use of this fact to calculate the set of all possible combinations of parallel independent productions. To reduce the effort and the number of results the method only returns maximal sets. A set is maximal if no other production applications can be added to it without a conflict. Since the production applications in these sets are parallel independent all subsets of these sets are valid solutions.

The remaining methods enable an analysis of adaptation-consistency conflicts. The method class *getFulfillmentConditionForCondition* from the *ConflictAnalysisTools* implements this analysis. This method requires an $IFormula$, describing the analysed condition, and a $MaximumGraphInfo$, describing the maximum graph. The maximum graph can be retrieved using *MaximumGraphTools.createMaximumGraph(...)*. The method *getFulfillmentConditionForCondition* implements the analysis algorithm for fulfilment conditions defined in Definition 22.

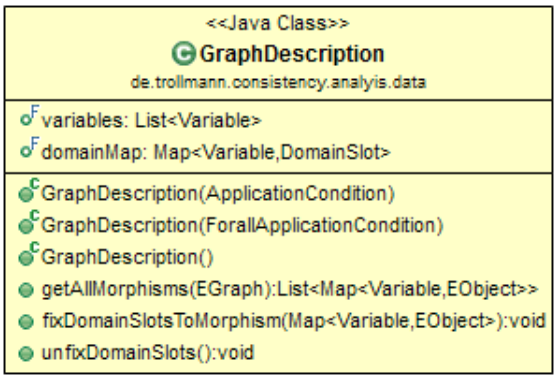


Figure 7.15.: UML representation of a graph description.

The method also exists in a second version that is used for iterating over the condition structure recursively. This method uses an additional $GraphDescription$ and a Map describing the morphism the condition is currently evaluated with. The class diagram of the graph description is shown in Figure 7.15. This class is a helper class that can be generated from an application condition. It describes the graph that is the basis for this condition and provides methods for keeping track of the morphisms in the condition. The method *getAllMorphisms* calculates all morphisms to another graph. In Henshin matchable graphs are described as variables and their matches are described as domain slots. These can be locked to fix a match. The analysis of an existence quantifier requires fixing a found morphism and then analysing the inner condition. For this purpose the methods *fixDomainSlotsToMorphism* and *unfixDomainSlots* can be used. Calling *fixDomainSlotsToMorphism* means fixing a morphism p of the current condition such that $getAllMorphisms$ can be used to find all morphisms q that commute with p. In a further recursion q can be fixed using the same method and a morphism for a further inner condition can be found. The method *unfixDomainSlots* is used after the recursion to free the fixed variables for a morphism.

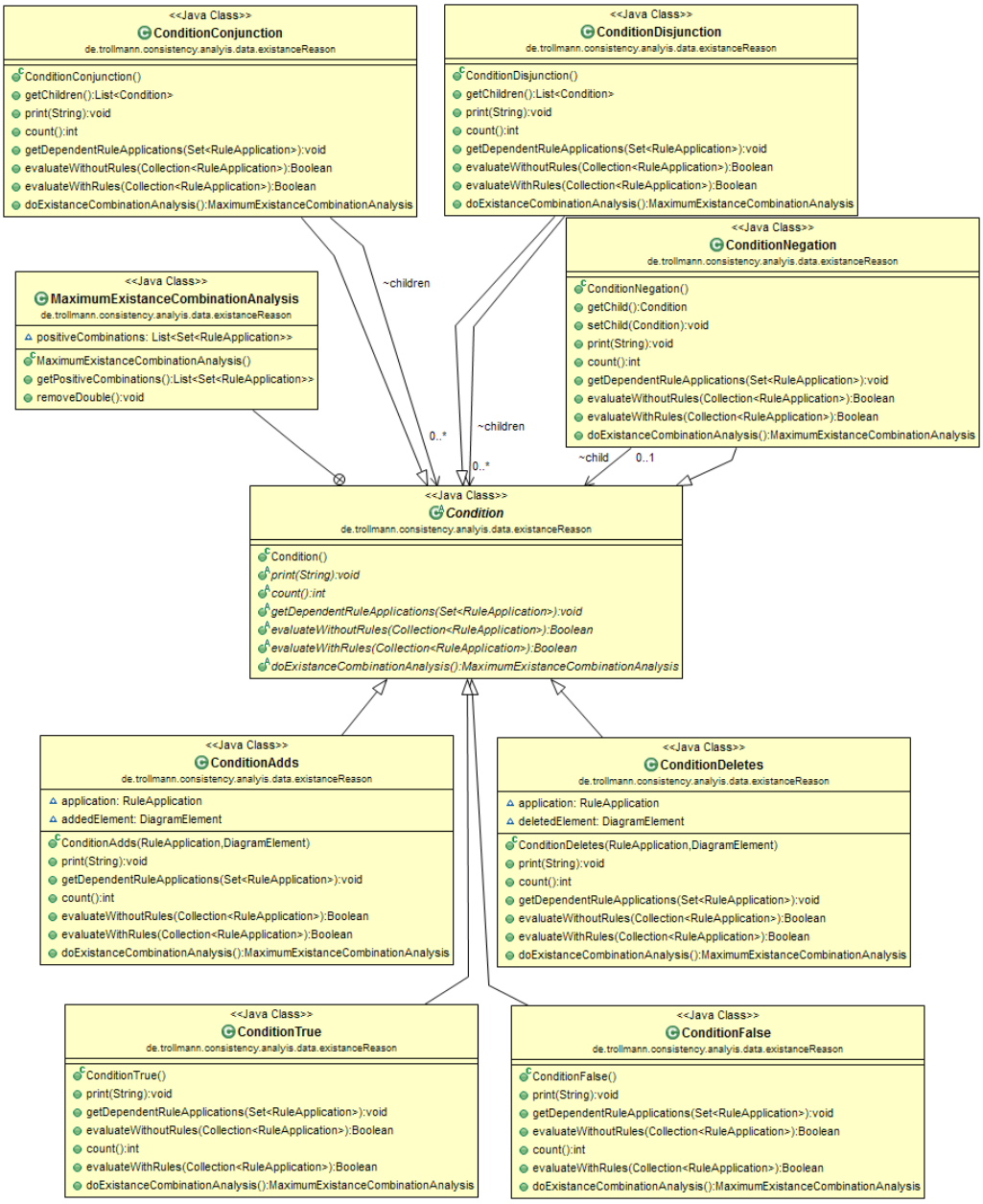


Figure 7.16.: UML representation of a fulfilment condition.

The analysis of condition conflicts yields a result of type *Condition*. This type is a representation of the notation for fulfilment conditions described in Definition 20. A UML representation of the respective classes is depicted in Figure 7.16. It is a direct representation of these conditions. The statements *Adds* and *Removes* are represented by *ConditionAdds* and *ConditionDeletes*. *true* and $false$ are represented by *Condition-*

True and *ConditionFalse*. Conjunction, disjunction and negation of fulfilment conditions is represented by *ConditionConjunction*, *ConditionDisjunction* and *ConditionNegation*.

The method *getFulfillingCombinationsForFulfillmentCondition* in the *ConditionAnalysisTools* calculates all subsets of production applications that fulfil the fulfilment condition. For optimization purposes this method uses the method *doExistanceCombinationAnalysis* of the condition. This analysis yields an object of type *MaximumExistanceCombinationAnalysis*, which enables the derivation of fulfilling combinations.

In the next section we describe how the *ReversionTools* can be used to calculate and maintain production applications for revering applied adaptations.

7.3.6. ReversionTools.java

A class diagram of the class *ReversionTools.java* is shown in Figure 7.17. The methods in this class implement the algorithms in Definition 23.

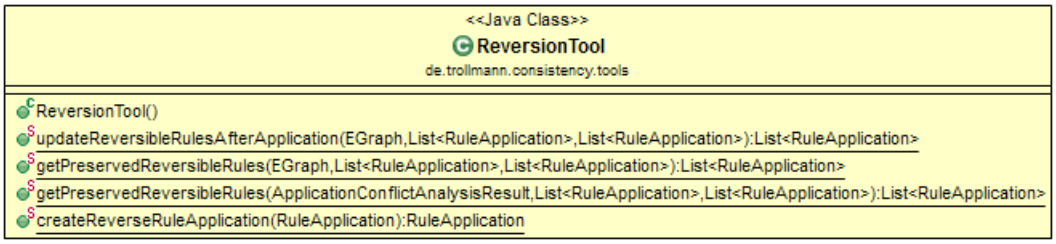


Figure 7.17.: UML representation of the class ReversionTools.

The class contains four methods. The method *createReverseRuleApplication* creates a reverse rule application. This method needs a comatch morphism. Accordingly, it assumes that the rule application has already been applied.

The two methods *getPreservedReversibleRules* can be used to determine which reverse rule applications can still be applied after a set of rule applications is applied. The preservation of reversible rule applications requires parallel independence. It uses analysis methods for adaptation-adaptation conflicts from class *ConflictAnalysisTools* for this purpose. In case this analysis has already been executed based on the reverse rule applications the result of the *ApplicationConflictAnalysisResult* can be handed to this method to avoid recalculating these parallel dependencies.

In most cases only the method *updateReversibleRulesAfterApplication* needs to be called. This method filters the set of reversible rule applications using the method *getPreservedReversibleRules*. Afterwards, it creates the reverse rule applications for the newly applied rule applications using the method *createReverseRuleApplication* and adds them to the this set. The method returns the updated set of reverse rule applications.

7.3.7. Example Implementation

This section describes an example implementation that showcases how our implementation can be used for conflict detection.

```
434    private static List<RuleApplication> consistencyAnalysis(EGraph model,Engine engine, List<RuleApplication> toApply,
435                                                              List<RuleApplication> reverseApplications, Rule conditionRule) {
436
437        // base test on the current adaptations as well as reverse adaptations
438        List<RuleApplication> applications = new LinkedList<RuleApplication>();
439        applications.addAll(toApply);
440        applications.addAll(reverseApplications);
441
442        // test for adaptation adaptation conflicts:
443        AdaptationConflictAnalysisResult analysisResult = ConflictAnalysisTools.getAdaptationConflicts(model, applications);
444
445         //Adaptation Conflict Resolution
446        // TODO: a more sophisticated conflict resolution should go here!
447        List<RuleApplication> adaptationConflictFreeAdaptations = new LinkedList<RuleApplication>();
448        adaptationConflictFreeAdaptations.addAll(analysisResult.getConflictFreeCombinations().get(0));
449
450        // create maximum graph and initialize condition with respect to this maximum graph
451         MaximumGraphInfo maximumGraph = MaximumGraphTools.createMaximumGraph(model, applications, engine);
452         SolutionFinder sf = ConditionTools.createSolutionFinder(conditionRule, maximumGraph.getMaximumGraph());
453
454
455         // calculate the fulfillment condition
456         Condition reason = ConflictAnalysisTools.getFulfillmentConditionForCondition(sf.formula, maximumGraph);
457
458        // Consistency Conflict Resolution
459        // TODO: a more sophisticated conflict resolution should go here!
460        Condition simplified = ConditionTools.simplify(reason);
461        ConflictAnalysisTools.getFulfillingCombinationsForFulfillmentCondition(applications, simplified);
462        List<RuleApplication> adaptationsToApply = new LinkedList<RuleApplication>();
463        adaptationsToApply.addAll(ConflictAnalysisTools.getFulfillingCombinationsForFulfillmentCondition(applications, reason).get(0));
464
465        // apply rules
466        for (RuleApplication ruleApplication : adaptationsToApply) {
467            ruleApplication.execute(null);
468        }
469
470        // return updated set of rule applications
471        return ReversionTool.updateReversibleRulesAfterApplication(analysisResult, model, reverseApplications, adaptationsToApply);
472    }
```

Figure 7.18.: Example code for conflict resolution.

An example in Java notation is shown in Figure 7.18. The code snippet shows the method *consistencyAnalysis*. This method uses the following parameters:

- *model*: The current state of the model
- *engine*: The graph transformation engine that is used for transformation purposes
- *toApply*: The set of production applications to be applied
- *reverseApplications*: The current set of reverse production applications
- *conditionRule*: A production containing a nested condition that needs to be fulfilled

The method facilitates a conflict analysis on the production applications and applies a solution. As a result the *model* is updated and the new set of reverse production applications is returned.

In lines 438 to 440 the set of production applications to analyse is created. Both the production applications from *toApply* and from *reverseApplications* are taken into account in this set. The analysis for adaptation-adaptation conflicts is triggered in line 443. As a result an object of type $ApplicationConflictAnalysisResult$ is available and can be used for reasoning about possible combinations of production applications.

The conflict resolution in the example is rudimentary. The resolution takes the first entry of conflict free combinations returned by the method *getConflictFreeCombinations*. An alternative resolution mechanism could iterate over all combinations to find the optimal one with respect to some criteria or test potential combinations using the method $isCombinationConflictFree$.

As basis for analysis of adaptation-consistency conflicts the maximum graph is calculated (line 451) and the condition is initialized with the maximum graph (line 452). The analysis for adaptation-consistency conflicts is done in line 456. The resulting fulfilment condition is the basis for resolution of these conflicts. Again, the implementation is rudimentary and uses the method $getFulfillingCombinationsForFulfillmentCondition$ of class $ConflictAnalysisTools$. Alternatively, the method $evaluateWithRules$ can be executed on the fulfilment condition to test single subsets for conflict-freeness. During a real analysis the result of the analysis for nested conditions should also be compared with the combinations that are free of parallel dependencies.

The resulting combination is then applied (line 467) and the set of reverse production application is updated and returned in line 471.

7.4. Evaluation of the Running Example

We implemented the running example as a proof of concept evaluation and estimation of the performance of our algorithms. The implementation of the running example in EMF and Henshin has already been described in Section 7.2. Based on this implementation we implemented a test class *Test.java* to systematically build models, graph transformation productions and conditions and execute a consistency analysis on this basis.

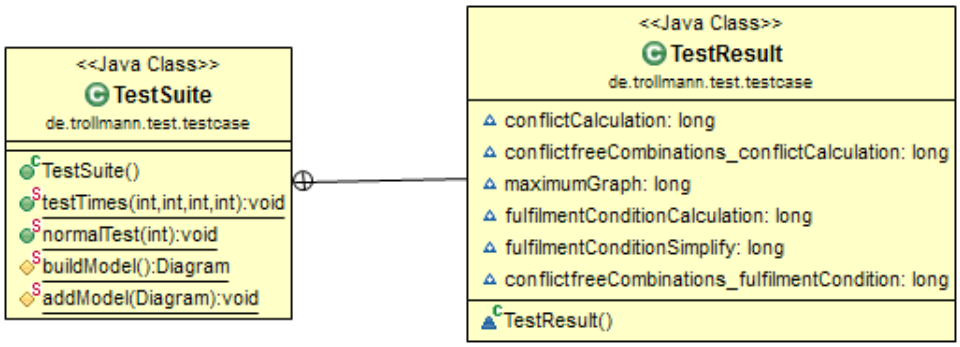


Figure 7.19.: The test class and data type.

The class *Test.java* is depicted in Figure 7.19. The methods *buildModel*, and *addModel* can be used to build models of different sizes for testing purposes. The method *normalTest* is a test method for testing different configurations of models and conditions with a given number of rules. This method is changed manually to test different configurations.

The method *testTimes* enables structured testing and the recording of execution times. It can be configured using four input parameters:

- *noRules*: the number of production applications to be applied
- *noTimes*: the number of test cases
- *modelSizeIndicator*: the indicator for model size

- *conditionSize*: the indicator for condition size

To test the performance of the implementation under different conditions the number of rules, the size of the model and the size of the condition can be varied. Extreme situations contain a lot of productions that are supposed to be applied in parallel. This can be simulated by using big numbers for the parameter *noRules*. Since this is one of the major variables at run time we test all configurations of other parameters with 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 production applications. In the tests these sets are randomly generated as combinations from the set of three implemented productions. The statistics of these three productions are shown in Table 7.1. In addition to the two productions from the running example this table contains one production *addWindow* which adds a new window to the UI model that is not connected to the dialog model to provoke adaptation-consistency conflicts.

Table 7.1.: Statistics of the adaptations in the running example.

Rule	Matched		Added		Deleted	
	Nodes	Edges	Nodes	Edges	Nodes	Edges
addBackButton	11	8	3	7	0	0
mergeWindows	20	23	0	3	7	16
addWindow	1	0	1	1	0	0

The parameter *noTimes* is always set to 1000 to test 1000 cases to get a good mean and balance out negative factors like slow initialization and garbage collection that stem from the programming language Java. To minimize the effect of garbage collection a *System.gc()*, followed by a two second waiting period is triggered after each test. This gives the Java Virtual Machine time to perform a garbage collection and thus lowers the likelihood of garbage collection during the tests.

The main concern when dealing with graph transformation is that the matching algorithms suffer in extreme cases. These are cases where the size of the model is big and thus contains a large number of potential match targets. This is simulated by using different values for the parameter *modelSizeIndicator*. This parameter is interpreted by duplicating the model structure from the running example. This serves to build complexity as the nodes that are matched in the original model are now available multiple times and thus the number of possible matches is guaranteed to increase at least linear with the parameter *modelSizeIndicator*. This parameter is tested with values 1, 10 and 100. Table 7.2 contains the size of the original model.

The size of conditions is an expected source of inefficiency for the analysis of adaptation-adaptation conflicts. The parameter *conditionSize* generates conditions of different sizes by selecting the according number of nested conditions. These conditions are compiled into one condition by using conjunction. The conditions are randomly selected from the set of implemented conditions. Thus, they can contain duplicates of the same conditions. This does not influence the test results as no optimization with regards to already tested

Table 7.2.: Statistics of the main window in the running example.

Running Examlpe - Main Window	
No. Nodes	56
No. Edges	91
No. Attributes	70

conditions is performed. Table 7.3 summarizes the statistics of the conditions from the running example. While conditions i) to v) contain approximately the same number of nesting levels and nodes, conditions vi) to viii) contain significantly more nodes. The analysis only selects from the conditions i) to v) to stabilize the size of the generated conditions. Larger sizes of conditions are tested by using a larger value for the parameter *conditionSize*. This parameter is tested with values 1, 10 and 100.

Table 7.3.: Statistics of the condition in the running example.

Condition	Nr. Nesting Levels	Nr. Nodes
i)	4	6
ii)	4	3
iii)	5	8
iv)	4	3
v)	4	4
vi)	4	15
vii)	4	18
viii)	6	32

While performing the test execution times for the implementation are recorded. The class *TestResult* contains these times. It reflects the following recorded times:

- *conflictCalculation*: The time for calculating adaptation-adaptation conflicts between the given production applications
- *conflictfreeCombinations_conflictCalculation*: The time for calculating the conflict free combinations of production applications after the analysis for adaptation-adaptation conflicts
- *maximumGraph*: The time for calculating the maximum graph
- *fulfilmentConditionCalculation*: The time for calculating fulfilment conditions for the given nested condition
- *fulfilmentConditionSimplify*: The time for simplifying fulfilment conditions for the given nested condition

- *conflictfreeCombinations_fulfilmentCondition*: The time for calculating all combinations of production applications that fulfil a fulfilment condition

The output of the test is a text in the Java console containing the test times for each of the cases. This text is formatted in a way that can be copied directly into an excel spreadsheet to calculate the mean, standard deviation and maximum / minimum values. The excel spreadsheet also allows to plot diagrams from these times.

We executed the tests on a notebook from Lenovo of model X230. The test notebook contains an Intel Core i3-3120M Processor (3MB Cache, 2.10GHz) and memory of 4 GB DDR3 - 1600MHz. More specification of the notebook can be found on the website of the manufacturer.[1]

In the remainder of this section we describe the results of this analysis. Figure 7.20 shows the times for varying the model sizes with model size indicators 1, 10 and 100. A models size of 1 corresponds to a model with 56 nodes, 70 attributes and 91 edges (including mapping edges between models). The model of factor 10 contains 290 nodes, 385 attributes and 487 edges. The model of factor 100 contains 2630 nodes, 3535 attributes and 4447 edges. As expected the times grow with number of productions and size of the model. For a model size of one the computation process for 100 productions takes 27 milliseconds. With 10 times model size the time grows to 58 milliseconds. At hundred times model size the maximum time is around 2.1 seconds. The main effort in the extreme cases lies in the calculation of fulfilment reasons for conditions. This is an expected result as this is the only part of the algorithm that requires finding a match, which is affected by the model size. In cases with a small structure the calculation of the maximum graph requires a significant amount of the overall time. However, this factor does not grow fast with the model size (ca. 9 ms for 1x modelSize, 10 ms for 10 x model size and 28 ms for 100 x model size) compared to the calculation of adaptation-consistency conflicts.

Figure 7.21 shows test results for different configurations of condition sizes. It shows results for 1, 10 and 100 conditions. As expected the times grow with number of productions and conditions. The overall time grows to 67 ms for 10 conditions and 478 ms for 100 conditions. Again, the main effort is in the calculation of fulfilment conditions.

According to these tests the times grow with model size and number of conditions. In a final test we explored the combination between both. We tested the combinations of 10 times model size and 10 conditions and 20 times model size and 20 conditions. Figure 7.22 shows the results. The overall time for a factor of 10 grows from 27 ms to 241 ms. For a factor of 20 the overall time is at 1168 ms. Again, the main effort is in the calculation of fulfilment conditions.

The tests have been performed to evaluate the performance of the implementation and tweak it. Naturally, the test times need to be interpreted with regards to an intended application of the implementation. They have to be seen in the light of the size of the potential model, number of conditions and number of parallel applicable production applications. In the next chapter we study a concrete example and test the performance of the algorithms against requirements from the project SOE.

[1] http://shop.lenovo.com/gb/en/laptops/thinkpad/x-series/x230/

7.5. Summary

In this chapter we described the implementation of our approach. The implementation is a Java library that can be used in the scope of the Eclipse Modeling Framework and the Henshin plugin for graph transformation. As a basis we discussed the relation between EMF models and graph diagrams. The detection algorithms for adaptation-adaptation and adaptation-consistency conflict are implemented as Java methods. In order to showcase the workflow of an implementation we gave an example of a conflict analysis using the implemented methods.

The running example has been used to evaluate this implementation. We discussed means to artificially increase the size of models and the number of conditions and productions to test the impact of these factors on the time required for analysis. The results show that the time grows with the number of model elements, conditions and productions as expected. The applicability of our algorithms in a specific domain depends on these factors and on the requirements of the domain. In the next chapter we evaluate our approach. In this chapter we test the applicability of the implementation in the scope of the case study from the project SOE.

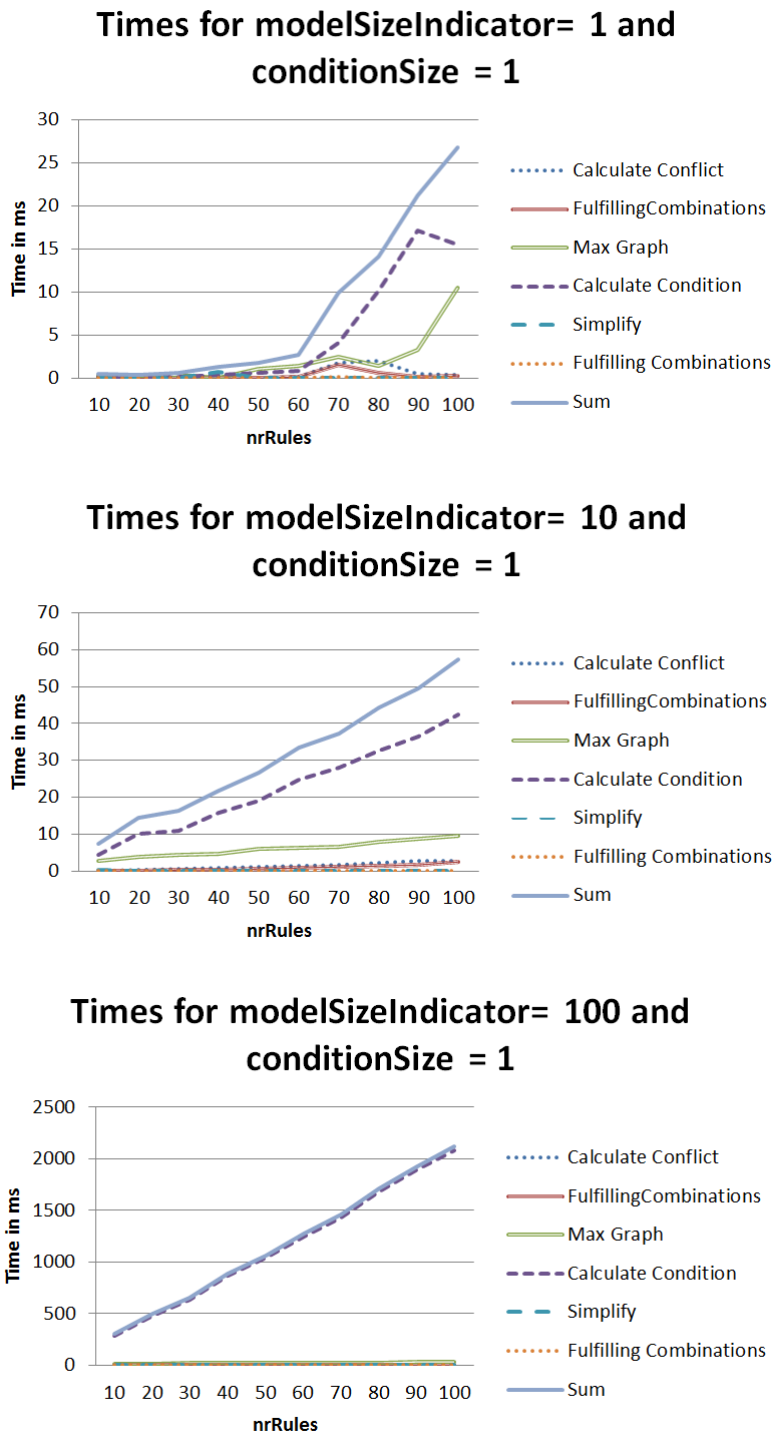


Figure 7.20.: Test results for model sizes 1, 10 and 100.

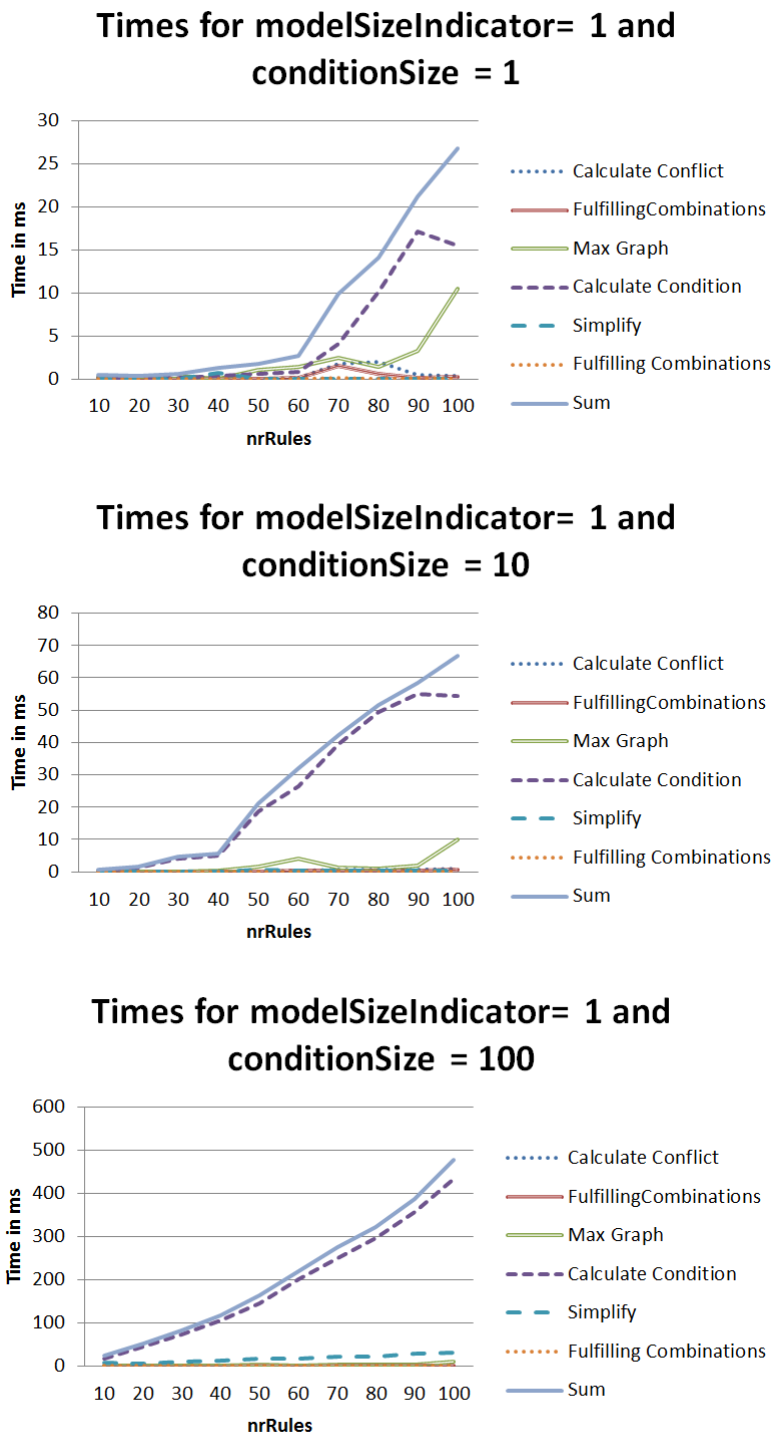


Figure 7.21.: Test results for condition sizes 1, 10 and 100.

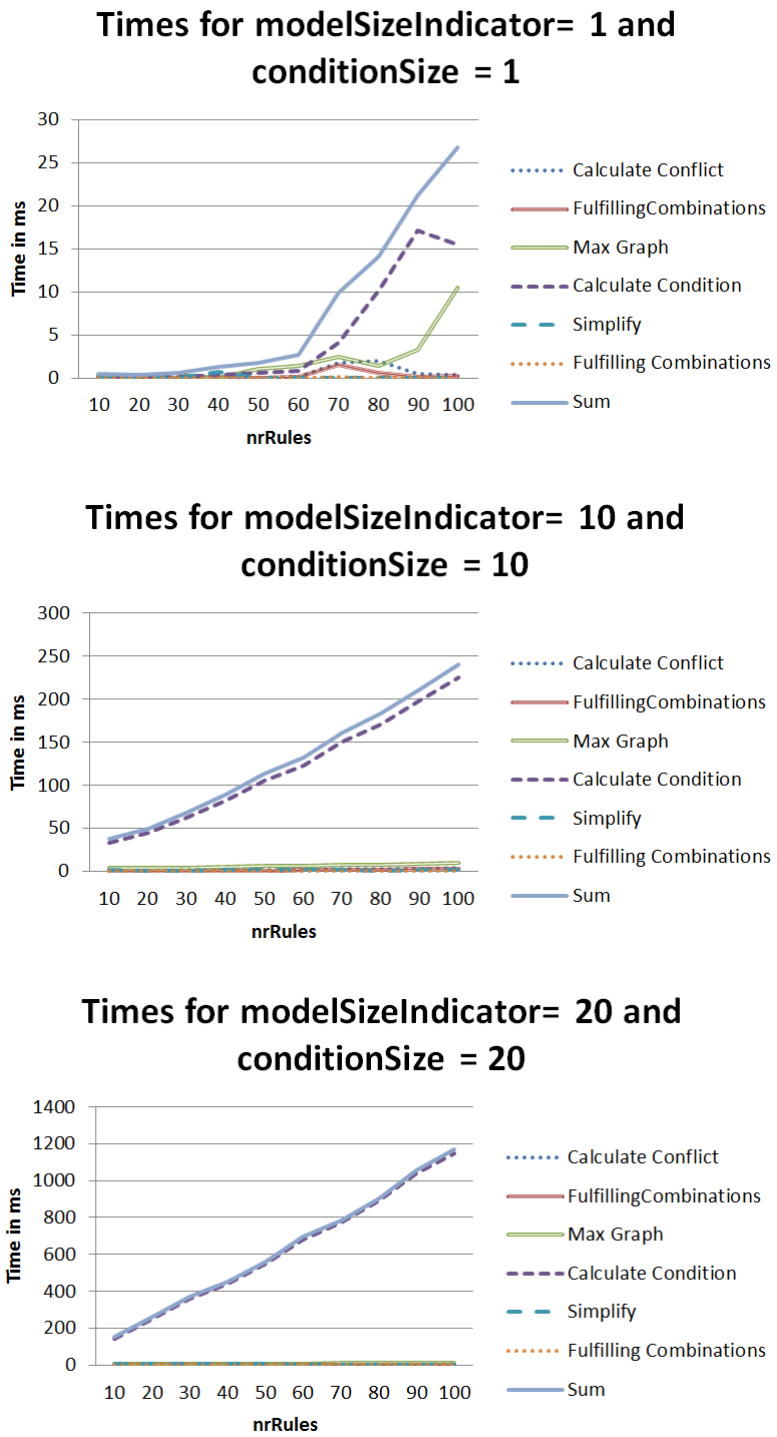


Figure 7.22.: Test results for model and condition sizes of 1, 10 and 20.

8. Evaluation

This chapter describes the evaluation of the Trollmann approach. We aim to evaluate the correctness and the practicality of our approach. Correctness is stated in a set of theorems in chapter 6. These theorems have been peen proven thus showing correctness. The main theorems and their significance are summarized in Section 8.1.

In the previous chapter we tested the performance of our algorithm based on different configurations of influencing factors. In this chapter we apply this implementation to the SOE project to test it in a realistic example. This test evaluates the practicality of the Trollmann approach in the scope of this project. It is described in Section 8.2.

8.1. Main Theorems and Proofs

The correct behaviour of the Trollmann approach has been states as theorems in Chapter 6. These theorems have been proven. The proofs are provided in Appendix B. In this section we review these theorems and explain their significance.

The first main theorem is Theorem 1. By proving this theorem we have shown that graph diagrams form a finitary $\mathcal{M}$-adhesive category with $\mathcal{M}$-initial object. This enables the application of graph transformation, nested conditions and several theoretical results. Parallel and sequential independence and the Local Church Rosser Theorem form the basis for the analysis of adaptation-adaptation conflicts. The Parallelism Theorem forms the basis for showing correctness of the analysis of adaptation-consistency conflicts. These theorems are applicable because Theorem 1 holds.

The second main theorem is Theorem 5. This theorem states that the set-based conditions from Definition 17 imply parallel independence of the involved production applications. We proof this by showing equivalence to the morphism-based conditions for parallel independence, whose correctness is stated in Theorem 4. The proof of these theorems enables us to calculate dependencies between a set of production applications without the need to apply them. This is the foundation for our analysis of adaptation-adaptation conflicts. The proofs for these theorems on the level of graph diagrams make use of the respective theorems on the level of attributed typed graphs (cf. Theorems 3 and 2). We also provide proof for these theorems in Appendix B.

The third main theorem is Theorem 10. This theorem describes the correct behaviour of the calculation of fulfilment conditions for a nested condition. It states that a nested condition is fulfilled after the application of a subset of production applications if the fulfilment condition, generated by the algorithm *conditions*, is evaluated to *true*. By proving this theorem we showed that this algorithm and thus the analysis of adaptation-consistency conflicts behaves correctly. The proof for this theorem relies on the correct

behaviour of the algorithm *existenceCondition*, which is stated in Theorem 9, and on the correct behaviour of the functions *averse*, *required* and *possibleMorphisms*, which is stated in Theorem 8. The maximum graph forms the foundation for these theorems. By proving Theorems 6 and 7 we have shown that the maximum graph always exists and that it contains all models that can be created by applying any subset of these production applications.

The proofs of these theorems guarantee the correct behaviour of our approach. In the next section we evaluate the practicality in the project SOE.

8.2. Evaluation of the Case Study

In this section we describe the implementation and evaluation of the case study scenario. The implementation is based on EMF, Henshin and our implementation of the adaptation conflict detection. It is presented in Section 8.2.1. We describe the evaluation setup and results in Section 8.2.2.

8.2.1. Implementation of the Case Study

This section describes the implementation of the case study scenario discussed in Section 5. As described in Section 7, the implementation is based on the Eclipse Modeling Framework and the Henshin plugin.

As a basis for the analysis, the plan window has been implemented as an EMF model. The structure of this window is as described in Section 5.2 in Figure 5.1. Table 8.1 summarizes the number of nodes, edges and attributes in this window.

Table 8.1.: Statistics of the model of the plan window in the case study.

Plan Window	
No. Nodes	2378
No. Edges	8764
No. Attributes	10565

In addition to the elements indicated in Section 5.2, the plan window contains a description of the house the software system is installed in. Model elements describe available rooms, devices and users. However, the majority of elements stems from the models related to the user interface. For the tests the complete model has been used. There is potential to reduce the size of the model and the size of the matched section of the model by excluding the irrelevant model elements. However, for the purpose of this case study these optimizations have not been implemented to enhance the complexity of the test. The graph representation of the model contains 2378 nodes, 8764 edges and 10565 attributes. Compared with the running example in Section 8.2.2 this window contains 42 times more nodes, 96 times more edges and 150 times more attributes.

Table 8.2.: Statistics of the adaptations in the case study.

Rule	Matched		Added		Deleted		Nr.
	Nodes	Edges	Nodes	Edges	Nodes	Edges	Matches
Adaptation 1-1	31	46	63	133	1	15	1
Adaptation 1-2	31	46	12	42	1	15	1
Adaptation 2	9	8	3	9	0	6	1
Adaptation 3-1	39	38	4	28	14	38	7
Adaptation 3-2	39	38	0	12	2	14	7
Adaptation 4	172	1	169	672	1	672	1

We implemented six Henshin rules to achieve the adaptations described in Section 5.3. Table 8.2 contains a summary of the size of these rules. The table describes the size of the matched graph by showing the number of nodes and edges in the left hand side of the production. These elements need to be matched when finding a match. It also shows the number of added and deleted nodes and edges in each adaptation. Some adaptations need to be applied multiple times to achieve the desired adaptation. For example, $Adaptation3-1$ manipulates the user interface of the layout for one day. Accordingly, it is applied seven times to the original plan window to manipulate all seven days. The number of matches is also shown in the table.

For adaptation 1 two rules have been implemented. Both rules change the dialog model by removing the place that previously represented all weekdays and add a different structure. $Adaptation1-1$ adds a separate place for each day, while $Adaptation1-2$ adds one place for the weekdays and one place for the weekend. These new places are connected to the interaction groups for each weekdays interactor in the UI model via mappings. The new dialog elements also contain transitions for switching between the new places. These transitions are connected to the control elements in the UI model (for switching forward and backwards) via mappings.

Figure 8.1 shows the second adaptation. The adaptation is achieved via one Henshin rule. The purpose of this rule is to reorder the days into two columns. The adaptation changes the layout model. Intermediate containers for representing rows of two weekdays are added, e.g., the container *mon_tue* contains the layouts *Monday* and *Tuesday*. This Henshin rule matches all days at once. Accordingly, only one match is required.

The third adaptation is implemented via two Henshin rules. These rules are depicted in Figure 8.2. Both rules change the layout for a single day from a 2 x 12 grid to a 4 x 6 ($Adaptation3-1$) or a 1 x 24 grid ($Adaptation3-2$) respectively. Both adaptations change the layout model tree and insert layouts to model the respective grid. $Adaptation3-1$ adds the four new containers $00-05$, $06-11$, $12-17$ and $18-23$, which represent the new rows in the layout. Each of these containers contains six of the original layouts for an hour. $Adaptation3-2$ removes the containers for the original two rows and adds the layouts for hours directly under the root node for the layout. Both

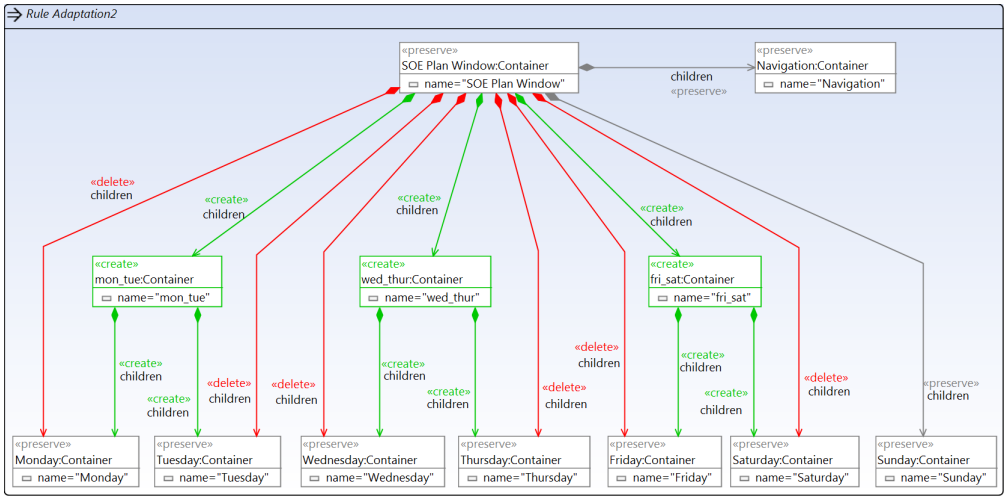


Figure 8.1.: The Henshin rule for Adaptation 2.

rules are parameterized via a parameter *containerName* that can be used to produce the correct match for one specific weekday. Seven matches into the plan window exist for this rule; one for every weekday.

Adaptation4 adds mappings that transfer the heating level for each hour to the respective output element in the user interface model. Figure 8.3 shows an excerpt from this rule. Each element of type *OutputOnly* represents an hour of a weekday and is connected to the heat plan of this room via a mapping type $ContextModeltoUIModelValue$. Along with each new mapping the *source* reference to the heat plan, the *target* reference to the hour label, the *type* reference to its mapping type and the *mappings* reference from the containing mapping model is added. The figure exemplifies the hours 0 to 9 on Monday.

The conditions of the scenario, given in Section 5.4, have been modelled as empty Henshin adaptations with left application conditions. Overall there are 11 conditions. The statistics of these conditions are shown in Table 8.3. The table shows the number of nodes and the number of nesting levels in each condition.

Figure 8.4 contains an example for a nested condition. The figure shows the condition $D1$. The condition requires that all places are connected to a mapping of type $DialogModeltoUIModel$. The missing universal-quantifier in Henshin conditions is substituted by the equivalent expression $\neg\exists\neg$. For the analysis all conditions are combined by creating a conjunction of these conditions. This enables us to check all conditions in one analysis pass.

The next section discusses the analysis setup and results.

8.2.2. Evaluation of the Case Study Szenario

This section describes the criteria, setup and results of the evaluation of our case study. We begin by describing the evaluation criteria defined by the SOE developers. Subsequently, we describe the test setup and discuss and interpret the results of the test.

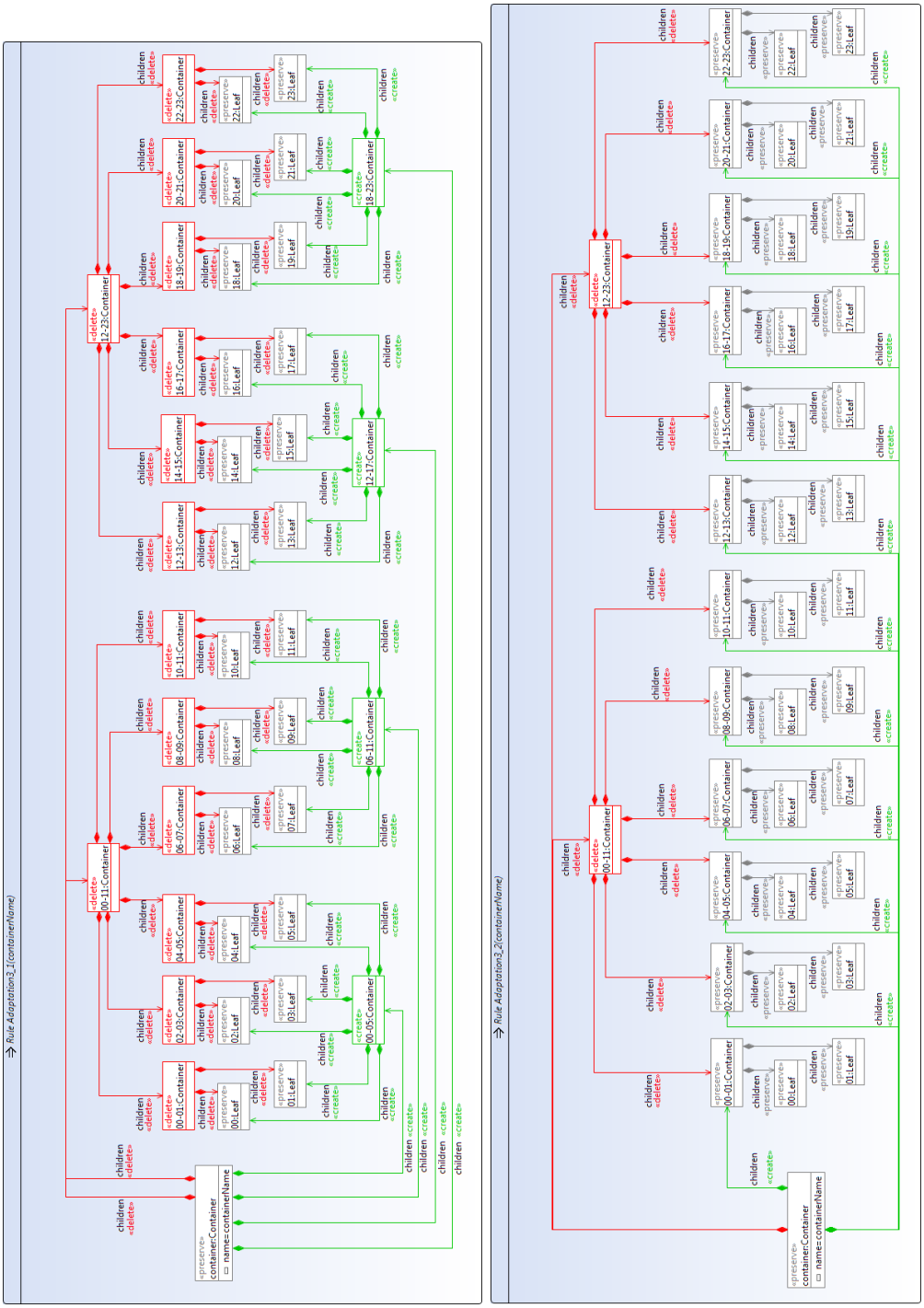


Figure 8.2.: Two Henshin rules for Adaptation 3.

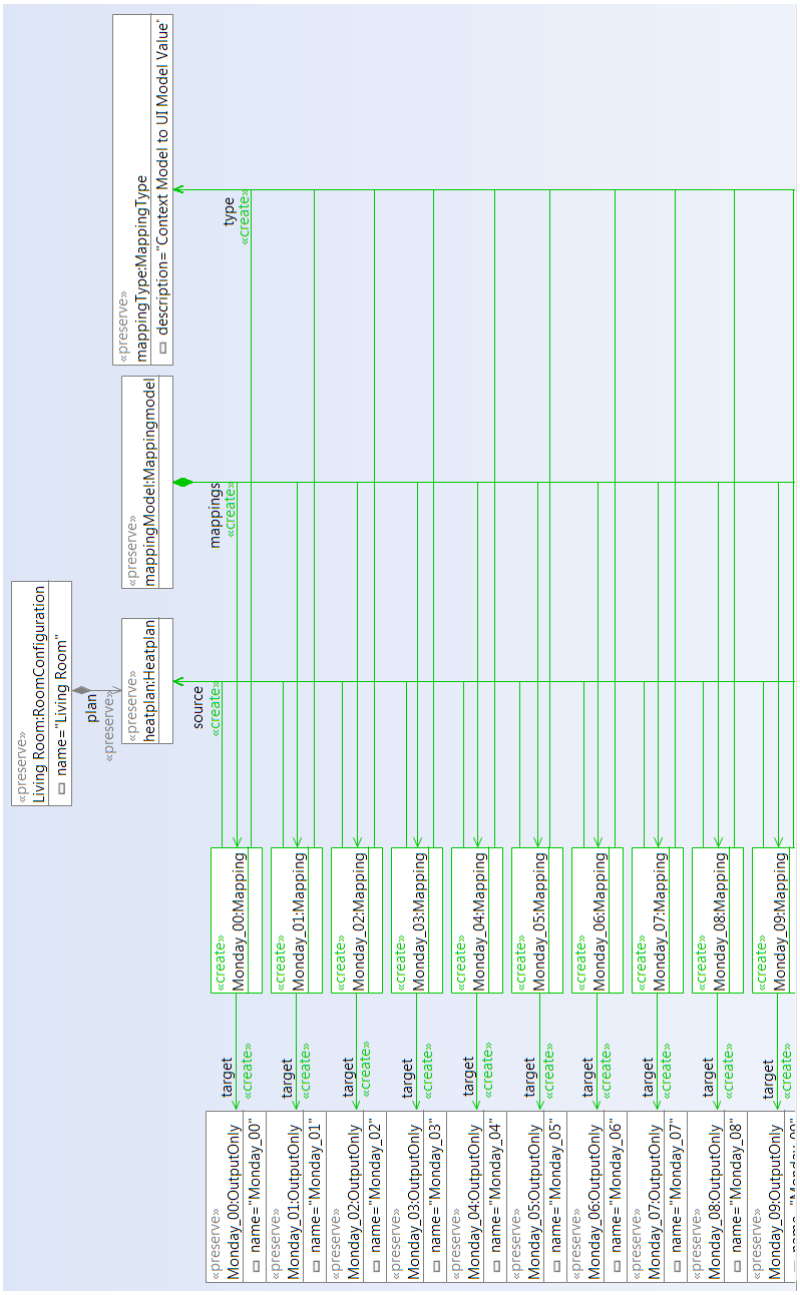


Figure 8.3.: An excerpt from the Henshin rule for Adaptation 4.

Table 8.3.: Statistics of the conditions in the case study.

Condition	Nr. Nesting Levels	Nr. Nodes
D1	4	4
D2	4	4
UI1	4	4
UI2	5	11
UI3	4	7
UI4	4	7
L1	5	11
M1	4	6
M2	4	6
M3	4	6
M4	5	13

Before executing the experiment we discussed the evaluation criteria with the lead developer for UI components in the project SOE. The main concern of the developer was the execution time of the analysis and its impact on the usability of the UI. The project uses a web-based graphical user interface and the consistency analysis is carried out at run time. Accordingly, it has been decided to use desired reaction times of web-based graphical user interfaces as criteria for evaluation. Hoxmeier and Dicesare found that long answering times impact user satisfaction and that reaction times in excess of 12 seconds can cause the user to discontinue the use of the software system [32]. Shneiderman further specifies different kinds of tasks and appropriate answering times [119]:

- *Elementary tasks (e.g., typing, cursor motion or mouse selection):* 50 - 150 milliseconds
- *Simple frequent tasks:* 1 second
- *Common tasks:* 2-4 seconds
- *Complex tasks:* 8-12 seconds

The opinion of the SOE developer is that conflict resolution should be classified as a common task. The detection of conflicts and calculation of additional information, like fulfilling combinations, should stay within the 2-4 second range. However, if there is no conflict the conflict detection should not delay other tasks. Adaptation itself (without conflict resolution) has been classified as a simple frequent task. Thus, it should not take longer than 1 second. Accordingly, the detection of conflicts (without the calculation of additional information) should stay way under this boundary. From this we derived the following evaluation criteria:

- The time for detection of conflicts should stay below 1000 milliseconds.

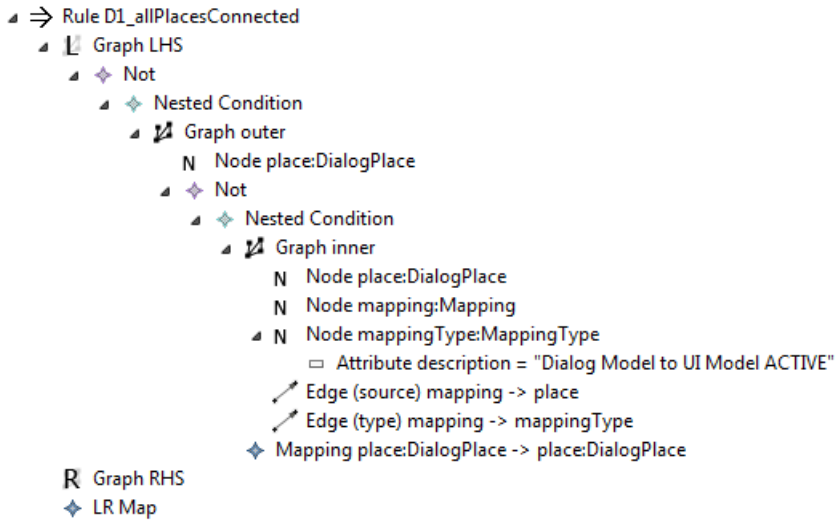


Figure 8.4.: The implementation of a condition.

- If there is a conflict the time for detection and calculation of additional information should stay below 2000 - 4000 milliseconds.

As a test scenario, the plan window, described in the previous section, is used as model G for the analysis. All adaptation rules are matched to this model as indicated by Table 8.2. Adaptations 3-1 and 3-2 are applied with all 7 matches. Thus, there is an overall set of 18 production applications to be applied. These production applications form the analysed set of production applications $Prods$. The conditions from the scenario (cf. Table 8.3) form the set of analysed conditions $Cons$. For analysis they are joined into one condition using conjunctions.

The test implementation loads the model, all rules and all conditions into memory and builds G, $Prods$ and $Cons$. Subsequently, the analysis for adaptation-adaptation and adaptation-consistency conflicts is executed. The analysis of adaptation-adaptation conflicts consists of the calculation of conflicts and the calculation of fulfilling combinations. The analysis of adaptation-consistency conflicts calculates the maximum graph and subsequently calculates and simplifies fulfilment conditions. Finally, fulfilling combinations are calculated. The tests are performed 1000 times. Since the implementation is in Java, the influence of garbage collection on single test runs is a concern. $System.gc()$ is manually triggered before each test and a waiting time of two seconds is forced to enable garbage collection to be executed between tests and thus minimise the chance of garbage collection during the tests. Furthermore, the first 100 tests have been disregarded to exclude start up overhead and simulate a running system.

We executed the tests on a notebook from Lenovo of model X230. The test notebook contains an Intel Core i3-3120M Processor (3MB Cache, 2.10GHz) and memory of 4 GB DDR3 - 1600MHz. A specification of the notebook can be found on the Lenovo website.[1]

[1] http://shop.lenovo.com/gb/en/laptops/thinkpad/x-series/x230/

The results of the analysis are listed in Table 8.4. The table shows minimum value, maximum value, mean and standard deviation of the times for all steps of the adaptation-adaptation and adaptation-consistency conflict analysis.

The average overall time is 384 milliseconds, of which 7 milliseconds derive from the analysis of adaptation-adaptation conflicts and 377 from adaptation-consistency conflicts. In adaptation-consistency conflicts the calculation of fulfilment conditions takes 341 milliseconds (89% of the overall time). The calculation of fulfilling combinations for both types of conflict is far under one millisecond. This is consistent with the values calculated in Section 7.4 where the time for calculating the fulfilling combinations was only significant for bigger numbers of production applications.

The minimum overall time is 372,39 ms. The maximum overall time is 1318 ms. However, this seems to be a runaway value. Of the 1000 samples taken only one had an overall time of over 1000 ms. Most other values are close to 380 ms, as can be derived from the standard deviation of five ms.

The mean time stays well below the one second required for simple frequent tasks. On average the conflict detection requires below 400 ms, which leaves 600 ms for other tasks. Compared with the time for complex tasks (2-4 seconds) the conflict detection requires 10% - 20% of the overall time. Accordingly, the average time is well below the available time for conflict resolution as a complex task.

As stated in Section 6.5.3 it is not always necessary to generate fulfilment conditions right away. Instead the result of the adaptation could be checked and the calculation of the fulfilment condition can be delayed till an inconsistency is detected. Since this analysis requires about 90% of the overall time it would be a good idea to delay it to reduce the effort for conflict detection when it is not necessary. However, even without this optimisation the time meets the specified criteria.

8.3. Summary

In this chapter we described two evaluations of the Trollmann approach. The correct behaviour has been shown by proving the main theorems. We explained the significance of these theorems for the Trollmann approach. In addition, we described the application of the implementation of our conflict detection to the case study from the project SOE. The implementation has been evaluated with regards to requirements posed by the lead developer for the user interface. The results show that the detection of adaptation conflicts can be performed in the given time. This evaluation shows the practicality of our approach within the parameters of this project.

The next chapter concludes the thesis.

Table 8.4.: Times for conflict detection in the case study in ms.

	Adaptation - Adaptation		**Adaptation - Consistency**				**Sum**
	detection	fulfilling combinations	maximum graph	conditions	simplification	fulfilling combinations	
minimum	6.22	0.30	32.70	331.39	0.47	0.01	372.39
maximum	8.39	0.36	36.83	537.56	937.48	0.019	1318.47
mean	6.31	0.31	33.06	341.41	3.07	0.01	384
standard deviation	0.04	0.01	0.18	3.20	3.71	0.00	5.58

9. Conclusions

The aim of this thesis is the detection of adaptation conflicts and the provision of knowledge for conflict resolution based on models@run.time. We categorised adaptation conflicts into adaptation-adaptation and adaptation-consistency conflicts and posited a set of questions that need to be answered for both types of conflicts to provide knowledge for conflict resolution. The detection of adaptation conflicts is based on a formalism for representing models, their adaptation and structural conditions. We defined the formalism of graph diagrams for this purpose. For this formalism, we developed a set of analysis methods that can extract the required knowledge about adaptation conflicts and answer the questions.

The evaluation is performed in two phases. On the one hand, the correctness of the formal definitions in this thesis has been proven in formal proofs. On the other hand, practical applicability of the algorithms has been shown in a proof of concept implementation of the framework that has been applied in the scope of a research project.

In this section we revisit the goals and contributions of this thesis and compare them with our results in Section 9.1. Section 9.2 points out future work for the synthesis of the results in this thesis. Finally, we make some concluding remarks in Section 9.3.

9.1. Contributions

As described in Section 1.1 the goal of this thesis is to enable the detection of adaptation-adaptation and adaptation-consistency conflicts based on models@run.time. The aim is to develop a formalism that can deal with multiple related models and define methods for conflict detection based on this formalism. We posited a set of questions about conflicts that need to be answered for both types of conflicts to provide knowledge for conflict resolution in Section 3.4.

Contribution 1 is the specification of a formalism for representing structure, adaptation and structural consistency requirements of multiple related models. The result is the formalism of typed graph diagrams, which is an extension of attributed typed graphs for the purpose of expressing the structure of multiple models (cf. Section 6.4.1). The formalism builds on the power of expression of attributed typed graphs to represent models of different modelling languages. Morphisms in the diagram are able to represent relations between models. We combine the formalism with nested conditions and graph transformation productions in the double pushout approach to represent structural requirements and adaptations of graph diagrams. This enables the application of the existing formal results from graph transformation theory.

Contribution 2 is the extension of these theoretical results to answer the questions that are relevant for conflict resolution (cf. Section 3.4). For the detection of adaptation-

adaptation conflicts the detection of parallel independence has been optimized to an algorithm that is able to find dependencies between a set of production applications without the need to apply them. As a result of this algorithm a set of conflict descriptions is produced. These conflict descriptions specify which adaptations are in conflict with respect to which elements. We showed how the questions about conflicts can be answered for adaptation-adaptation conflicts based on these conflict descriptions.

For the detection of adaptation-consistency conflicts an upper-bound estimation of potential morphisms is derived from the maximum graph in Section 6.5.3. This maximum graph is used to estimate morphisms that can exist in any subset of adaptations and thus to reason about subsets that fulfil a nested condition when applied. On this basis we defined an algorithm for generating fulfilment conditions. The generated conditions can be evaluated with a subset of adaptations to find out whether this subset fulfils the condition when applied. The fulfilment condition is used to answer the questions about conflicts for adaptation-consistency conflicts.

The main results of the detection of both types of conflicts have been formulated as theorems in Chapter 6. These theorems have been proven, thus showing correct behaviour of these algorithms. The meaning of these theorems has been discussed in Section 8.1. We supply our proofs for these theorems in Appendix B.

The third contribution is an implementation of the analysis methods. This implementation is based on the Eclipse Modeling Framework and the Henshin plugin for graph transformation. It consists of a set of static classes described in Section 7. These classes can be used for analysis of adaptation-adaptation and adaptation-consistency conflicts.

Our fourth contribution is a test of the implementation that evaluates its applicability at run time. Two tests have been performed. Section 7.4 tests the implementation based on the running example. We test the performance with different configurations of parameters to assess their impact. These results can be used to estimate whether it is practical to apply our algorithms in a specific modelling framework. We supplemented these tests with the implementation of a case study in the scope of adaptive user interface models from the research project SOE. The conflict detection mechanism was evaluated and met the requirements posed by the developers in this project. We described the implementation of this case study in Section 8.

9.2. Future Work

This section describes potential future work. The most obvious way to extend our research is to complement the conflict detection algorithms described in this thesis with conflict resolution strategies. Specifically, we envision an interactive approach which resolves conflicts with the user in the loop. The implementation of such strategies will also yield further requirements to the analysis environment. In addition to information about conflicting adaptations and elements, information about the actual adaptation or the severity of the conflict may be required to find a suitable resolution strategy. Algorithms for providing this additional information or to reason about properties of potential states of the models are subject to future work.

The selection of adaptations and generation of matches has been excluded from the conflict detection. Both aspects could be integrated into a more sophisticated analysis of adaptation conflicts. The analysis of the selection of adaptations could lead to information regarding conflicting preconditions for adaptations. The match generation can also be integrated into conflict analysis. For example, adaptations that are not strictly parallel independent may still be both applicable with different match morphisms. An analysis that can propose alternate or changed matches could be a great benefit for conflict resolution. For this, a formal representation of the rule selection and matching strategy is required. The maximum graph could provide an upper bound for reasoning about potential matches.

Another potential extension of the approach is a more sophisticated algorithm for dealing with the reversion of production applications. In this thesis we restrict the reversion to production applications that can be reverted independent of other production applications. A more sophisticated algorithm could take dependencies between applied production applications into account. For example, such an algorithm could revert all dependent production applications along with the original production application. The resolution of such dependencies is a complex task. However, as long as the reversion is still represented by a single graph transformation production, it can be integrated into our algorithms the same way as the reverse production applications. The generation and maintenance of the set of reverse production applications as well as its interpretation in a conflict resolution is more complex in these cases.

In the Trollmann approach we use the formalisms of graph transformation and nested conditions as a formal basis. We selected these formalisms because of their good formal foundation and the variety of existing results that can be applied for conflict detection. However, there are also other formalisms for representing adaptations and structural requirements as described in the related work in Section 4.2. One interesting point for future work is to explore the relation to these formalisms and the applicability of our algorithms in these formalisms. For example, OCL constraints have already been translated into nested conditions for formal analysis [89] and could be integrated into the Trollmann approach based on this translation.

Graph diagrams are the result of a categorical construction over the category of attributed graphs. A similar construction can be made for other categories, e.g., petri nets. As long as the category is an $\mathcal{M}$-adhesive category, the diagram is also $\mathcal{M}$-adhesive. Several of the results given in this section are not specific to attributed graphs as a basis. The only definitions that are specific to attributed graphs are the algorithm for detecting adaptation-adaptation conflicts and the conditions for fulfilment of nested conditions. Both definitions refer to specific elements within the attributed graphs contained in the diagram. We plan to explore how elements could be generalized so our approach can also be applied to other categories. Intuitively, it should be applicable to any $\mathcal{M}$-adhesive category that is constructed over sets similarly to attributed typed graphs.

One other thing that will be explored in future work is the applicability of our formal framework to existing model-based frameworks. The example in Section 8 shows a model-based framework based on the theory in this thesis can be created. It would be interesting to see it applied to existing run time frameworks.

9.3. Concluding Remarks

In this thesis we make the first step towards an analysis of adaptations based on run time models. This analysis is required to assure that a self-adaptive software system performs correctly despite adaptations. Since the full assurance at design time is not always possible a run time method is required for this.

The results of this thesis enable an automated detection of adaptation-adaptation and adaptation-consistency conflicts. As pointed out in the future work, this analysis needs to be complemented with resolution mechanisms that can resolve the detected conflicts and assure correct behaviour. Several other extensions of these results are also possible in order to broaden applicability and gather more information for analysis.

The detection and resolution of adaptation conflicts is only one of the aspects of adaptations that needs to be assured. A full feedback loop for the adaptation process needs to be established that can dynamically analyse and maintain the adaptive behaviour based on run time models of the adapted aspects and the adaptations themselves [10].

One of the key challenges for such assurance approaches is the consistency of the run time models with respect to their system under study. An analysis method can only be as good as the information it is based on. Thus, the maintenance of the causal connection at run time is a central aspect for all assurance approaches. A structured approach for handling and assuring the causal connection itself is required to enable run time assurance to tap its full potential.

References

[1] Weiser, M.: The computer for the 21st century. Scientific American **265**(3) (1991) 66–75

[2] Cheng, B.H.C., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.D.M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: A research roadmap. In: Software Engineering for Self-Adaptive Systems, Springer-Verlag (2009) 1–26

[3] Blumendorf, M.: Multimodal Interaction in Smart Environments A Model-based Runtime System for Ubiquitous User Interfaces. PhD thesis, Technische Universität Berlin (2009)

[4] Zhang, J., Cheng, B.H.C., Goldsby, H.: Amoeba-rt: Run-time verification of adaptive software. In: Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Reports and Revised Selected Papers. Volume 5002 of Lecture Notes in Computer Science., Springer-Verlag (2007) 212–224

[5] McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. IEEE Computer **37**(7) (2004) 56–64

[6] Kent, S.: Model driven engineering. In: Proceedings of the Third International Conference on Integrated Formal Methods (IFM 2002), Springer-Verlag (2002) 286–298

[7] Clarke, E.M., Grumberg, O., Peled., D.: Model checking. The MIT Press (2001)

[8] Chandra, S., Godefroid, P., Palm, C.: Software model checking in practice: an industrial case study. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), ACM Press (2002) 431–441

[9] Blair, G., Bencomo, N., France, R.B.: Models@ run.time. IEEE Computer **42**(10) (2009) 22–27

[10] Eder, K.I., Villegas, N.M., Müller, H.A., Trollmann, F., Pelliccione, P., Schneider, D., Grunske, L., Rumpe, B., Litoiu, M., Perini, A., Gogolla, M., Qureshi, N.A., Cheng, B.H.: Assurance using models at runtime for self-adaptive software systems. In: Special Issue on Models@run.time. Lecture Notes in Computer Science, Springer-Verlag (2014) to appear.

[11] Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H., Bruel, J.M.: RELAX: Incorporating uncertainty into the specification of self-adaptive systems. In: Proceedings of the 17th International Requirements Engineering Conference (RE 2009), IEEE Computer Society (2009) 79–88

[12] Object Management Group: UML Specification, Version 2.0 (2006)

[13] Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE 1998), Springer-Verlag (1998) 21–37

[14] Villegas, N.M., Müller, H.A., Tamura, G.: On Designing Self-Adaptive Software Systems. Sistemas & Telemática **9**(18) (2011) 29–51

[15] Calvary, G., Coutaz, J., Thevenin, D.: A unifying reference framework for the development of plastic user interfaces. In: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction (EHCI 2001), Springer-Verlag (2001) 173–192

[16] Calvary, G., Coutaz, J., Thevenin, D., Bouillon, L., Florins, M., Limbourg, Q., Souchon, N., Vanderdonckt, J., Marucci, L., Paternò, F., Santoro, C.: The cameleon reference framework. Deliverable 1.1 (2002)

[17] Vanderdonckt, J.: A mda-compliant environment for developing user interfaces of information systems. In: Proceedings of the 16th Conference on Advanced Information Systems Engineering (CAiSE 2005). Volume 3520 of Lecture Notes in Computer Science., Springer-Verlag (2005) 16–31

[18] Ehrig, H., Habel, A., Kreowski, H.J., Parisi-Presicce, F.: From graph grammars to high level replacement systems. In: Proceedings of the 4th International Workshop on Graph-Grammars and their Application to Computer Science, Springer-Verlag (1991) 269–291

[19] Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science **19** (2009) 245–296

[20] Eclipse Foundation: The Eclipse Modeling Framework (EMF) Overview (2005)

[21] Favre, J.M., Nguyen, T.: Towards a megamodel to model software evolution through transformations. Electronic Notes in Theoretical Computer Science **127**(3) (2005) 59 – 74

[22] Miller, J., Mukerji, J.: Model driven architecture (MDA). Draft Technical Report ormsc/2001-07-01, Architecture Board ORMSC (2001)

[23] Seidewitz, E.: What models mean. IEEE Software **20**(5) (2003) 26–32

[24] Favre, J.M.: Foundations of meta-pyramids: Languages vs. metamodels – episode ii: Story of thotus the baboon. In: Language Engineering for Model-Driven Software Development. Number 04101 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI) (2005)

[25] Favre, J.M.: Foundations of model (driven) (reverse) engineering : Models - episode i: Stories of the fidus papyrus and of the solarus. In: Language Engineering for Model-Driven Software Development. Volume 04101 of Dagstuhl Seminar Proceedings., Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI) (2004)

[26] Mahr, B.: Ein modell des modellseins. ein beitrag zur aufklärung des modellbegriffs. In: Modelle, Peter Lang Verlag (2008)

[27] Muller, P.A., Fondement, F., Baudry, B.: Modeling modeling. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009). Volume 5795 of Lecture Notes in Computer Science., Springer-Verlag (2009) 2–16

[28] Muller, P.A., Fondement, F., Baudry, B., Combemale, B.: Modeling modeling modeling. Software and Systems Modeling (2010) 1–13–13

[29] Mens, T., Czarnecki, K., Gorp, P.V.: A taxonomy of model transformation. In: Proceedings of the Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development", Internationales Begegnungs- und Forschungszentrum (IBFI) (2005)

[30] Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Workshop on Generative Techniques in the Context of Model-Driven Architecture (OOPSLA 2003), ACM Press (2003)

[31] Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical definition of in-place transformations in the eclipse modeling framework. In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), Springer-Verlag (2006) 425–439

[32] Hoxmeier, J.A., Dicesare, C.: System response time and user satisfaction: An experimental study of browser-based applications. In: Proceedings of the Association of Information Systems Americas Conference (AMCIS 2000), Academic Publishing Limited (2000) 10–13

[33] Maes, P.: Concepts and experiments in computational reflection. In: Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA 1987), ACM Press (1987) 147–155

[34] Lehmann, G., Blumendorf, M., Trollmann, F., Albayrak, S.: Meta-modeling runtime models. In: Models in Software Engineering - Workshops and Symposia at

MODELS 2010, Reports and Revised Selected Papers. Volume 6627 of Lecture Notes in Computer Science., Springer-Verag (2010) 209–223

[35] Ehrig, H., Golas, U., Hermann, F.: Categorical Frameworks for Graph Transformation and HLR Systems based on the DPO Approach. Bulletin of the European Association for Theoretical Computer Science **102** (2010) 111–121

[36] Lack, S., Sobocinski, P.: Adhesive categories. In: Proceedings of the 7th International Conference on Foundations of Software Science and Computation Structures, Springer-Verlag (2004) 273–288

[37] Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive high-level replacement categories and systems. In: Proceedings of the International Conference on Graph Transformation (ICGT2004), Springer-Verlag (2004) 144–160

[38] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer-Verlag (2006)

[39] Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science **109**(1&2) (1993) 181–224

[40] Gabriel, K., Braatz, B., Ehrig, H., Golas, U.: Finitary m-adhesive categories - unabridged version. Technical Report 2010/12, Technical University Berlin (2010)

[41] de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance: Long version. Technical report, Theoretical Computer Science (2005)

[42] Calvary, G., Coutaz, J., Dâassi, O., Balme, L., Demeure, A.: Towards a new generation of widgets for supporting software plasticity: The "comet". In: Proceedings of the 2004 international conference on Engineering Human Computer Interaction and Interactive Systems (EHCI-DSVIS 2004). Volume 3425 of Lecture Notes in Computer Science., Springer-Verlag (2004) 306–324

[43] Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal analysis and verification of self-healing systems. In: Proceedings of the International Conference on Fundamental Aspects of Software Engineering (FASE 2010). Volume 6013 of Lecture Notes in Computer Science., Springer-Verlag (2010) 139–153

[44] Badr, N., Reilly, D.: A conflict resolution control architecture for self-adaptive software. In: Proceedings of the International Workshop on Architecting Dependable Systems (WADS 2002), IEEE Computer Society (2002)

[45] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Souchon, N., Bouillon, L., Florins, M., Vanderdonckt, J.: Plasticity of user interfaces: A revised reference framework. In: Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design (TAMODIA 2002), INFOREC Publishing House Bucharest (2002) 127–134

[46] Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of the 28th international conference on Software engineering (ICSE 2006), ACM Press (2006) 371–380

[47] Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems (WOSS 2004), ACM Press (2004) 28–33

[48] Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, IBM (2001)

[49] Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. Autonomous and Adaptive Systems **1**(2) (2006) 223–259

[50] IBM Corporation: An Architectural Blueprint for Autonomic Computing. IBM Corporation (2006)

[51] Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer **36**(1) (2003) 41–50

[52] Müller, H., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Proceedings of the Second International Workshop on Ultra-large-scale Software-intensive Systems (ULSSIS 2008), ACM Press (2008) 23–26

[53] Hebig, R., Giese, H., Becker, B.: Making control loops explicit when architecting self-adaptive systems. In: Proceedings of the Second International Workshop on Self-organizing Architectures (SOAR 2010), ACM Press (2010) 21–28

[54] López-Jaquero, V., Vanderdonckt, J., Simarro, F.M., Gonzlez, P.: Towards an extended model of user interface adaptation: The isatine framework. In: Proceedings on the conference on Engineering Interactive Systems (EIS 2007). Volume 4940 of Lecture Notes in Computer Science., Springer-Verlag (2007) 374–392

[55] Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous Adaptive Systems **4**(2) (2009) 1–42

[56] Hellerstein, J.L.: Self-managing systems: A control theory foundation. In: Proceedings of the 29th Annual IEEE Conference on Local Computer Networks (LCN 2004), IEEE Computer Society (2004) 708

[57] Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14**(3) (1999) 54–62

[58] Laddaga, R.: Active software. In: Proceedings of the First International Workshop on Self-Adaptive Software (IWSAS 2000), Springer-Verlag (2000) 11–26

[59] Kephart, J.O., Walsh, W.E.: An artificial intelligence perspective on autonomic computing policies. In: POLICY, IEEE Computer Society (2004) 3–12

[60] Hussein, M., Han, J., Colman, A.: Specifying and verifying the context-aware adaptive behaviour of software systems. Technical Report C3-516-03, Swinburne University of Technology, Faculty of Information and Communication Technologies (FICT) (2010)

[61] Balasubramanian, S., Desmarais, R., Müller, H.A., Stege, U., Venkatesh, S.: Characterizing problems for realizing policies in self-adaptive and self-managing systems. In: SEAMS, ACM Press (2011) 70–79

[62] Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994), Springer-Verlag (1995) 151–163

[63] Ehrig, H., Orejas, F., Prange, U.: Categorical foundations of distributed graph transformation. In: Proceedings of the Third international conference on Graph Transformations (ICGT 2006), Springer-Verlag (2006) 215–229

[64] Galvao, I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), IEEE Computer Society (2007) 313–324

[65] Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Transactions on Software Engineering **27**(1) (2001) 58–93

[66] Jouault, F.: Loosely coupled traceability for atl. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Springer-Verlag (2005) 29–37

[67] Falleri, J.R., Huchard, M., Nebut, C.: Towards a traceability framework for model transformations in kermeta. In: Proceedings of the 2nd Workshop on Traceability (ECMDA-TW 2006). (2006)

[68] Sottet, J.S., Favre, J.M.: Mapping model: A first step to ensure usability for sustaining user interface plasticity. In: Proceedings of the 2nd International Workshop on Model Driven Developement of Advanced User Interfaces (MDDAUI 2006), CEUR-WS.org (2006) 214

[69] Sottet, J.S., Calvary, G., Coutaz, J., Favre, J.M.: A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces. In: Engineering Interactive Systems. Volume 4940 of Lecture Notes in Computer Science., Springer-Verlag (2008) 140–157

[70] Wolfe, C., Graham, T.C., Phillips, W.G.: An incremental algorithm for high-performance runtime model consistency. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Springer-Verlag (2009) 357–371

[71] Diskin, Z.: Mathematics of uml - making the odysseys of uml less dramatic. In: Practical Foundations of Business System Specifications, Springer-Verlag (2003) 145–178

[72] Del Fabro, M.D., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: Amw: A generic model weaver. Proceedings of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles (2005)

[73] Besova, G., Walther, S., Wehrheim, H., Becker, S.: Weaving-based configuration and modular transformation of multi-layer systems. In: 15th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2012). Volume 7590 of Lecture Notes in Computer Science., Springer-Verlag (2012) 776–792

[74] Hermann, F., Ehrig, H., Taentzer, G.: A typed attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams. In: Proceedings of the International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006). Volume 211 of Electronic Notes in Theoretical Computer Science., Elsevier Science (2008) 261–269

[75] Maximova, M., Ehrig, H., Ermel, C.: Formal relationship between petri net and graph transformation systems based on functors between m-adhesive categories. Electronic Communications of the EASST **40** (2010)

[76] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. Fundamentae Informatica **74**(1) (2006) 31–61

[77] Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: Conflict detection for model versioning based on graph modifications. In: Proceeding of the Fifth International Conference on Graph Transformation (ICGT 2010), Springer-Verlag (2010) 171–186

[78] Trollmann, F., Blumendorf, M., Schwartze, V., Albayrak, S.: Formalizing model consistency based on the abstract syntax. In: Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems (EICS 2011), ACM Press (2011) 79–84

[79] Jurack, S., Taentzer, G.: A component concept for typed graphs with inheritance and containment structures. In: Proceedings of the 5th International Conference on Graph Transformations (ICGT 2010), Springer-Verlag (2010) 187–202

[80] Jurack, S., Taentzer, G.: Transformation of typed composite graphs with inheritance and containment structures. Fundama Informaticae **118**(1-2) (2012) 97–134

[81] Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A formalisation of constraint-aware model transformations. In: Proceedings of the conference on Fundamental

Approaches to Software Engineering (FASE 2010). Volume 6013 of Lecture Notes in Computer Science., Springer-Verlag (2010) 13–28

[82] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. 2 edn. Addison-Wesley Longman Publishing Co., Inc. (2003)

[83] ISO/IEC JTC1/SC34: Information technology – Document Schema Definition Languages (DSDL) – Part 3: Rule-based validation – Schematron (2006)

[84] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2012)

[85] Sottet, J.S., Calvary, G., Favre, J.M.: Towards mapping and model transformation for consistency of plastic user interfaces (2006)

[86] Rensink, A.: Representing first-order logic using graphs. In: Proceedings of the International Conference on Graph Transformations (ICGT 2004). Volume 3256 of Lecture Notes in Computer Science., Springer-Verlag (2004) 319–335

[87] Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Synthesis of ocl pre-conditions for graph transformation rules. In: Proceedings of the International Conference on Model Transformation (ICMT 2010), Springer-Verlag (2010) 45–60

[88] Kuhlmann, M., Gogolla, M.: From uml and ocl to relational logic and back. In: Proceedings of the international conference on Model Driven Engineering Languages and Systems (MoDELS 2012). Volume 7590 of Lecture Notes in Computer Science., Springer-Verlag (2012) 415–431

[89] Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted ocl constraints into graph constraints for generating meta model instances by graph grammars. Electronic Notes in Theoretical Computer Science **211** (2008) 159–170

[90] Malý, J., Nečaský, M.: When grammars do not suffice: Data and content integrity constraints verification in xml through a conceptual model. In: Proceedings of the Ninth Asia-Pacific Conference on Conceptual Modelling - Volume 143 (APCCM 2013), Australian Computer Society, Inc. (2013) 21–30

[91] He, Y.: Comparison of the modeling languages alloy and uml. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2006), CSREA Press (2006) 671–677

[92] Jouault, F., Kurtev, I.: Transforming models with atl. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science., Springer-Verlag (2006) 128–138

[93] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All about maude - a high-performance logical framework, how to specify, program and verify systems in rewriting logic. In: All About Maude. Volume 4350 of Lecture Notes in Computer Science., Springer-Verlag (2007)

[94] OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0. http://www.omg.org/spec/QVT/1.0/PDF/ (2008)

[95] Kay, M.: XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer). 4 edn. Wrox Press Ltd. (2008)

[96] Patrascoiu, O.: YATL: Yet another transformation language. In: Proceedings of the 1st European MDA Workshop (MDA-IA). (2004) 83–90

[97] Braun, P., Marschall, F.: Transforming object oriented models with BOTL. Electronic Notes in Theoretical Computer Science **72**(3) (2003) 103 – 117

[98] Becker, B., Giese, H.: Modeling of correct self-adaptive systems: a graph transformation system based approach. In: Proceedings of the 5th international conference on soft computing as transdisciplinary science and technology (CSTST 2008), ACM Press (2008) 508–516

[99] Troya, J., Vallecillo, A.: A rewriting logic semantics for atl. Journal of Object Technology **10** (2011) 5: 1–29

[100] Giandini, R.S., Pons, C., Prez, G.: A two-level formal semantics for the qvt language. In: Memorias de la XII Conferencia Iberoamericana de Software Engineering (CIbSE 2009). (2009) 73–86

[101] Peltier, M., Bézivin, J.J., Guillaume, G.: Mtrans: A general framework, based on xslt, for model transformations. In: Proceedings of the Workshop on Transformations in UML (WTUML 2001). (2001)

[102] Bex, G.J., Maneth, S., Neven, F.: A formal model for an expressive fragment of xslt. In: Computational Logic. Volume 1861 of Lecture Notes in Computer Science., Springer-Verlag (2000) 1137–1151

[103] Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electronic Notes in Theoretical Computer Science **127**(3) (2005) 113–128

[104] Mens, T., Straeten, R.V.D., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), Springer-Verlag (2006) 200–214

[105] Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation (icse 2002). In: Proceedings of the 24th International Conference on Software Engineering, ACM Press (2002) 105–115

[106] Ehrig, H., Ermel, C., Taentzer, G.: A formal resolution strategy for operation-based conflicts in model versioning using graph modifications. In: Proceedings of

the 14th International Conference on Fundamental Approaches to Software Engineering (FASE 2011), Springer-Verlag (2011) 202–216

[107] Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A category-theoretical approach to the formalisation of version control in mde. In: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009), Springer-Verlag (2009) 64–78

[108] Cicchetti, A., Ruscio, D., Pierantonio, A.: Managing model conflicts in distributed development. In: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (MoDELS 2008), Springer-Verlag (2008) 311–325

[109] Goedicke, M., Meyer, T., Taentzer, G.: Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies. In: 4th IEEE International Symposium on Requirements Engineering (RE '99), IEEE Computer Society (1999) 92–99

[110] Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logics to maintain consistency between UML models. In: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Springer-Verlag (2003) 326–340

[111] Coutaz, J.: Meta-user interfaces for ambient spaces. In: Task Models and Diagrams for Users Interface Design. Volume 4385 of Lecture Notes in Computer Science. Springer-Verlag (2007) 1–15

[112] Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn (2007)

[113] Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010). Volume 6394 of Lecture Notes in Computer Science., Springer-Verlag (2010) 121–135

[114] Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental model synchronization for efficient run-time monitoring. In: Proceedings of the 4th International Workshop on Models@run.time. Volume 509 of CEUR Workshop Proceedings., CEUR-WS.org (2009) 1–10 (best paper).

[115] Bull, R.I.: Model Driven Visualization: Towards A Model Driven Engineering Approach For Information Visualization. PhD thesis, University of Victoria Canada (2008)

[116] Object Management Group: Meta object facility (mof) 2.0 core specification (2003)

[117] Beyer, M.: AGG An Algebraic Graph System, User Manual. Technical University of Berlin, Department of Computer Science. (1993)

[118] Biermann, E., Ermel, C., Taentzer, G.: Precise semantics of EMF model transformations by graph transformation. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008). Volume 5301 of Lecture Notes in Computer Science., Springer-Verlag (2008) 53–67

[119] Shneiderman, B.: Designing the user interface: strategies for effective human-computer interaction. Addison-Wesley Longman Publishing (1986)

Appendices

A. Existing Formal Definitions

In this appendix we list the formal definitions for the conceptual framework.

A.1. Definition of an $\mathcal{M}$-Adhesive Category

This section contains the definition of an $\mathcal{M}$-Adhesive Category . The definitions in this section are taken from [35]. An $\mathcal{M}$-Adhesive Category is defined as follows:

Definition 24 ($\mathcal{M}$-Adhesive Category (Definition 2.4 from [35]))**.** *An $\mathcal{M}$-adhesive category $(C, \mathcal{M})$ is a short notation for a vertical weak adhesive HLR category. The corresponding vertical weak van Kampen (VK) squares are called $\mathcal{M}$-VK squares.*

According to this definition an M-adhesive category is a vertical weak adhesive HLR category. A vertical weak adhesive HLR category is defined as follows:

Definition 25 (Vertical Weak Adhesive HLR Category (Based on Def. 2.3 and 2.1 in [35]))**.** *Given a PO-PB compatible class $\mathcal{M}$ of monomorphisms in a category C then $(C, \mathcal{M})$ is called a vertical weak adhesive HLR category if pushouts in C along $m \in \mathcal{M}$ exist and they are vertical weak VK squares, i.e. the van Kampen property is only required for commutative cubes, where all vertical morphisms (a, b, c, d) are in $\mathcal{M}$.*

For all commutative cubes with the given pushout in the bottom and the back squares being pullbacks (see Figure A.1) the VK property is the following equivalence: The top square is a pushout if and only if the front squares are pullbacks.

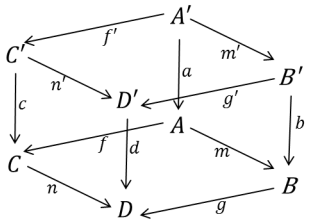


Figure A.1.: Van Campen Cube for the Van Campen Property.

A.2. Graph Transformation Production and Transformation

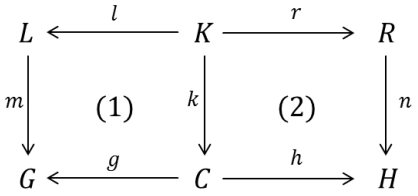
In this section we give the definition of a graph transformation production. The application of a graph transformation production with a specific match morphism implies

a direct transformation. We also give the definition of a direct transformation. The definitions are taken from [38]. A graph transformation production is defined as follows:

Definition 26 (Production (Definition 5.1 from [38]))**.** *Given a (weak) adhesive HLR category $(C, \mathcal{M})$, a production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ (also called rule) consists of three objects L, K and R, called the left-hand side, gluing object and right-hand side, respectively, and morphisms $l : K \to L$ and $r : K \to R$ with $l, r \in \mathcal{M}$.*

A transformation is defined as follows:

Definition 27 (Transformation (Part of Definition 5.2 from [38]))**.** *Given a production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and an object G with a morphism $m : L \to G$, called a match, a direct transformation $G \overset{p,m}{\Rightarrow} H$ from G to an object H is given by the following diagram, where (1) and (2) are pushouts:*



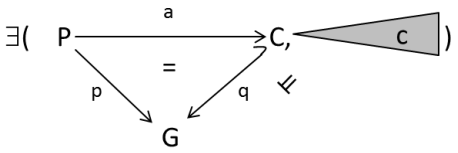
A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of direct transformations is called a transformation and is denoted by $G_0 \overset{*}{\Rightarrow} G_n$. For $n = 0$ we have the identical transformation $G_0 \overset{id}{\Rightarrow} G_0$, i.e. $g = g = id_G$.

A.3. Definition of Nested Conditions

This section contains the definition of nested conditions. This definition has been taken from [19]. It is as follows:

Definition 28 (Nested Conditions (Definition 4 from [19]))**.** *A nested condition over an object P is of the form true or $\exists(a, c)$, where $a : P \to C$ is a morphism and c is a condition over C. Moreover, Boolean formulas over conditions over P yield conditions over P, i.e., $\neg c$ and $\wedge_{j \in J} c_j$ are (Boolean) conditions over P, where J is an index set and $c, (cj)_{j \in J}$ are conditions over P. Additionally, $\exists a$ abbreviates $\exists(a, true)$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$, $false$ abbreviates $\neg true$, $\vee_{j \in J} c_j$ abbreviates $\neg \wedge_{j \in J} \neg c_j$ and $c \Rightarrow d$ abbreviates $\neg c \vee d$.*

Every object and morphism satisfies true. A morphism p satisfies a condition $\exists(a, c)$, if there exists a morphism q in M such that $q \circ a = p$ and q satisfies c.



An object G satisfies a condition $\exists(a, c)$, if the condition is over the initial object I and the initial morphism $i_G : I \to G$ satisfies the condition. The satisfaction of conditions by objects and morphisms is extended onto Boolean conditions in the usual way. We write $G \models c$ resp. $p \models c$ to denote that the object G resp. the morphism p satisfies c. Two conditions c and c_0 are equivalent, denoted by $c \equiv c_0$, if, for all morphisms p, $p \models c$ iff $p \models c_0$.

A.4. Restrictions of $\mathcal{M}$-Adhesive Categories

This section gives the definition of two restrictions of $\mathcal{M}$-adhesive categories. The definitions are taken from [40]. They are as follows:

Definition 29 (Finite Object and Finitary $\mathcal{M}$-adhesive category (Def. 2 in [40]))**.** *An object A in an $\mathcal{M}$-adhesive category (C, M) is called finite if A has finitely many $\mathcal{M}$-subobjects. An $\mathcal{M}$-adhesive category (C, M) is called finitary, if each object $A \in C$ is finite.*

Definition 30 ($\mathcal{M}$-Initial Object (Definition 1 in [40]))**.** *An initial object I in (C, M) is called $\mathcal{M}$-initial if for each object $A \in C$ the unique morphism $i_A : I \to A$ is in $\mathcal{M}$.*

A.5. Independence and the Local Church Rosser Theorem

In this section we describe parallel and sequential independence and the Local Church Rosser Theorem. The definitions and the theorem are taken from [38]. The definitions of parallel and sequential independence are as follows:

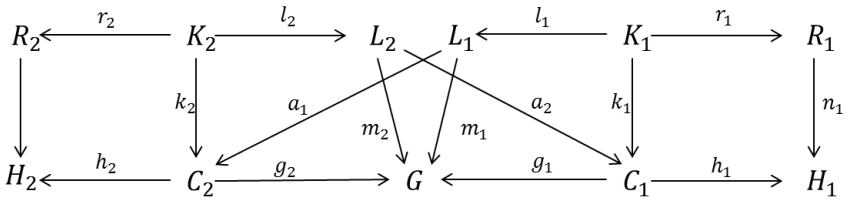


Figure A.2.: Morphisms for parallel independence.

Definition 31 (Parallel Independence (See Definition 5.9 in [38]))**.** *Two direct transformations $G \overset{p_1,m_1}{\Rightarrow} H_1$ and $G \overset{p_2,m_2}{\Rightarrow} H_2$ are parallel independent if there exist morphisms $a_1 : L_1 \to C_2$ and $a_2 : L_2 \to C_1$ such that $g_2 \circ a_1 = m_1$ and $g_1 \circ a_2 = m_2$*

Definition 32 (Sequential Independence (See Definition 5.9 in [38]))**.** *Two direct transformations $G \overset{p_1,m_1}{\Rightarrow} H_1 \overset{p_2,m_2}{\Rightarrow} H_2$ are sequential independent if there exist morphisms $a_1 : L_2 \to C_1$ and $a_2 : R_1 \to C_2$ such that $h_2 \circ a_1 = m_2$ and $g_2 \circ a_2 = n_1$*

The Local Church-Rosser Theorem is as follows:

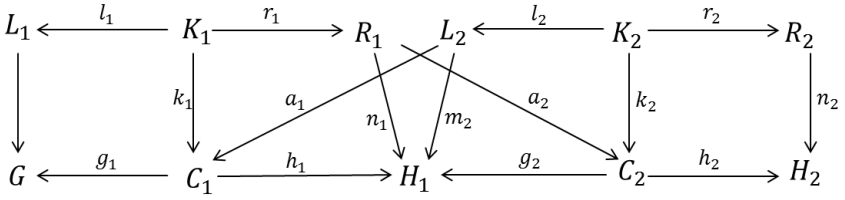


Figure A.3.: Morphisms for sequential independence.

Theorem 11 (Local Church-Rosser Theorem (See Theorem 5.12 in [38]))**.** *Given an adhesive HLR system AHS and two parallel independent direct transformations* $G \overset{p1,m1}{\Rightarrow} H_1$ *and* $G \overset{p2,m2}{\Rightarrow} H_2$ *there are an object* M *and there are direct transformations* $H_1 \overset{p2,m2'}{\Rightarrow} M$ *and* $H_2 \overset{p1,m1'}{\Rightarrow} M$ *such that* $G \overset{p1,m1}{\Rightarrow} H_1 \overset{p2,m2'}{\Rightarrow} M$ *and* $G \overset{p2,m2}{\Rightarrow} H_2 \overset{p1,m1'}{\Rightarrow} M$ *are sequentially independent.*

Given two sequentially independent direct transformations $G \overset{p1,m1}{\Rightarrow} H_1 \overset{p2,m2}{\Rightarrow} M$ *there are an object* H_2 *and direct transformations* $G \overset{p2,m2'}{\Rightarrow} H_2 \overset{p1,m1'}{\Rightarrow} M$ *such that* $G \overset{p1,m1}{\Rightarrow} H_1$ *and* $G \overset{p2,m2'}{\Rightarrow} H_2$ *are parallel independent.*

A.6. Parallel Production and Parallelism Theorem

In this section we provide the definition of parallel productions and the parallelism theorem. The Definitions and the theorem are taken from [38]. A parallel production is defined as follows:

Definition 33 (Parallel Production and Transformation (Definition 5.16 in [38]))**.** *Let* $AHS = (C, \mathcal{M}, P)$ *be an adhesive HLR System, where* $(C, \mathcal{M})$ *has binary coproducts compatible with* $\mathcal{M}$*. Given two productions* $p_1 = (L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1)$ *and* $p_2 = (L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2)$ *the parallel production* $p_1 + p_2$ *is defined by the coproduct constructions over the corresponding objects and morphisms:*

$p_1 + p_2 = (L_1 + L_2 \overset{l_1+l_2}{\leftarrow} K_1 + K_2 \overset{r_1+r_2}{\rightarrow} R_1 + R_2)$

The application of a parallel production is called a parallel direct transformation, or parallel transformation for short.

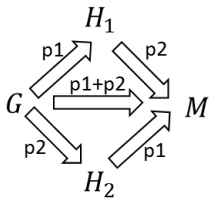
The parallelism theorem is as follows:

Theorem 12 (Parallelism Theorem (Theorem 5.18 in [38]))**.** *Let* $AHS = (C, \mathcal{M}, P)$ *be an adhesive HLR System, where* $(C, \mathcal{M})$ *has binary coproducts compatible with* $\mathcal{M}$*.*

1. ***Synthesis.*** *Given a sequentially independent direct transformation sequence* $G \Rightarrow H_1 \Rightarrow M$ *via productions* p_1 *and* p_2*, then there is a construction leading to a parallel transformation* $G \Rightarrow M$ *via the parallel production* $p_1 + p_2$*, called a synthesis construction.*

2. ***Analysis.*** *Given a parallel transformation $G \Rightarrow M$ via $p_1 + p_2$, then there is a construction leading to two sequentially independent transformation sequences $G \Rightarrow H_1 \Rightarrow M$ via p_1 and p_2 and $G \Rightarrow H_2 \Rightarrow M$ via p_2 and p_1, called an analysis construction.*

3. ***Bijective correspondence.*** *The synthesis and analysis constructions are inverse to each other up to isomorphism:*



A.7. Transformation of Application Conditions

In this section we give the main theorems for the transformation of a global nested condition to right application conditions and from right application conditions to left application conditions. The theorems are taken from [19]. The definitions of the construction operations can be found in this paper.

The theorems are as follows:

Theorem 13 (Transformation of Constraints to Right Application Conditions (Theorem 5 in [19]))**.** *Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a weak adhesive HLR category with $\mathcal{M}$-initial object I and epi-$\mathcal{M}$-factorization. There is a transformation A such that, for all conditions c over I, all rules p with righthand side R, and all morphisms $m^* : R \to H$.*

$$m^* \models A(p, c) \Leftrightarrow H \models c.$$

In this thesis we use $rightAC(p, c)$ to denote $A(p, c)$.

Theorem 14 (Transformation of Application Conditions (Theorem 6 in [19]))**.** *Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a weak adhesive HLR category with $\mathcal{M}$-initial object. There is a transformation L such that, for every rule p, every right application condition c for p, and every direct derivation $G \overset{p,m}{\Rightarrow} H$ with comatch n,*

$$m \models L(p, c) \Leftrightarrow n \models c.$$

In this thesis we use $leftAC(p, c)$ to denote $L(p, rightAC(c))$ for a nested condition c and a production p.

A.8. Attributed Typed Graphs with Inheritance

In this section we provide the definition of attributed typed graphs with node type inheritance. For this the fundamental definitions of graphs, E-graphs, attributed graphs and attributed typed graphs are given. The definitions are taken from [41].

The definitions are as follows:

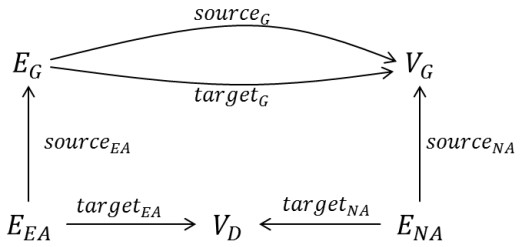
Definition 34 (Graph (Definition 1 in [41]))**.** *A graph $G = (V, E, s, t)$ consists of a set V of vertices (also called nodes), a set E of edges and the source and target functions $s, t : E \to V$.*

Definition 35 (E-Graph and E-Graph Morphism (Definition 6 in [41]))**.** *An E-Graph G with $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}})$ consists of sets:*

- *V_G and V_D called graph and data nodes (or vertices) respectively;*
- *E_G, E_{NA}, E_{EA} called graph, node attribute and edge attribute edges respectively.*

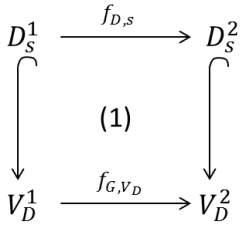
and source and target functions

- *$source_G : E_G \to V_G, target_G : E_G \to V_G$ for graph edges;*
- *$source_{NA} : E_{NA} \to V_G, target_{NA} : E_{NA} \to V_D$ for node attribute edges;*
- *$source_{EA} : E_{EA} \to E_G, target_{EA} : E_{EA} \to V_D$ for edge attribute edges.*



Let $G^k = (V_G^k, V_D^k, E_G^k, E_{NA}^k, E_{EA}^k, (source_j^k, target_j^k)_{j \in \{G,NA,EA\}})$ for $k = 1, 2$ be two E-graphs. An E-Graph morphism $f : G^1 \to G^2$ is a tuple (f_{V_G}, f_{V_D}, f_{E_G}, $f_{E_{NA}}$, $f_{E_{EA}}$) with $f_{V_i} : V_i^1 \to V_i^2$ and $f_{E_j} : E_j^1 \to E_j^2$ for $i \in \{G, D\}, j \in \{G, NA, EA\}$ such that f commutes with all source and target functions, e.g. $f_{V_G} \circ source_G^1 = source_G^2 \circ f_{E_G}$

Definition 36 (Attributed Graph and Attributed Graph Morphism (Definition 7 in [41]))**.** *Let $DSIG = (S_D, OP_D)$ be a data signature with attribute value sorts $S'_D \subseteq S_D$. An attributed graph $AG = (G, D)$ consists of an E-graph G together with a DSIG-Algebra D such that $\biguplus_{s \in S'_D} D_S = V_D$. For two attributed graphs $AG^i = (G^i, D^i)$ with $i = 1, 2$ an attributed graph morphism $f : AG^1 \to AG^2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^1 \to G^2$ and an algebra homomorphism $f_D : D^1 \to D^2$ such that (1) commutes for all $s \in S'_D$.*



Definition 37 (Typed Attributed Graph and Typed Attributed Graph Morphism (Definition 8 in [41]))**.** *Given a data signature $DSIG$, an attributed type graph is an attributed graph $ATG = (TG, Z)$, where Z is the final $DSIG$-algebra.*

A typed attributed graph (AG, t) over ATG consists of an attributed graph AG together with an attributed graph morphism $t : AG \to ATG$.

A typed attributed graph morphism $f : (AG^1, t^1) \to (AG^2, t^2)$ is an attributed graph morphism $f : AG^1 \to AG^2$ such that $t^2 \circ f = t^1$.

Definition 38 (Attributed Type Graph with Inheritance (Definition 9 in [41]))**.** *An attributed type graph with inheritance $ATGI = (TG, Z, I, A)$ consists of an attributed type graph $ATG = (TG, Z)$, where TG is an E-graph $TG = (TG_{V_G}, TG_{V_D}, TG_{E_G}, TG_{E_{NA}}, TG_{E_{EA}}, source_i, target_i)_{i \in \{G,NA,EA\}})$ with $TG_{V_D} = S'_D$ and Z the final $DSIG$-algebra and an inheritance graph $I = (I_V, I_E, s, t)$, with $I_V = TG_{V_G}$, and a set $A \subseteq I_V$, called abstract nodes.*

B. Proofs

This appendix contains proofs for the main theorems and definitions in Section 6. The proofs are ordered by theorem.

B.1. Theorem 1 - The Category TypedGraphDiagrams$^D_{TG}$ is a Finitary $\mathcal{M}$-Adhesive Category with $\mathcal{M}$-Initial object

This section gives proof that the category $TypedGraphDiagrams^D_{TG}$ is an $\mathcal{M}$-adhesive category, is finitary and contains an $\mathcal{M}$-initial object. This is stated in Theorem 1.

The proof is as follows:

Proof. The proof consists of four parts. First, it is shown that the category of typed graph diagrams $TypedGraphDiagrams_{TG}$ with a suitable class of monomorphisms is an $\mathcal{M}$-adhesive category. Based on that it is shown that the category of algebra-restricted typed graph diagrams $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ is $\mathcal{M}$-adhesive. Subsequently, we proof finitariness and the existence of an $\mathcal{M}$-initial object. The four proofs are as follows:

1. $(TypedGraphDiagrams_{TG}, \mathcal{M}_{TG})$ is an $\mathcal{M}$-adhesive category.

   The category $TypedGraphDiagrams_{TG}$ is constructed via slice and functor category construction from the category $AGraphs$. This category is an $\mathcal{M}$-adhesive HLR category [38] with the class $\mathcal{M}_{AGraphs}$ of injective morphisms that contain isomorphisms on their data components. According to [38] both constructions preserve the $\mathcal{M}$-adhesive properties. Accordingly, $TypedGraphDiagrams_{TG}$ with a class $\mathcal{M}_{TG}$ of natural transformations consisting of morphisms from $\mathcal{M}_{AGraphs}$ is an $\mathcal{M}$-adhesive HLR category and also an $\mathcal{M}$-adhesive category [35].

2. $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ is an $\mathcal{M}$-adhesive category.

   The category $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ is derived via two restrictions from the category $(TypedGraphDiagrams_{TG}, \mathcal{M}_{TG})$ (cf. Definition 9). On the one hand, the scheme of the diagram is restricted to contain a finite number of objects and morphisms. Since $(TypedGraphDiagrams_{TG}, \mathcal{M}_{TG})$ is $\mathcal{M} - adhesive$ for any scheme, this restriction has no influence on $\mathcal{M} - adhesiveness$. The second restriction is the restriction to objects and morphisms from the category $AGraphs^D$, which is a subcategory of $AGraphs$. In the remainder of this proof we show that the $\mathcal{M}$-adhesive properties hold despite this restriction with the class $\mathcal{M}$ that contains all monomorphisms.

   The class $\mathcal{M}$ contains all natural transformations that contain only monomorphisms in $AGraphs^D$ which also are monomorphisms in $AGraphs$ since $AGraphs^D$

is a subcategory of $AGraphs$. All morphisms in $AGraphs^D$ are isomorphic on their data type. Accordingly, all monomorphisms in $AGraphs^D$ are in $\mathcal{M}_{AGraphs}$. Thus, the natural transformations in $\mathcal{M}$ consist of morphisms from $\mathcal{M}_{AGraphs}$ and hence $\mathcal{M} \subset \mathcal{M}_{TG}$ and all morphisms in $TypedGraphDiagrams_{TG}$ that are contained in $\mathcal{M}_{TG}$ are also contained in $\mathcal{M}$.

$\mathcal{M}$ is closed under isomorphisms, composition, and decomposition, because this is valid for $\mathcal{M}_{TG}$.

Pushouts and pullbacks in $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ can be constructed in $(TypedGraphDiagrams_{TG}, \mathcal{M}_{TG})$. There, they are component-wise constructed in $AGraphs$. Since the pushout and pullback along attributed graph morphisms with identical data components yield morphisms with identical data components and objects with the same algebra the pushout and pullback yield objects and morphisms in the category $AGraphs^D$. Thus, the constructed pushout and pullback objects in $TypedGraphDiagrams_{TG}$ are also in $TypedGraphDiagrams^D_{TG}$. Since $\mathcal{M}_{TG}$ is preserved by pushouts and pullbacks along $\mathcal{M}_{TG}$ in the category $(TypedGraphDiagrams_{TG}, \mathcal{M}_{TG})$ and $\mathcal{M}$ contains all morphisms from this class that are contained in $TypedGraphDiagrams^D_{TG}$ the class $\mathcal{M}$ is also preserved by pushout and pullbacks along $\mathcal{M}$.

Finally, the weak Van Kampen Property of $(TypedGraphDiagrams_{TG}, \mathcal{M}_{TG})$ implies the weak Van Kampen Property of $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ since pushouts/pullbacks of morphisms that are in both categories are also pushouts/ pullbacks in the respective other category and $\mathcal{M} \subset \mathcal{M}_{TG}$.

3. $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ is a finitary $\mathcal{M}$-adhesive category.

   According to Definition 29 we need to show that all objects are finite to show that $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ is finitary

   The category $AGraphs^D$ is finitary according to Gabriel since the number of nodes, edges, node attribute edges and edge attribute edges are finite. Accordingly, all objects in this category are finite [40].

   A typed graph diagram A in $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ is finite if it only contains finitely many $\mathcal{M}$-subobjects, meaning there are only finitely many attributed typed graph diagrams A' (up to isomorphism) for which a morphism $m : A' \to A \in \mathcal{M}$ exists. The morphism m is a natural transformation that contains a morphism in $\mathcal{M}_{AGraphs}$ for each attributed graph a in A. Since $AGraphs^D$ is finitary there is only a finite number of such morphisms for each a.

   Since the scheme in $(TypedGraphDiagrams^D_{TG}, \mathcal{M})$ is assumed to have a finite number of nodes and edges there are only a finite number of attributed graphs in A. As stated above, for each A there is only a finite number of morphisms between attributed graphs. Accordingly, there are only finitely many possibilities to combine morphisms targeting the attributed graphs in A to morphisms m. Since there is only a finite number of such morphisms, A is finite.

This property holds for each typed graph diagram in A. Accordingly, the category $(TypedGraphDiagrams^{D}_{TG}, \mathcal{M})$ is a finitary $\mathcal{M}$-adhesive category.

4. $(TypedGraphDiagrams^{D}_{TG}, \mathcal{M})$ contains an $\mathcal{M}$-initial object.

   For this part of the proof we show that the empty diagram *init* in the category $TypedGraphDiagrams^{D}_{TG}$ is an $\mathcal{M}$-initial object.

   The empty diagram *init* is a diagram from $TypedGraphDiagrams^{D}_{TG}$ that only contains empty attributed graphs. The morphisms in *init* are isomorphisms for the algebra and data component and empty for all other components.

   To show that *init* is an $\mathcal{M}$-initial object we show that for any object A from $TypedGraphDiagrams^{D}_{TG}$ there is a morphism $m : init \to A$ in $\mathcal{M}$. This morphism is a natural transformation that is component-wise defined on the objects n of the scheme between the objects $init_n$ and m_n.

   For each n a morphism $m_n : init_n \to A_n$ can be defined that is empty for nodes, edges, attribute nodes and attribute edges and an identity on the algebra and data nodes. This morphism always exists. It is injective and thus a monomorphism.

   Since such a morphism exists for each node in the scheme, the morphism m can be constructed. It is compatible with the morphisms in the diagrams since all morphisms in *init* consist of isomorphisms and empty components. Since m is a natural transformation containing monomorphisms it is also a monomorphism and thus contained in $\mathcal{M}$.

   Such a morphism can be found for any object A and is always in $\mathcal{M}$. Since all components except for the data elements in the attributed graphs in *init* are empty and the mapping of the data elements is prescribed by the category no other morphism can exist. Thus, the morphism for each object A is unique. Accordingly, the object *init* is an $\mathcal{M}$-initial object of $(TypedGraphDiagrams^{D}_{TG}, \mathcal{M})$.

Since all four parts can be shown the theorem holds.

□

B.2. Theorem 2 - The Condition for the Existence of a Morphism for Attributed Graphs are Correct and Complete

This section contains proof that the conditions for the construction of a morphism as given in Definition 14 describe exactly all situations where a morphism exists. This is stated in Theorem 2 and illustrated in Figure B.1.

The proof is as follows

Proof. The proof consists of two directions which are proven separately. The first part proofs that a morphism can be constructed as described in the definition if the condi-

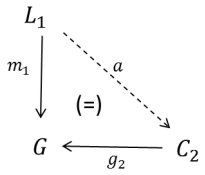


Figure B.1.: Illustration of the morphisms used in the proof for Theorem 2.

tions are fulfilled ($\Rightarrow$). The second part proofs that it is not possible to construct any commuting morphism if any of the conditions are not fulfilled ($\Leftarrow$).

- $\Rightarrow$

  For this direction it is assumed that all conditions given in Definition 14 hold it is proven that the morphism as described in this definition is well-defined in all components, fulfils all conditions on attributed graph morphisms and fulfils $m_1 = g_2 \circ a$.

  Well-Definedness:

  The following statements hold because g_2 and thus all components of this morphism are monomorphisms.

  1. f_{V_G}: if property one is fulfilled $f^{-1}_{V_G,g_2}$ is defined and unique for all nodes in $f_{V_G,m_1}(V_{G,L_1})$. Thus, the function $f^{-1}_{V_G,g_2} \circ f_{V_G,m_1}$ is well-defined.
  2. f_{V_D}: This component is defined as identity and thus is well-defined.
  3. f_{E_G}: if property two is fulfilled $f^{-1}_{E_G,g_2}$ is defined and unique for all edges in $f_{E_G,m_1}(V_{G,L_1})$. Thus, the function $f^{-1}_{E_G,g_2} \circ f_{E_G,m_1}$ is well-defined.
  4. $f_{E_{NA}}$: if property three is fulfilled $f^{-1}_{E_{NA},g_2}$ is defined and unique for all node attribute edges in $f_{E_{NA},m_1}(V_{G,L_1})$. Thus, the function $f^{-1}_{E_{NA},g_2} \circ f_{E_{NA},m_1}$ is well-defined.
  5. $f_{E_{EA}}$: if property four is fulfilled $f^{-1}_{E_{EA},g_2}$ is defined and unique for all edge attribute edges in $f_{E_{EA},m_1}(V_{G,L_1})$. Thus, the function $f^{-1}_{E_{EA},g_2} \circ f_{E_{EA},m_1}$ is well-defined.
  6. f_D: This component is defined as identity and thus is well-defined.

  Fulfilment of the conditions:

  1. *Commutation of the data components (diagram in Definition 36):*

     The morphism f_D as well as the morphism f_{V_D} are defined as identities and the set of data nodes and the algebra in L_1 and C_2 are the same. Thus, the diagram commutes.

  2. *Commutation of the morphism a with all source and target functions :*

For the source function $source_G$ the following holds:

$$
\begin{aligned}
& f_{V_G} \circ source_G^{L_1} \\
=& f_{V_G,g_2}^{-1} \circ f_{V_G,m_1} \circ source_G^{L_1} && \text{(Definition } f_{V_G}\text{)} \\
=& f_{V_G,g_2}^{-1} \circ source_G^{G} \circ f_{E_G,m_1} && \text{(Equation holds for } m_1\text{)} \\
=& f_{V_G,g_2}^{-1} \circ source_G^{G} \circ f_{E_G,g_2} \circ f_{E_G,g_2}^{-1} \circ f_{E_G,m_1} \\
& && (f_{E_G,g_2}^{-1} \circ f_{E_G,m_1} \text{ exists due to condition 1)} \\
=& f_{V_G,g_2}^{-1} \circ f_{V_G,g_2} \circ source_G^{C_2} \circ f_{E_G,g_2}^{-1} \circ f_{E_G,m_1} && \text{(Equation holds for } g_2\text{)} \\
=& source_G^{C_2} \circ f_{E_G,g_2}^{-1} \circ f_{E_G,m_1} && (f_{V_G,g_2}^{-1} \circ f_{V_G,g_2} \text{ is neutral)} \\
=& source_G^{C_2} \circ f_{E_G} && \text{(Definition } f_{E_G}\text{)}
\end{aligned}
$$

The proof for $target_G$ is analogous.

For the source function $source_{EA}$ the following holds:

$$
\begin{aligned}
& f_{V_D} \circ source_{EA}^{L_1} \\
=& id_{V_D} \circ source_{EA}^{L_1} && \text{(Definition of } f_{V_D}\text{)} \\
=& source_{EA}^{G} \circ f_{E_{EA},m_1} && \text{(Equation holds for } m_1\text{)} \\
=& source_{EA}^{G} \circ f_{E_{EA},g_2} \circ f_{E_{EA},g_2}^{-1} \circ f_{E_{EA},m_1} \\
& && (f_{E_{EA},g_2}^{-1} \circ f_{E_{EA},m_1} \text{ exists due to condition 4)} \\
=& id_{V_D} \circ source_{EA}^{C_2} \circ f_{E_{EA},g_2}^{-1} \circ f_{E_{EA},m_1} && \text{(Equation holds for } g_2\text{)} \\
=& source_{EA}^{C_2} \circ f_{E_{EA},g_2}^{-1} \circ f_{E_{EA},m_1} && (id_{V_D}) \text{ is neutral} \\
=& source_{EA}^{C_2} \circ f_{E_{EA}} && \text{(Definition } f_{E_{EA}}\text{)}
\end{aligned}
$$

The proof for $target_{EA}$, $source_{NA}$ and $target_{NA}$ is analogous.

The equation $m_1 = g_2 \circ a$ holds

this property is proven component-wise over the components in a.

1. **Components f_{V_D} and id_D:**

   Both morphisms are defined as the identity in m_1, g_2 and a and the result of the composition of identities is the identity. Thus, the property holds.

2. **Components $f_{V_G}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}}$:**

   For any f_x with $x \in \{V_G, E_G, E_{NA}, E_{EA}\}$ the following equation holds:

$$
\begin{aligned}
& f_{x,g_2} \circ f_x \\
=& f_{x,g_2} \circ f_{x,g_2}^{-1} \circ f_{x,m_1} && \text{(Condition for the respective component)} \\
=& f_{x,m_1} && (f_{x,g_2} \circ f_{x,g_2}^{-1} \text{ is neutral})
\end{aligned}
$$

Thus, the property also holds for these components.

Since the property holds for all components of a, it holds for a.

- $\Leftarrow$

  This direction is shown by indirect proof. Starting from the assumption that there is a morphism and one of the properties is not satisfied, we shows a contradiction. This needs to be proven for all properties. Since the properties are defined analogous on different components of the morphisms the proofs are also analogous. The proof for Condition 1 for component f_{V_G} is used as an example. Here the assumptions are as follows:

  1. There is a morphism a with $m_1 = g_2 \circ a$.
  2. Condition 1 is not fulfilled i.e. there is a node $v_1 \in V_{G,L_1}$ such that there does not exist a node $v_2 \in V_{G_{C_2}}$ with $f_{V_G,g_2}(v_2) = f_{V_G,m_1}(v_1)$

  According to Assumption 1 the equation $m_1 = g_2 \circ a$ holds and thus the equation holds component-wise for all components of a. Thus, for the component V_G the following holds: $f_{V_G,m_1} = f_{V_G,g_2} \circ f_{V_G,a}$. For the node v_1 from Assumption 2 this leads to $f_{V_G,m_1}(v_1) = f_{V_G,g_2}(f_{V_G,a}(v_1))$. This statement is in contradiction with Assumption 2 since $v_2 = f_{V_G}(v_1)$ fulfils the equation, although Assumption 2 states that no such node exists.

  An analogue contradiction can be shown for conditions two (component f_{E_G}), three (component $f_{E_{NA}}$) and four (component $f_{E_{EA}}$).

  Thus, if at least one condition is not fulfilled there is no morphism a with the above property.

- Both directions have been proven. Thus, the morphism a exists if and only if all conditions are fulfilled and Theorem 2 holds.

□

B.3. Theorem 3 - The Conditions for the Existence of a Morphism for Typed Graph Diagrams are Correct and Complete

This section contains proof for Theorem 3. This theorem states that the conditions for the existence of a morphism in typed graph diagrams are correct and complete. The conditions are given in Definition 15.

The proof is as follows.

Proof. We proof each direction separately. First, we prove that the construction of the morphism given in Definition 15 is possible when the condition is fulfilled and leads to a well-defined morphism that fulfils the required property ($\Rightarrow$). The second part proves that the conditions are fulfilled if a morphism exists ($\Leftarrow$).

- $\Rightarrow$

  The proof for this direction consists of two parts. First, we prove that, given the condition holds, the construction leads to a well-defined morphism. Second, we prove that the equation $m_1 = g_2 \circ a$ holds.

  Well-Definednes:

  For each object s from the scheme S the morphism $a(s)$ is defined as the construction of a morphism in attributed graphs from $m_1(s)$ and $g_2(s)$. This construction is well defined as the conditions on typed graph diagrams contain the conditions for the construction of $a(s)$ from $m_1(s)$ and $g_2(s)$ from Definition 14. As stated in Theorem 2 this construction leads to a well-defined morphism in attributed graphs.

  The family of such morphisms $a(s)$ forms a natural transformation. Each $a(s)$ is defined as $g_2^{-1}(s) \circ m_1(s)$ (Because $a(s)$ is component-wise defined in this way). Since g_2 and m_1 are natural transformations and thus commute with all morphisms in the diagram $a(s)$ also commutes. Thus, the morphism a is well-defined.

  The equation $m_1 = g_2 \circ a$ holds:

  This statement can also be proven component-wise over the attributed graph morphisms contained in a. For each object s from the scheme the constructed morphism $a(s)$ fulfils the following equation $m_1(s) = g_2(s) \circ a(s)$ because the conditions in graph diagrams contain the conditions in attributed graphs from Definition 14. The correctness of the construction in attributed graphs is stated in Theorem 2.

  Since each contained morphism $a(s)$ fulfils this equation the equation $m_1 = g_2 \circ a$ is also fulfilled.

- $\Leftarrow$

  This direction is proven via a contradiction. Based on the assumption that there is a morphism a but not all conditions are fulfilled a contradiction is proven.

  The assumptions are as follows:

1. There is a morphism a with $m_1 = g_2 \circ a$.
2. For at least one object s from S the condition in Definition 14 for the existence of a morphism x with $m_1(s) = g_2(s) \circ x$ is not fulfilled.

According to Assumption 2 the condition is not fulfilled for s. Accordingly, there is no morphism x that fulfils $m_1(s) = g_2(s) \circ x$ because the conditions on attributed graphs are correct and complete as stated in Theorem 2. However, the existence of the morphism a in typed graph diagrams with $m_1 = g_2 \circ a$ (Assumption 1) implicates the existence of a morphism $a(s)$ with $m_1(s) = g_2(s) \circ a(s)$. Accordingly, we can find a morphism $x = a(s)$ that fulfils the equation. This is a contradiction to the second assumption.

If the condition from Definition 14 is not fulfilled for at least one node of the scheme this contradiction can be proven. Accordingly, it has to be fulfilled for all nodes if there is a morphism a, which is the condition from Definition 15.

- Both directions have been proven. Thus, the morphism exists if and only if the conditions are fulfilled and Theorem 3 holds.

□

B.4. Theorem 4 - Equivalence of Conflict Detection Conditions in Attributed Graphs

This section contains the proof that the set-based condition for the existence of a morphism from Definition 16 correctly specifies when an according morphism exists. This is stated in Theorem 4.

The proof is as follows:

Proof. It has already been shown in Appendix B.2 that the conditions in Definition 14 correctly describe when a morphism exists. In order to show Theorem 4 this proof shows that the conditions in Definition 16 are equivalent to the conditions in Definition 14 applied to L_2, G and C_2.

In both definitions the conditions are defined component-wise for the components V_G, E_G, E_{NA} and E_{EA} of attributed graphs and their respective morphisms. Conditions 1, 2, 3 and 4 of each condition handle the same component and thus correspond to each other. All conditions in each definition are identical but formulated for different components. We proof the equivalence for the first pair of conditions for V_G. The other pairs can be proven analogous.

The proof consists of two directions. First, we show that the condition in Definition 14 implies the condition in Definition 16 ($\Rightarrow$). Subsequently, an implication in the other direction ($\Leftarrow$) is shown.

- $\Rightarrow$ For this direction we show that condition 1 from Definition 14 implies condition 1 from Definition 16. We proof this by indirect proof. We assume that condition 1

from Definition 14 holds and condition 1 from Definition 16 does not hold. From this we derive a contradiction.

The following is assumed:

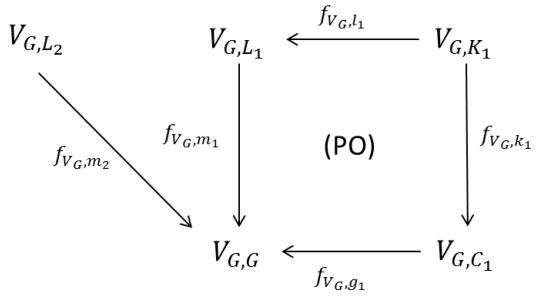
1. condition 1 in Definition 14 holds, meaning:

   $f^{-1}_{VG,g_1} \circ f_{V_G,m_2}$ is fully defined.

2. the set-based property does not hold, meaning:

   $$f_{V_G,m_2}(V_{G,L_2}) \cap (f_{V_G,m_1}(V_{G,L_1}) \backslash f_{V_G,m_1}(f_{V_G,l_1}(V_{G,K_1}))) \neq \emptyset$$

The morphisms l_1, k_1, m_1 and g_1 form a pushout as they constitute the first step of the application of one production application. Because of the component-wise construction of morphisms in attributed typed graphs the morphisms f_{V_G,l_1}, f_{V_G,k_1}, f_{V_G,m_1} and f_{V_G,g_1} also form a pushout. The elements in this pushout and the morphism f_{V_G,m_2} are shown in the following figure.



From assumption 2 it it can be derived that there is at least one element $v_G \in V_{G,G}$, and element $v_{L_2} \in V_{G,L_2}$ such that $f_{V_G,m_2}(v_{L_2}) = v_G$, an element $v_{L_1} \in V_{G,L_1}$ such that $f_{V_G,m_1}(v_{L_1}) = v_G$ and no element $v_{K_1} \in V_{G,K_1}$ with $f_{V_G,l_1}(v_{K_1}) = v_{L_1}$.

Because assumption 1 holds we know that there is a $v_{C_1} \in V_{G,C_1}$ with $f_{V_G,g_1}(v_{C_1}) = f_{V_G,m_2}(v_{L_2}) = v_G$.

In this situation the diagram spanned by the morphisms f_{V_G,l_1}, f_{V_G,k_1}, f_{V_G,m_1} and f_{V_G,g_1} cannot be a pushout as v_G is mapped from v_{L_1} and v_{C_1} but v_{L_1} is not targeted by f_{V_G,l_1}. This can be derived from the construction of pushouts in sets and the fact that pushouts are unique up to isomorphisms. In this construction the elements v_{L_1} and v_{C_1} would be mapped to different elements. Since this is not the case this diagram would not be isomorph to a constructed pushout and is thus not a pushout.

This shows a contradiction. Accordingly, the implication has to hold.

- $\Leftarrow$

  This direction is also shown by indirect proof. We assume that condition 1 from Definition 16 holds and condition 1 from Definition 14 does not hold. From these assumptions we show a contradiction, thus proving that condition 1 from Definition 14 has to hold when condition 1 from Definition 16 holds.

The assumptions are as follows:

1. The set based condition holds, meaning:

   $$f_{V_G,m_2}(V_{G,L_2}) \cap (f_{V_G,m_1}(V_{G,L_1}) \backslash f_{V_G,m_1}(f_{V_G,l_1}(V_{G,K_1})) = \emptyset$$

2. Condition 1 in Definition 14 does not hold, meaning:

   $f^{-1}_{VG,g_1} \circ f_{V_G,m_2}$ is not fully defined.

According to Assumption 2 there is at least one node $v_{L_2} \in V_{G,L_2}$ such that there is no $v_{C_1} \in V_{G,C_1}$ with $f_{V_G,g_1}(v_{C_1}) = f_{V_G,m_2}(v_{L_2})$.

$V_{G,G}$ and the morphisms f_{V_G,m_1} and f_{V_G,g_1} are the result of the pushout of morphisms f_{V_G,l_1}, f_{V_G,k_1}, f_{V_G,m_1} and f_{V_G,g_1} in sets. Accordingly, the morphisms f_{V_G,g_1} and f_{V_G,m_1} are jointly surjective. Thus, if $f_{V_G,m_2}(v_{L_2})$ is not targeted by the morphism f_{V_G,g_1} it has to be targeted by f_{V_G,m_1} and there is a $v_{L_1} \in V_{G,L_1}$ with $f_{V_G,m_1}(v_{L_1}) = f_{V_G,m_2}(v_{L_2})$.

According to Assumption 1 there is a $v_{K_1} \in V_{G,K_1}$ with $f_{V_G,l_1}(v_{K_1}) = (v_{L_1})$.

However, because the pushout with morphisms f_{V_G,l_1}, f_{V_G,k_1}, f_{V_G,m_1} and f_{V_G,g_1} is a commuting diagram, this means there is an element $v_{C_1} = f_{V_G,k_1}(v_{K_1}) \in V_{G,L_1}$ with $f_{V_G,g_1}(v_{C_1}) = f_{V_G,m_2}(v_{L_2})$. This shows a contradiction to assumption 2 which states that there is no node v_{C_1} with this property.

Since the assumptions lead to a contradiction Condition 1 in Definition 14 has to hold if Condition 1 in Definition 16 holds.

Both directions have been proven. Thus, the conditions are equivalent.

An equivalent proof exists for all other pairs of conditions. Thus, the conditions in both definitions are equivalent.

□

B.5. Theorem 5 - Equivalence of Conflict Detection Conditions on Typed Graph Diagrams

In this section we proof that the set-based condition for graph diagrams given in Definition 17 is equivalent to the condition given in Definition 15. This is stated in Theorem 5.

The proof is as follows:

Proof. The conditions from both definitions are as follows for all objects o in the scheme:

1. *Definition 15:* A morphism a_o with $m_1(o) = g_2(o) \circ a_o$ exists in the category **AGraphs** according to Definition 14.

2. *Definition 17:* The production $p_{2,o} = (O_{L_2}(o) \leftarrow O_{K_2}(0) \rightarrow O_{R_2}(o))$ with morphisms $l_2(o)$ and $r_2(o)$ and match $m_2(o)$ can be applied to $O_G(o)$ after the production $p_{1,o} = (O_{L_1}(o) \leftarrow O_{K_1}(0) \rightarrow O_{R_1}(o))$ with morphisms $l_1(o)$ and $r_1(o)$ and match $m_1(o)$ according to the conditions in Definition 16.

Both conditions iterate over all objects in the scheme and for each object rely on the condition for attributed graphs in Definitions 14 and 16 for the components of the graph diagrams and morphisms for this object in the diagram.

The equivalence of the conditions in Definitions 14 and 16 is stated in Theorem 4. Accordingly, the condition for each object of the scheme is equivalent.

Since the conditions for each object o in the scheme is equivalent the overall conditions are also equivalent.

□

B.6. Theorem 6 - Well-Definedness of the Maximum Graph and Supporting Lemmata

This section proofs that the maximum graph, as defined in Definition 18, can always be constructed under the given conditions. Before giving the main proof in Appendix B.6.2 we define and proof several lemmata regarding the relation of maximum and deletion production applications in Appendix B.6.1. These lemmata are used in the following the proof of the main theorem and in the following proofs.

B.6.1. Lemmata

The first lemma describes that the maximum and deletion production application constructed from the same production application are parallel independent and if both are applied lead to the same result as the original production application. This is stated in the following lemma:

Lemma 1 (Decomposition of production applications)**.** *Given a production application* $PA = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, m)$ *to an object* G*, the maximum production application* PA_{max} *and the deletion production application* PA_{del} *are parallel independent and* $apply(G, \{PA_{max}, PA_{del}\})$ *is isomorph to* $apply(G, \{PA\})$

The proof for this lemma is as follows:

Proof. Decomposition of production applications. In this proof we first show the parallel independence of maximum and deletion production applications. Subsequently, we show that their application leads to the same result as the application of PA.

The application of the relevant production applications is shown in Figure B.2. The application of PA is shown in the upper half and the parallel application of PA_{max} and PA_{del} is shown on the lower half.

The parallel independence of PA_{add} and PA_{del} requires the following morphisms [38]:

1. $a : L \to G$ with $id_G \circ a = m$
2. $b : K \to I$ with $u \circ b = m \circ l$

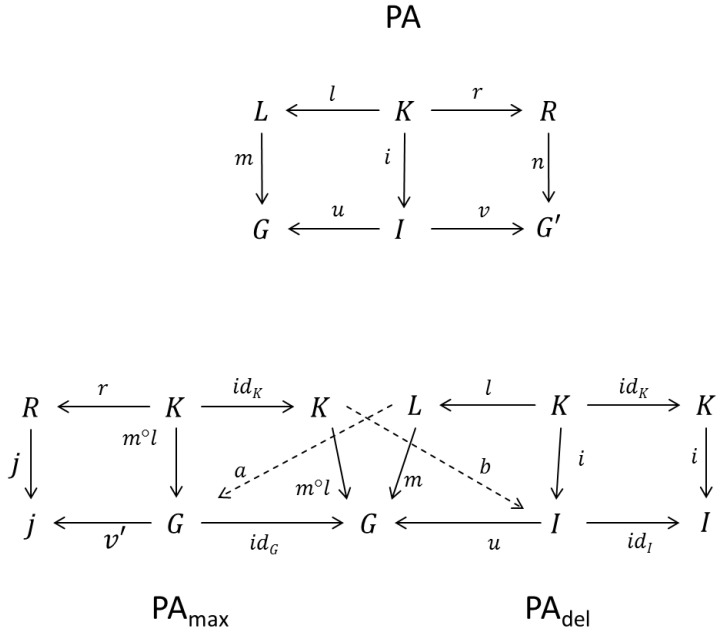


Figure B.2.: The relation between a production application and its maximum and deletion production application. Upper Half: The application of PA to G. Lower Half: The application of PA_{max} and PA_{del} to G.

The morphism $a = m$ fulfils the first equation. The morphism $b = i$ fulfils the second equation because the left pushout in the application of PA_{del} has to commute. Since, both morphisms can be found the maximum and deletion production applications are parallel independent.

In a sequence that first applies PA_{del} and then PA_{max}, the match morphism of PA_{max} is $id_I \circ b$. The application of PA_{max} with this match leads to a span $I \overset{id_I}{\leftarrow} I \overset{v}{\rightarrow} G'$. The right pushout in this application is a pushout along the morphisms $id_I \circ i = i$ and r. The right pushout in the application of PA is a pushout along the same morphisms. Since pushouts are unique up to isomorphisms the result of the application of PA_{max} after PA_{del} is isomorph to the result of the application of PA.

□

The following lemma states under which conditions maximum and deletion production applications are parallel independent:

Lemma 2 (Maximum and deletion production applications preserve parallel independence)**.** *Given production application* $PA_1 = (L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1, m_1)$ *with maximum production application* $PA_{1,max}$ *and deletion production application* $PA_{1,del}$ *and production application* $PA_2 = (L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2, m_2)$ *with maximum production application* $PA_{2,max}$ *and deletion production application* $PA_{2,del}$ *the following statements hold:*

1. *$PA_{1,max}$ and $PA_{2,max}$ are parallel independent.*
2. *$PA_{1,del}$ and $PA_{2,del}$ are parallel independent if PA_1 and PA_2 are parallel independent*
3. *$PA_{1,max}$ and $PA_{1,del}$ are parallel independent*
4. *$PA_{1,max}$ and $PA_{2,del}$ are parallel independent if PA_1 and PA_2 are parallel independent*

The proof for this lemma is as follows:

Proof. Maximum and deletion production applications preserve parallel independence. The four cases can be proven separately as follows:

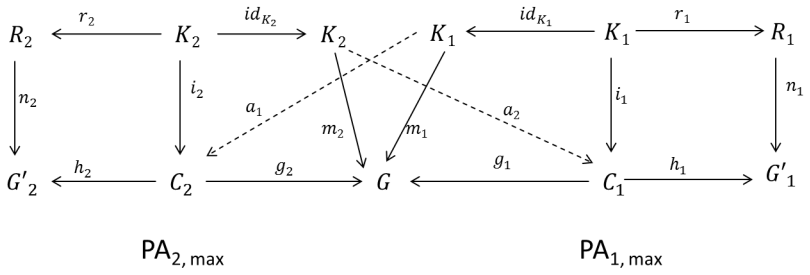
- *Case 1: Parallel independence of maximum production applications.*

  Given two maximum production applications

  – $PA_{1,max} = ((K_1 \overset{id_{K_1}}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1), m_1)$

  – $PA_{2,max} = ((K_2 \overset{id_{K_2}}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2), m_2)$

  applied, to a graph G with the following naming:



  In order to show parallel independence, two morphisms $a_1 : K_1 \rightarrow C_2$ and $a_2 : K_2 \rightarrow C_1$ with $m_1 = g_2 \circ a_1$ and $m_2 = g_1 \circ a_2$ are required [38].

  The morphisms g_1 and g_2 are identities since they are the pushout complements of identities id_{K_1} and id_{K_2}. Thus, $C_1 = C_2 = G$ and $g_1 = g_2 = id_G$. Accordingly, the following equations hold:

  $m_1 = g_2 \circ a_1 = id_G \circ a_1 = a_1$

  $m_2 = g_1 \circ a_2 = id_G \circ a_2 = a_2$

  Thus, it is always possible to find the morphisms $a_1 = m_1$ and $a_2 = m_2$. Accordingly, two maximum production applications to the same object G are always parallel independent.

- *Case 2: Parallel independence of deletion production applications.*

  The two deletion production applications $PA_{1,del}$ and $PA_{2,del}$ have the same left hand side morphism and match as PA_1 and PA_2. Accordingly, the left pushout of

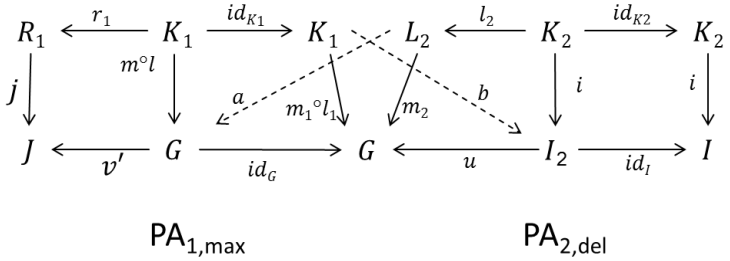
their application is the same and the required morphisms for parallel independence have to fulfil the same condition. These morphisms have to exist because PA_1 and PA_2 are parallel independent. Thus, the deletion production applications are also parallel independent.

- *Case 3: Parallel independence of maximum and deletion production application of the same production application.*

  This is true according to Lemma 1.

- *Case 4: Parallel independence of maximum and deletion production application of different production applications.*

  For the independence of $PA_{1,max}$ and $PA_{2,del}$ two morphisms $a : L_2 \to G$ and $b : K_1 \to I_2$ are required with $id_G \circ a = m_2$ and $u \circ b = m_1 \circ l_1$. An illustration of the application of both production applications can be seen in the following figure.



  The morphism $a = m_2$ fulfils the first equation. Because PA_1 and PA_2 are parallel independent there is a morphism $x : L_1 \to I_2$ with $u \circ x = m_1$. Accordingly, the morphism $b = x \circ l_1$ fulfils the second equation. Thus, $PA_{1,max}$ and $PA_{2,del}$ are parallel independent.

  The parallel independence of $PA_{2,max}$ and $PA_{1,del}$ can be shown analogous.

All four statements can be proven. Thus, the lemma holds. □

The following lemma states that the application of maximum and deletion production applications can each be done in one pushout

Lemma 3 (Representing Pushouts for Maximum and Deletion Production Applications)**.** *Given a production application PA to an object G with maximum production application $PA_{max} = (K \overset{id_K}{\leftarrow} K \overset{r}{\rightarrow} R, m \circ l)$ and deletion production application $PA_{del} = (L \overset{l}{\leftarrow} K \overset{id_K}{\rightarrow} K, m)$ the following statements hold:*

1. *$apply(G, \{PA_{max}\})$ leads to the same result as constructing the pushout $G \overset{m \circ l}{\leftarrow} K \overset{r}{\rightarrow} R$*
2. *$apply(G, \{PA_{del}\})$ leads to the same result as constructing the pushout complement of $G \overset{m}{\leftarrow} L \overset{l}{\leftarrow} K$*

The proof is as follows:

Proof. Representing Pushouts for Maximum and Deletion Production Applications. The proof for both statements relies on the fact that one of the morphism in the respective production application is an identity. The proofs are as follows:

1. *Maximum Production Application*

   For the maximum production application the left hand side morphism of the production is an identity. Accordingly, the result of the first step of its application (building the pushout complement along the left morphism and the match) is isomorph to G. The right hand side morphism of this production is exactly the pushout of $G \overset{m \circ l}{\leftarrow} K \overset{r}{\rightarrow} R$. Accordingly, the application of the maximum production application leads to a result that is isomorph to the result of of the construction of this pushout.

2. *Deletion Production Application*

   For the deletion production application the first step of the application results in the pushout complement of $G \overset{m}{\leftarrow} L \overset{l}{\leftarrow} K$. Accordingly, this pushout complement exists. The right hand side morphism of the production and thus the second step of the application is an identity and thus the result of the second step is isomorph to that of the first step.

Since both statements can be shown the lemma holds.

□

B.6.2. Main Proof

The well-definedness of the maximum graph is described in Theorem 6. The proof for this theorem is as follows:

Proof. The maximum graph is constructed by applying a set of maximum production applications to the object G. This proof shows that the construction of maximum production applications from $Prods$ leads to valid production applications and that the application of all maximum production applications to G is always possible.

The proof is done in two steps. First, we show that the construction of a maximum production application leads to a valid production application. The second subproof shows that it is always possible to apply all maximum production applications.

- **Well-definedness of the maximum production application**

  Given a production application $pa_i = ((L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i), m_i)$, the maximum production application $pa_{i,max} = ((K_i \overset{id_{K_i}}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i), m_i \circ l_i)$ consists of a graph transformation production $(K_i \overset{id_{K_i}}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i)$ and match $m_i \circ l_i$.

The production correctly defines a span and both morphisms are in $\mathcal{M}$. The morphism r_i is in $\mathcal{M}$ as it is contained in pa_i and pa_i is a valid production application. The identity id_{K_i} is an isomorphism and thus also in class $\mathcal{M}$.

The composition $m_i \circ l_i$ leads to a valid morphism from K_i (the left hand side of the production in $pa_{i,max}$) to the object G. The production is applicable with this match morphism because the pushout complement along the identity id_{K_i} can always be constructed.

Since the production and match of $pa_{i,max}$ are well-defined, the maximum production application is also well-defined.

- **Applicability of all maximum production applications**

  From Lemma 2 we know that two maximum production applications are always parallel independent, even if the original production applications are not. According to parallel independence and the Local Church Rosser Theorem (See Section 2.2.4) it is always possible to apply a set of parallel independent production applications and the result is the same for any order of application.

  Accordingly, given a set of production applications to an object G, the set of maximum production applications for these production applications is parallel independent and we can apply all of them to G. The result is the maximum graph.

Since maximum production applications are well-defined and parallel independent the maximum graph can always be constructed given a set of production applications to the same object.

□

B.7. Theorem 7 - Subset Properties of the Maximum Graph

In this section we give proof that a maximum graph contains all objects that result from the application of any subset of the original production applications as sub-object. This is stated in Theorem 7 by the existence of three inclusion morphisms for each subset.

The proof for the existence of these morphisms is as follows:

Proof. Inclusion Morphisms. Given a parallel independent set of n production applications $Prods$, the proof assumes an arbitrary but fixed subset of production applications $prod = \{PA_1, ..., PA_i\} \subset Prods$ where $Prods \backslash prod = \{PA_{i+1}, ..., PA_n\}$. This proof first shows that the application of production applications in the theorem is correct and leads to the described objects and morphisms. Subsequently, we show that the morphisms are monomorphisms.

Since all production applications in $Prods$ are parallel independent we know that their maximum and deletion production applications are also parallel independent (Lemma 2). Accordingly, we can apply the Parallelism Theorem (See Section 2.2.4) and build parallel rules from them. Thus, the parallel rule $PA_{prod,max}$ of all maximum production

applications for production applications in $prod$, the parallel rule $PA_{prod,del}$ of all deletion production applications for production applications in $prod$ and the parallel rule $PA_{Prods\backslash prod,max}$ for all maximum production applications in $Prods\backslash prod$ do exist.

Since all production applications in $prod$ and their maximum and deletion production applications are parallel independent, any order of their application leads to the same result according to the Local Church Rosser Theorem (See Section 2.2.4). Also, according to Lemma 1 the pairwise application of maximum and deletion production applications leads to the same result as the application of the original production application. Accordingly, the application of $PA_{prod,max}$ and $PA_{prod,del}$ leads to the same result as $apply(G, prod)$. Similarly, the application of $PA_{prod,max}$ and $PA_{Prods\backslash prod,max}$ leads to the same result as $max(G, Prods)$. The application of these parallel production applications is depicted in Figure B.3.

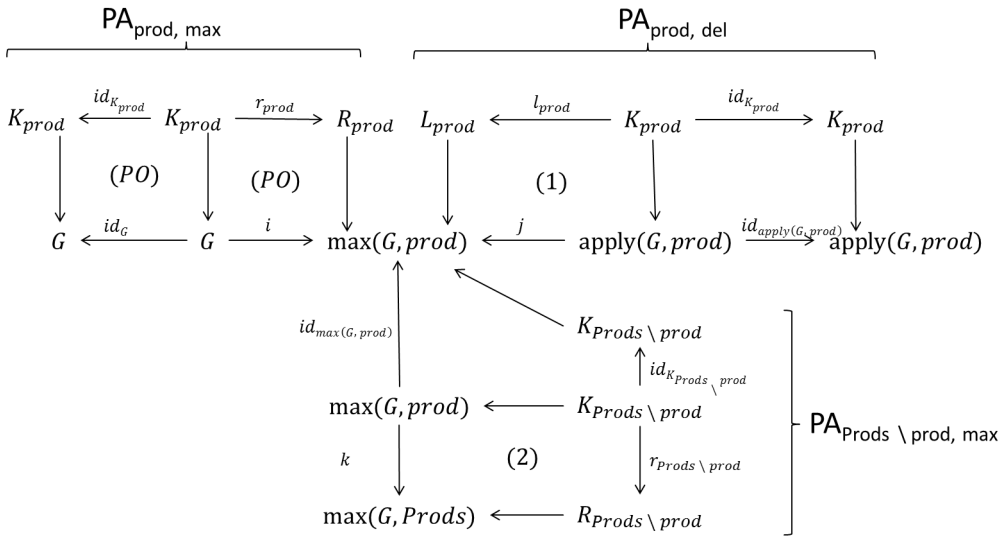


Figure B.3.: Illustration of the two orders of production applications and their intersection for the proof of Theorem 7.

As described in Lemma 3 the application of maximum and deletion production applications can be described as a single pushout. Accordingly, the application of $PA_{prod,del}$ to $max(G, prod)$ can be derived from the single pushout (1) in the figure. Therefore, the morphism $j = map'_{prod}$ exists.

Similarly, the application of $PA_{Prods\backslash prod,max}$ to $max(G, prod)$ can be derived from the single pushout (2) in the figure. Thus, the morphism $k = incl_{prod}$ exists.

Since both the morphisms map'_{prod} and $incl_{prod}$ exist their composition map'_{prod} also has to exist.

The category of typed graph diagrams is $\mathcal{M}$-adhesive. Accordingly, the morphisms j and k are in $\mathcal{M}$ because $\mathcal{M}$ is preserved by pushouts (1) and (2). l_{prod} and $r_{Prods\backslash prod}$ are in $\mathcal{M}$ because they are morphisms in a graph transformation production. Since $\mathcal{M}$ is a class of monomorphisms we know that j and k and thus map'_{prod} and $incl_{prod}$ are

monomorphisms and their composition map_{prod} is also a monomorphism.

Thus, all three morphisms exist and are monomorphisms. Therefore, the inclusion morphisms into the maximum graph always exist.

□

B.8. Theorem 8 - Correct Definition of the Items in Definition 19

In this section we proof that the elements defined in Definition 19 are defined correctly. We prove the following three statements:

- The function *required* describes all production applications that are required for the existence of a morphism.
- The function *averse* describes all production applications whose application is averse to the existence of a morphism.
- The function $possibleMorphisms$ describes all morphisms that are possible in any subset of production applications.

This is stated in Theorem 8.
The proof is as follows:

Proof. Each statement can be proven separately as follows:

1. *required*

   The proof for this statement assumes an arbitrary but fixed production application $p \in Prods$. Both directions of the equivalence are proven separately. First, we prove that if $p \in required(q, Prods)$ the right hand side of the equivalence holds ($\Rightarrow$). Subsequently, we prove that if the right hand side of the equivalence holds then p is in $required(q, Prods)$ ($\Leftarrow$).

   - $\Rightarrow$

     This statement is proven by indirect proof. We assume that the left hand side of the implication does not hold but the right hand side does. From this we show a contradiction.

     We make the following two assumptions:

     i) The production p is an element of $required(q, Prods)$.

     ii) There is a set of production applications $prod \subset Prods$ that does not contain p for which a morphism q_{prod} with $q = map_{prod} \circ q_{prod}$ exists.

     To show the contradiction we apply the productions in a specific order as shown in Figure B.4. First, we apply the parallel maximum production application for $prod$ leading to $max(G, prod)$ via one pushout (1). According to Lemma 3 the application via one pushout is possible. Then, we apply p to this

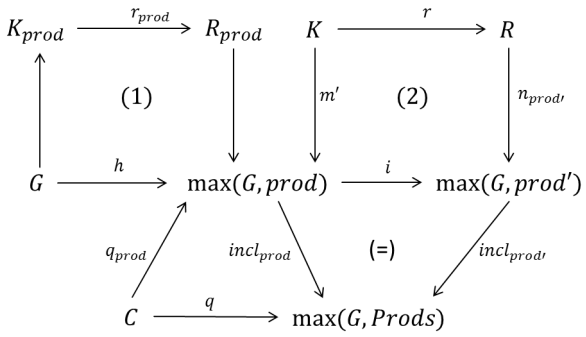


Figure B.4.: Illustration of the morphisms used in the proof for Theorem 8 (1)

maximum graph leading to $max(G, prod')$ where $prod' = prod \cup p$. According to Theorem 7 there are inclusion morphisms $incl_{prod}$ and $incl_{prod'}$ into the maximum graph. Since these inclusion morphisms are defined over the morphisms that result from the application of maximum production applications we know that

iii) $incl_{prod} = incl_{prod'} \circ i$.

Because p is assumed to be in $required(q, Prods)$ (Assumption i)) the following statement holds: $overlappAddElements(q, p) \neq \emptyset$. Accordingly, there is at least one element e in $max(G, Prods)$ that is targeted by q (there is an $e_c \in C$ with $q(e_c) = e$) and is contained in $incl_{\{p\}}(n(R)) \backslash incl_{\{p\}}(n(r(K)))$. For the application of p after prod we know that e is also contained in $incl_{prod'}(n_{prod'}(R)) \backslash incl_{prod'}(n_{prod'}(r(K)))$ because the production application adds the same elements when it is applied after $prod$. Thus, we know:

iv) $\exists e_{prod'} \in max(G, prod') : incl_{prod'}(e_{prod'}) = e$

v) $\nexists e_R \in R : incl_{prod'}(n_{prod'}(e_R))' = e$

From Assumption ii) we know that there is an element $e_{prod} = q(e_c)$ for which the following holds:

vi) $\exists e_{prod} \in max(G, prod) : incl_{prod}(e_{prod}) = e$

The diagram in Figure B.4 is on the level of graph diagrams. However, the same diagram exists for each contained attributed typed graph and for each component in this graph, because the objects and morphisms in graph diagrams are component-wise constructed over ATGs and the ones in ATGs are component-wise constructed over sets. In the following we argue over the component of the attributed graph that contains e, e_{prod}, $e_{prod'}$ and e_R. The objects and morphisms are all in sets, but form the same diagram as in the figure. In this component we know that $e_{prod'} = i(e_{prod})$. This can be derived from iii) and from the fact that $incl_{prod'}$ is a monomorphism and thus is surjective in sets. Since $i(e_{prod})$ and $e_{prod'}$ are both mapped to the same element e by this morphism they are the same element.

Since (2) is also a pushout in sets and because $e_{prod'}$ is mapped both from $max(G, prod)$ and from R we know that there is also an element in K that is mapped to $e_{prod'}$ via $i \circ m'$ and $n_{prod'} \circ r$ from the construction of a pushout in sets. However, this is in contradiction with v), which states that such an element does not exist.

Since the assumption that p is required but the right hand side does not hold leads to a contradiction the right hand side has to hold.

- $\Leftarrow$

  This direction is proven indirectly by assuming that p is not contained in $required(q, Prods)$ and showing that in this case the right hand side cannot be fulfilled. The right hand side consists of an all-quantified expression over all subsets of $Prods$. The proof assumes one arbitrary but fixed subset of production applications $prod$ that contains p and for which a morphism $q_{prod} : C \to apply(G, prod)$ exists and fulfils:

  i) $q = map_{prod} \circ q_{prod}$

  From this we show that for $prods' = prod \backslash \{p\}$ a morphism $q_{prod'}$ with the required property exists although $prods'$ does not contain p. In this case the all-quantifier does not hold. Figure B.5 illustrates the relevant morphisms.

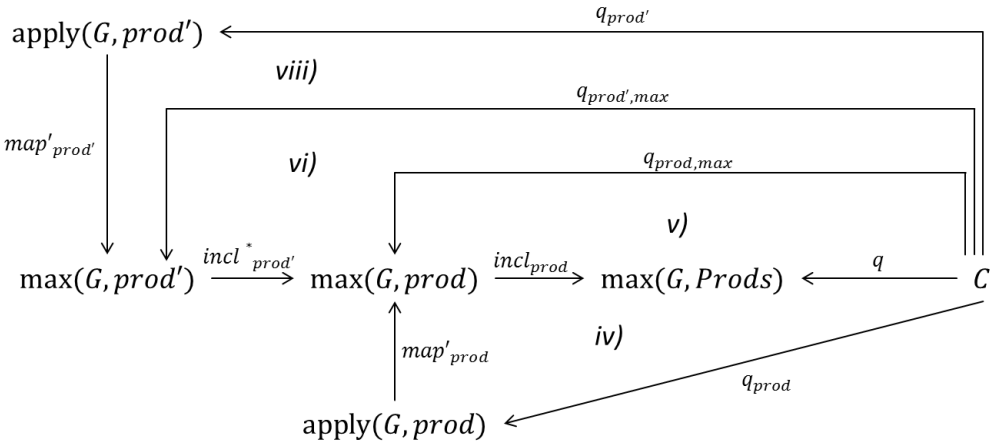


Figure B.5.: Illustration of the morphisms used in the proof for Theorem 8 (2).

If $p \notin required(q, prod)$ then according to Definition 19 the following holds:

ii) $overlapAddElements(q, p) = \emptyset$

According to Theorem 7 map_{prod} can be deconstructed as follows

iii) $map_{prod} = incl_{prod} \circ map'_{prod}$

From i) and iii) it follows that:

iv) $q = incl_{prod} \circ map'_{prod} \circ q_{prod}$

For this reason a morphism $q_{prod,max} : C \to max(G, prod) = map'_{prod} \circ q_{prod}$ exists and fulfills:

v) $q = incl_{prod} \circ q_{prod,max}$

Because of ii) we know that p does not add any element that is targeted by q. Accordingly, all elements in $q(C)$ exist before applying p. Thus, all elements $q_{prod,max}(C)$ in $max(G, prod)$ have a pre-image in $max(G, prod')$ under the inclusion morphism $incl^*_{prod'} : max(G, prod') \to max(G, prod)$. This morphism exists according to Theorem 7, if $prod$ is interpreted as the set of all production applications and $prod'$ as the subset. Accordingly, there is a morphism $q_{prod',max}C \to max(G, prod')$ with

vi) $q_{prod,max} = incl^*_{prod'} \circ q_{prod',max}$

Since a morphism q_{prod} that fulfills i) exists we know from Theorem 8 point 2 that all production applications in $prod$ are not averse to q. Accordingly, all $p_i \in prod$ are not averse and according to Definition 19 it follows that:

vii) $overlapRemoveElements(q, p_i) = \emptyset$

From this we derive that no production application in $prod'$ removes any element targeted by q. Accordingly, all targeted elements of $q_{prod',max}$ are present in $apply(G, prod')$ and for the morphism $map'_{prod'} : apply(G, prod') \to max(G, prod')$ there is a morphism $q_{prod'} : C \to apply(G, prod')$ with

viii) $q_{prod',max} = map'_{prod'} \circ q_{prod'}$

From viii) and vi) it follows that

ix) $q_{prod,max} = incl^*_{prod'} \circ map'_{prod'} \circ q_{prod'}$

From ix) and v) it follows that

x) $q = incl_{prod} \circ incl^*_{prod'} \circ map'_{prod'} \circ q_{prod'}$

The morphism $incl^*_{prod'}$ is the inclusion from $max(G, prod')$ into $max(G, prod)$ according to Theorem 7. The morphism $incl_{prod}$ is the inclusion morphism from $max(G, prod)$ into $max(G, Prods)$ according to Theorem 7. Both morphisms are derived from the application of parallel independent maximum production applications via one pushout according to Lemma 3. They can be combined to retrieve $incl_{prod'} : max(G, prod') \to max(G, Prods)$ and the following holds:

xi) $q = incl_{prod'} \circ map'_{prod'} \circ q_{prod'}$

From Theorem 7 it follows that:

xii) $q = map_{prod'} \circ q_{prod'}$

Since $prod'$ is an example for a set that does not contain p but still fulfils property xii) the all-quantifier on the right hand side does not hold. Thus, whenever the statement on the right hand side holds for a production application this production application is in $required(q, Prods)$.

2. This proof is divided into two parts. First, we show that for each production application in $averse(q, Prod)$ the right hand side of the equivalence is fulfilled($\Rightarrow$). Second, we show that each production application for which the right hand side is fulfilled is contained in $averse(q, Prod)$ ($\Leftarrow$).

   - $\Rightarrow$: This direction is shown by indirect proof. We assume an arbitrary but fixed set of production applications $prod \subset Prods$ that contains an averse production application p and show that for this set no morphism q_{prod} exists that fulfils

     i) $q = map_{prod} \circ q_{prod}$

     where map_{prod} is the morphism according to Theorem 7 which can be decomposed into two morphisms as follows:

     ii) $map_{prod} = incl_{prod} \circ map'_{prod}$

     The statement is proven by assuming the existence of this morphism and showing a contradiction. The objects, morphisms and elements used in this definition are illustrated in Figure B.6.

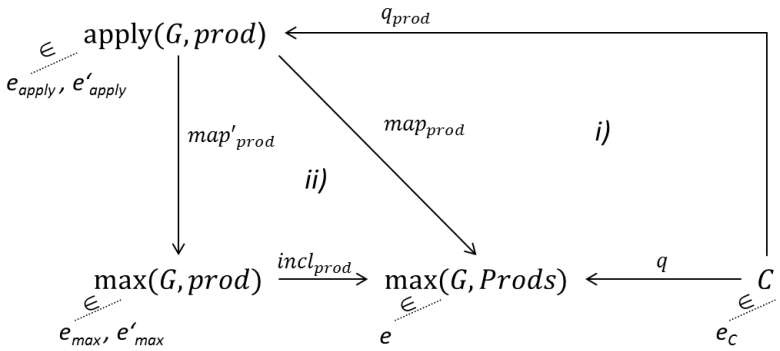


Figure B.6.: Illustration of the morphisms used in the proof for Theorem 8 (3).

According to Definition 19 each production application p in $averse(q, Prod)$ fulfils the equation $overlapRemoveElements(q, p) \neq \emptyset$. This set contains all elements in $max(G, prod)$ that are deleted by p and targeted by q. Let e be one element from $overlapRemoveElements(q, p)$ and e_C be the element in C with $q(e_C) = e$. The fact that e is from $overlapRemoveElements(q, p)$ means that it is mapped from the left hand side of p but deleted in its first application step. Since the maximum graph does not take into account deletions there is an element $e_{max} \in max(G, prod)$ with

iii) $incl_{prod}(e_{max}) = e$

However, since p deletes this element there is no element $e_{apply} \in apply(G, prod)$ with $map'_{prod}(e_{apply}) = e_{max}$.

The assumption, that a morphism q_{prod} with the required property exists now means that there is an element $e'_{apply} = q_{prod}(e_Q)$ with

iv) $map_{prod}(e'_{apply}) = e$

From the decomposition of map_{prod} and iv) it follows that there is an element $e'_{max} \in max(G, prod) = map'_{prod}(e'_{apply})$ with

v) $incl_{prod}(e'_{max}) = e$

Since e_{max} is not mapped by map'_{prod} we know that $e_{max} \neq e'_{max}$. However, this means that both e_{max} and e'_{max} are mapped to e by $incl_{prod}$ according to iii) and v). The morphism $incl_{prod}$ is assumed to be a monomorphism in graph diagrams, which means, the morphism for the attributed graph that contains e is a monomorphism in attributed graphs. Accordingly, the component morphism for the component in this attributed graph that contains e (node, edge, node attribute edge, edge attribute edge) is a monomorphism in the category of sets. Monomorphisms in sets are injective morphisms. However, since the morphism needs to map the different elements e_{prod} and e'_{prod} to the same element e it cannot be injective and thus $incl_{prod}$ cannot be a monomorphism. This is a contradiction to the assumptions.

The assumption of the existence of q_{prod} with the above property leads to a contradiction for an arbitrary set $prod$ that contains p. Thus, such a set cannot exist and the right hand side of the equivalence has to hold for all averse production applications p.

- $\Leftarrow$: This proof shows that given the right hand side of the equivalence holds it follows that $p \in averse(q, Prods)$. We show this by proofing that the set $overlapRemoveElements(q, p)$ is not empty and thus, p is averse according to Definition 19.

  According to Point 3 in Theorem 8 the fact that q is contained in the set $possibleMorphisms(p, a)$ means that there is at least one subset $prod \subset prods$ for which a morphism q_{prod} exists and fulfils

  i) $q = map_{prod} \circ q_{prod}$

  where map_{prod} is the morphism from Theorem 7.

  Since p is assumed to fulfil the right hand side of the equivalence (meaning it is not part of any subset for which such a morphism exists) this means $p \notin prod$. From the right hand side it also follows that for $prod' = prod \cup \{p\}$ no morphism $q_{prod'}$ exists with $q = map_{prod'} \circ q_{prod'}$ where $map_{prod'}$ is the morphism from Theorem 7.

  Since all production applications in $Prods$ are parallel independent their maximum and deletion production applications are also parallel independent according to Lemma 2. Accordingly, they can be applied in any order. Figure B.7 shows the relation of two application sequences that lead to $apply(G, prod)$ and $apply(G, prod')$. All maximum and deletion production

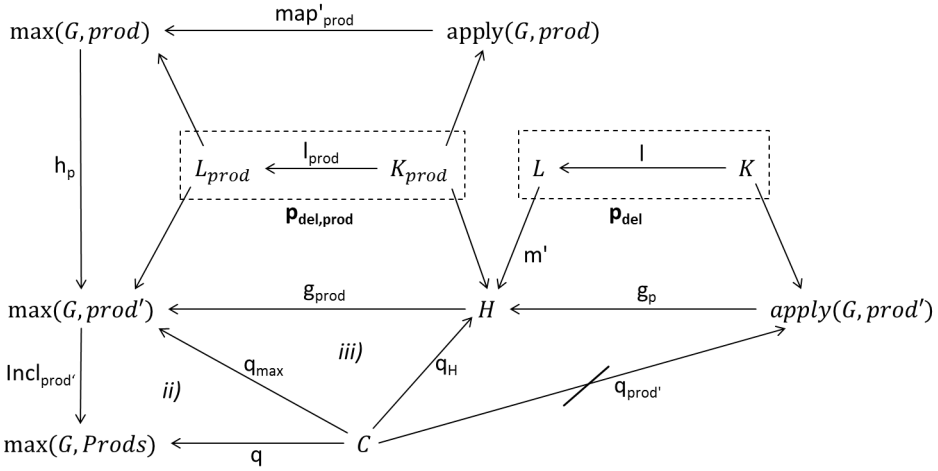


Figure B.7.: Illustration of the morphisms used in the proof for Theorem 8 (4).

applications have been applied as single pushouts as described in Lemma 3. Furthermore, the production applications in *prod* have been abbreviated by building the parallel production application for their maximum and deletion production applications.

In the figure the graph $max(G, prod)$ has been created by applying the parallel production application for all maximum production applications in *prod* to G. To this graph the maximum production application p_{max} for p is applied to arrive at $max(G, prod')$ via morphism h_p. The parallel production application $prod_{del}$ for all deletion production applications in *prod* is applied to $max(G, prod)$ to arrive at $apply(G, prod)$. To arrive at $apply(G, prod')$ the production $prod_{del}$ is applied first to arrive at H. Both applications of this parallel deletion production applications are pushouts with the common morphism l_{prod} which is the left hand side of the parallel rule for the deletion production applications in *prod*. From H it we can arrive at $apply(G, prods')$ by applying p_{del}, the deletion production application for p. The result of the application of this production application is the morphism g_p.

From i) and from the fact that $incl_{prod}$ is the same as $h_p \circ incl_{prod'}$ because of its construction in Theorem 7 it can be derived that there is a morphism $q_{max} : C \to max(G, prods')$ which is defined as $h_p \circ map'_{prod} \circ q_{prod}$ with

ii) $q = incl_{prod'} \circ q_{max}$

The objects L_{prod} and K_{prod} are the coproduct of the left hand sides and interfaces of the deletion production applications in *prod*. None of the production applications in *prod* are averse to q. Otherwise the morphism q_{prod} that fulfils i) would not exist according to direction $\Rightarrow$ of this proof. Accordingly, $overlapRemoveElements(q, p_i)$ is empty for each production applica-

tion $p_i \in prod$. This means none of the production applications p_i delete any element that is targeted by q_{max}. Accordingly, we can find a morphism $q_H : C \to H$ with

iii) $q = incl_{prod'} \circ g_{prod} \circ q_H$

From the assumption that $q_{prod'}$ does not exist it follows that there is at least one element c in C for which there is no element $c_{prods'}$ in $apply(G, prod')$ with $q_H(c) = g_P(c_{prods'})$. Accordingly, the element $q_H(c)$ has been deleted by p_{del}, which means there is an element c_L in L for which $m'(c_L) = q_H(c)$ but no element c_K in K with $l(c_k) = c_L$.

Because of parallel independence of maximum and deletion production applications the mapping of L into the maximum graph via morphism $incl_{prod} \circ g_{prod}$ and match m' is the same as its mapping via $incl_p \circ h$ and match m as used in Definition 19. Accordingly we know that

iv) $incl_{prod}(g_{prod}(m'(c_L))) \in incl_p(h(m(L)))\backslash incl_p(h(m(l(K))))$

Thus, $overlapRemoveElements(q, p)$ is not empty and thus p is averse for q.

Both directions hold and for this reason $averse(q, Prod)$ contains exactly the set of productions that forbid the existence of q.

3. In this proof we show that $possibleMorphisms(Prod, p, a)$ contains all morphisms q which exist in at least one subset of production applications from $Prod$. We show both directions of the equivalence separately. First, we show that whenever the morphism q is contained in $possibleMorphisms(Prod, p, a)$ the right hand side of the equivalence is fulfilled ($\Rightarrow$). Second, we show that the right hand side of the equivalence implies that q is contained in $possibleMorphisms(Prod, p, a)$ ($\Leftarrow$).

   - $\Rightarrow$

     In this direction we show that a set $prod$ that contains all required production applications but no reverse production applications always exists and fulfils the right hand side of the equivalence if $q \in possibleMorphisms(Prods, p, a)$. The morphisms mentioned in the Theorem are illustrated in Figure B.8.

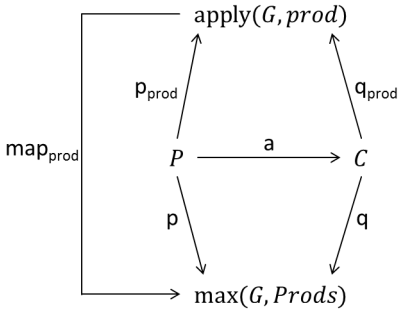


Figure B.8.: Illustration of the morphisms used in the proof for Theorem 8 (5).

From the fact that a morphism q is in $possibleMorphisms(Prods, p, a)$ and from the definition of $possibleMorphisms$ the following can be derived:

i) $p = q \circ a$

ii) $required(q, Prods) \cap averse(q, Prods) = \emptyset$

Because of ii) the set $prod$ always exists as required and averse production applications are disjoint.

The elements in $q(C)$ can be divided into three sets. Added elements are the elements that need to be added by a production application in $Prods$ to exist. They are not contained in the original diagram G. Removed elements are elements that are removed by at least one production application in $Prods$. They are in G, but not in $apply(G, Prods)$. Preserved elements are elements that are neither added nor removed by any production application in $Prods$.

Preserved elements are already present in G and are not deleted by any production application. Accordingly, for every preserved element $q(pres)$ in $max(G, Prods)$ there is an element $pres_{prod}$ in $apply(G, prod)$ with $q(pres) = map_{prod}(pres_{prod})$.

Each added element $q(a)$ is added by exactly one production application p. In this production application the respective element that is mapped to a is in the right hand side but not in the interface. Thus, $overlapAddElements(q, p_i)$ contains $q(a)$ and is not empty. After applying all production applications in $prod$ this element has been added. Thus, there is an element $a_{prod} \in apply(G, prod)$ with $q(a) = map_{prod}(a_{prod})$.

Similarly, each deleted element $q(d)$ is deleted by a production application for which $overlapRemoveElements(q, p_i)$ is not empty. Accordingly, that production application is averse. The set $prod$ does not contain any averse production applications. Accordingly, the application of the production applications in $prod$ does not remove this elements from G. Thus, we can find an element $d_{prod} \in apply(G, prod)$ with $q(d) = map_{prod}(d_{prod})$.

This shows that the preimage of $q(C)$ exists for all elements under the morphism map_{prod}. It is thus possible to define the morphism q_{prod} as $map_{prod}^{-1} \circ q$. This morphism fulfils:

iii) $map_{prod} \circ q_{prod} = map_{prod} \circ map_{prod}^{-1} \circ q = q$

p_{prod} can be constructed as $q_{prod} \circ a$. Thus, it fulfils the following properties:

iv) $p_{prod} = q_{prod} \circ a$

v) $p = q \circ a = map_{prod} \circ q_{prod} \circ a = map_{prod} \circ p_{prod}$

Since iii), iv) and v) are fulfilled the right hand side of the equivalence is fulfilled. The set $required(q, Prods)$ is an example of such a subset $prod$.

- $\Leftarrow$

From the equation on the right hand side the following is given for a set or production applications $prod$:

i) $prod \subset Prods$

ii) morphism q with $p = q \circ a$

iii) morphism $p_{prod} : P \to apply(G, prod)$ with $p = map_{prod} \circ p_{prod}$

iv) morphism $q_{prod} : C \to apply(G, prod)$ with $q = map_{prod} \circ q_{prod}$

v) $p_{prod} = q_{prod} \circ a$

According to $\Rightarrow$ of part 1 of this proof it follows that $required(q, Prods) \subset prod$ from iv). From $\Rightarrow$ of part 2 of this proof and iv) it follows that $averse(q, Prods) \cap prod = \emptyset$. Accordingly it follows that:

vi) $required(q, Prods) \cap averse(q, Prods) = \emptyset$

Because of ii) and vi) q is in $possibleMorphisms(Prods, p, a)$ as defined in Definition 19

Since both directions can be proven the statement holds.

□

B.9. Theorem 9 - Correctness and Completeness of Existence Conditions for a Morphism

In this section we proof that the existence conditions for morphisms defined in Definition 21 are correct and complete. This is stated in Theorem 9

The proof is as follows:

Proof. The statement $existenceCondition(Prods, q)$ is defined as

- $positiveCondition(Prods, q) \wedge \neg negativeCondition(Prods, q)$

A morphism q' exists in a subset of production applications $prod \subset Prods$ if and only if this subset contains all required and no averse production applications. The first direction ($\Rightarrow$) holds according to Statements 1 and 2 in Theorem 8). The second direction ($\Leftarrow$) has been shown in direction $\Rightarrow$ of subproof 3 for Theorem 8) in Appendix B.8.

In this proof we show that the statement $existenceCondition(Prods, q)$ is equivalent to the statement that there is a set of production applications $prod \subset Prods$ that contains all required and no averse production applications. This is shown by proving the following two statements:

1. $eval(positiveCondition(Prods, q), prod) = True \Leftrightarrow required(p, Prods) \subset prod$
2. $eval(negativeCondition(Prods, q), prod) = False \Leftrightarrow averse(p, Prods) \cap prod = \emptyset$

Both statements are shown separately:

- $eval(positiveCondition(Prod, q), prod) = True \Leftrightarrow required(p, Prod) \subset prod$
  Given an arbitrary but fixed $prod \subset Prods$ the following holds:

$$
\begin{aligned}
& eval(positiveCondition(Prods, q), prod) = True \\
\Leftrightarrow & eval(\wedge_{p_i \in required(p, Prods)}(\wedge_{elem \in overlapAddElements(q,p_i)}(Adds(p_i, elem))), prod) \\
& = True && \text{(Definition of positiveCondition)} \\
\Leftrightarrow & \forall p_i \in required(p, Prods).eval(\wedge_{elem \in overlapAddElements(q,p_i)}(Adds(p_i, elem)), \\
& prod) = True && \text{(Evaluation of conjunction)} \\
\Leftrightarrow & \forall p_i \in required(p, Prods).p_i \in prod && \text{(Evaluation of Adds)} \\
& && (overlappAddElements(q, p_i) \neq \emptyset) \text{ if } p_i \in required(q, Prods)) \\
\Leftrightarrow & required(p, Prods) \subset prod && \text{(Definition of subset)}
\end{aligned}
$$

- $eval(negativeCondition(Prods, q), prod) = False \Leftrightarrow averse(p, Prods) \cap prod = \emptyset$
  The following equation holds:

$$
\begin{aligned}
& eval(negativeCondition(Prods, q), prod) = False \\
\Leftrightarrow & eval(\vee p_i \in averse(p, Prods)(\vee_{elem \in overlapRemoveElements(q,p_i)} \\
& (Removes(p_i, elem))), prod) = False && \text{(Definition of negativeCondition)} \\
\Leftrightarrow & \forall p_i \in averse(p, Prods) eval(\vee_{elem \in overlapRemoveElements(q,p_i)}(Removes(p_i, elem)) \\
& , prod) = False && \text{(Definition of conjunction)} \\
\Leftrightarrow & \forall p_i \in averse(p, Prods).p_i \notin prod && \text{(Evaluation of Removes)} \\
& && \text{(existence of an overlappRemoveElement if } p_i \in averse(p, Prod)) \\
\Leftrightarrow & averse(p, Prods) \cap prod = \emptyset && \text{(Definition of conjunction)}
\end{aligned}
$$

Since the equivalence of both statements holds the morphism q' always exists iff the existence condition is fulfilled because a subset of $Prods$ that contains all required and no averse production applications exists iff it fulfils both conditions and the morphism q' exists iff such a subset exists. □

B.10. Theorem 10 - Correctness and Completeness of Reasons for the Fulfilment of a Nested Condition

In this section we proof that the fulfilment reasons for constraints defined in Definition 22 are correct and complete. This is described in Theorem 10.

The proof is as follows:

Proof. The proof assumes an arbitrary but fixed set $prod \subset Prods$.

The statement is proven via structural induction for conditions with finite nesting levels. The proof is done for the conditions *true*, *exists*, *neg* and $\wedge$. All other nested conditions are shortcuts for combinations of these conditions. In the proof we assume that the theorem holds for all inner conditions and show that it also holds for the outer condition. The element *true* is the inductive beginning.

The proofs for each condition are as follows:

- $c = true$:

  The following equation holds:

  $$\begin{aligned}
  & eval(existenceCondition(Prods, p) \wedge conditions(true, p, Prods), prod) = true \\
  \Leftrightarrow & eval(existenceCondition(Prods, p) \wedge true, prod) = true && \text{(Definition 22)} \\
  \Leftrightarrow & eval(existenceCondition(Prods, p), prod) = true \\
  & && (true \text{ is neutral in conjunctions}) \\
  \Leftrightarrow & \exists p' : P \to apply(G, prod) : p = map_{prod} \circ p' && \text{(Theorem 9)} \\
  \Leftrightarrow & \exists p' : P \to apply(G, prod) : p = map_{prod} \circ p' \wedge p' \models true \\
  & && \text{(true is always fulfilled)}
  \end{aligned}$$

  Thus, for $c = true$ the statement holds.

- $c = \neg c'$

  The following equation holds:

  $$\begin{aligned}
  & eval(existenceCondition(Prods, p) \wedge conditions(\neg c', p, Prods), prod) = true \\
  \Leftrightarrow & eval(existenceCondition(Prods, p) \wedge \neg conditions(c', p, Prods), prod) = true \\
  & && \text{(Definition 22)} \\
  \Leftrightarrow & eval(existenceCondition(Prods, p), prod) = true \wedge \\
  & eval(existenceCondition(Prods, p) \wedge conditions(c', p, Prods), prod) = false \\
  & && \text{(evaluation of conjunction and negation)} \\
  \Leftrightarrow & (\exists p' : P \to apply(G, prod) : p = map_{prod}) \wedge \\
  & eval(existenceCondition(Prods, p) \wedge conditions(c', p, Prods), prod) = false \\
  & && \text{(Theorem 9)} \\
  \Leftrightarrow & (\exists p' : P \to apply(G, prod) : p = map_{prod}) \wedge \\
  & \neg((\exists p' : P \to apply(G, prods) : p = map_{prod}) \wedge p' \models c') \\
  & && \text{(theorem holds for inner condition)} \\
  \Leftrightarrow & (\exists p' : P \to apply(G, prod) : p = map_{prod}) \wedge \neg(p' \models c') && \text{(conjunction)} \\
  \Leftrightarrow & (\exists p' : P \to apply(G, prod) : p = map_{prod}) \wedge p \models \neg c' \\
  & && \text{(definition of fulfillment of negated condition)}
  \end{aligned}$$

- $c = \wedge_{j \in J} c_j$

  The following equation holds:

$$
\begin{aligned}
& eval(existenceCondition(Prods, p) \wedge reasons(\wedge_{j \in J} c_j, p, Prods), prod) = true \\
\Leftrightarrow & eval(existenceCondition(Prods, p) \wedge \wedge_{j \in J} reasons(c_j, p, Prods), prod) = true \\
& \qquad \text{(Definition 22)} \\
\Leftrightarrow & (\wedge_{j \in J} eval(existenceCondition(Prods, p) \wedge reasons(c_j, p, Prods), prod)) \\
& = true \qquad \text{(Definition 20)} \\
\Leftrightarrow & \forall_{j \in J} (eval(existenceCondition(Prods, p) \wedge reasons(c_j, p, Prods), prod) \\
& = true) \qquad \text{(definition of conjunction)} \\
\Leftrightarrow & \forall_{j \in J} (\exists p' : P \to apply(G, prod) : p = map_{prod} \circ p' \wedge p' \models c_j) \\
& \qquad \text{(fulfilled for inner condition)} \\
\Leftrightarrow & \exists p' : P \to apply(G, prod) : p = map_{prod} \circ p' \wedge \forall_{j \in J} (p' \models c_j) \\
& \qquad (map_{prod} \text{ is a monomorphism}) \\
\Leftrightarrow & \exists p' : P \to apply(G, prod) : p = map_{prod} \circ p' \wedge p' \models (\wedge_{j \in J} c_j) \\
& \qquad \text{(definition of conjunction)}
\end{aligned}
$$

  Thus, for $c = \wedge_{j \in J} c_j$ the statement holds.

- $c = \exists(a, c')$

  The following equation holds:

$$
\begin{aligned}
& eval(existenceCondition(Prods, p) \wedge reasons(\exists(a, c'), p, Prods), prod) = true \\
\Leftrightarrow & eval(existenceCondition(Prods, p) \wedge \\
& \vee_{q \in possibleMorphisms(Prods,p,a)} (reasons(c', q, Prods)), prod) = true \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Definition 22)} \\
\Leftrightarrow & eval(existenceCondition(Prods, p), prod) = true \wedge \\
& \vee_{q \in possibleMorphisms(Prods,p,a)} eval(existenceCondition(Prods, q) \\
& \wedge reasons(c', q, Prods), prod) = true \qquad\qquad\qquad\qquad \text{(Definition 20)} \\
\Leftrightarrow & \exists p' : C \to apply(G, prod) : p = map_{prod} \circ p' \wedge \\
& \vee_{q \in possibleMorphisms(Prods,p,a)} eval(existenceCondition(Prods, q) \\
& \wedge reasons(c', q, Prods), prod) = True \qquad\qquad\qquad\qquad \text{(Definition 21)} \\
\Leftrightarrow & \exists p' : C \to apply(G, prod).p = map_{prod} \circ p' \wedge \\
& (\vee_{q \in possibleMorphisms(Prods,p,a)} \exists q' : \\
& C \to apply(G, prod) : q = map_{prod} \circ q' \wedge q' \models c') \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(fulfilled for inner conditions)} \\
\Leftrightarrow & \exists p' : C \to apply(G, prod).p = map_{prod} \circ p' \wedge \\
& \exists q : C \to max(G, Prods).p = q \circ a \\
& \exists q' : C \to apply(G, prod) : q = map_{prod} \circ q' \wedge q' \models c' \\
& \qquad\qquad\qquad\qquad\qquad\qquad (q \in possibleMorphisms(Prods, p, a)) \\
\Leftrightarrow & \exists p' : C \to apply(G, prod).p = map_{prod} \circ p' \wedge \\
& \exists q : C \to max(G, Prods).p = q \circ a \\
& \exists q' : C \to apply(G, prod) : q = map_{prod} \circ q' \wedge q' \models c' \wedge \\
& p' = q' \circ a \qquad\qquad\qquad\qquad (map_{prod} \text{ is a monomorphism}) \\
\Leftrightarrow & \exists p' : C \to apply(G, prod).p = map_{prod} \circ p' \wedge p' \models \exists(a, c') \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(definition of existence)}
\end{aligned}
$$

  Thus, for $c = \exists(a, c')$ the statement holds.

The statement holds for the inductive beginning *true* and for all ways to construct nested conditions out of other nested conditions and thus holds for all nested conditions with finite nesting levels. □