Software Systems Research Centre,
School of Design, Engineering & Computing,
Bournemouth University

Enabling collaborative modelling for a multi-site model-driven software development approach for electronic control units

Frank Grimm

December, 2012

A Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy of Bournemouth University

Copyright statement

Author's declaration

Abstract

Name of Author: Frank Grimm

Thesis Title: Enabling collaborative modelling for a multi-site model-driven software development approach for electronic control units

An important aspect of support for distributed work is to enable users at different sites to work collaboratively, across different sites, even different countries but where they may be working on the same artefacts. Where the case is the design of software systems, design models need to be accessible by more than one modeller at a time allowing them to work independently from each other in what can be called a collaborative modelling process supporting parallel evolution.

In addition, as such design is a largely creative process users are free to create layouts which appear to better depict their understanding of certain model elements presented in a diagram. That is, that the layout of the model brings meaning which exceed the simple structural or topological connections. However, tools for merging such models tend to do so from a purely structural perspective, thus losing an important aspect of the meaning which was intended to be conveyed by the modeller.

This thesis presents a novel approach to model merging which allows the preservation of such layout meaning when merging. It first presents evidence from an industrial study which demonstrates how modellers use layout to convey meanings. An important finding of the study is that diagram layout conveys domain-specific meaning and is important for modellers. This thesis therefore demonstrates the importance of diagram layout in model-based software engineering. It then introduces an approach to merging which allows for the preservation of domain-specific meaning in diagrams of models, and finally describes a prototype tool and core aspects of its implementation.

Acknowledgements

This PhD journey has been quite some learning experience, on many levels, and there are many people who have had an impact in one way or another.

I would particularly like to say “thank you” to the two gentlemen who had most influence on me as a researcher and my research, Prof. Georg Beier and Prof. Keith Phalp. I’m deeply grateful for your constant support, advice, and for helping me to become a more aware researcher – I thank you both.

I am also very grateful to Prof. Ray Dawson for the most insightful comments and suggestions on this research he provided during the viva voce.

I would like to take this opportunity to acknowledge the kind support that I received from my second supervisor Dr. Lai Xu, the School Research Administrator Naomi Bailey, and especially Dr. Sheridan Jeary.

I am also very grateful to my employer for taking into account that I had additional research workload.

I would also like to take this opportunity to thank Robert Bosch GmbH for funding parts of the research that lead to and supporting this thesis. Special thanks to Markus Schweizer, Martin Magira, and Uwe Maienberg from Bosch’s automatic gearbox controller software unit for their insights to and support with OMOS-related topics.

Very special thanks are given to Susi and all my family for their unconditional love, support, understanding, and patience.

Published materials

1. Phalp, K., Grimm, F., Xu, L., Supporting Collaborative Work by Preserving Model Meaning when Merging Graphical Models, 13th IFIP Working Conference on Virtual Enterprises, Bournemouth, UK, 2012

2. Grimm, F., Beier, G., Phalp, K., Vincent, J., Enabling multi-stakeholder co-operative modelling in automotive software development and implications for model driven software development. In: Proceedings of the 1st International Workshop on Business Support for MDA (MDABIZ 2008), co-located with TOOLS EUROPE 2008, ETH Zürich, Zürich, Switzerland, 2008

3. Grimm, F., Increasing the reliability of model-driven software family engineering and product configuration for automotive controller software, in Pohl, K., Heymans, P., Kang, K., Metzger A. (Ed.), Second International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'2008), Essen, Germany, 2008

4. Grimm, F., An approach for semi-automatic, mental map preserving visual merging of UML class models, Young researchers workshop, Westsächsische Hochschule Zwickau, Germany, 2007

5. Grimm, F., Beier, G., Phalp, K., Vincent, J., Towards semi-automatic, mental map preserving visual merging of UML class models, in Proceedings of IADIS International Conference of Applied Computing 2007, Salamanca, Spain, 2007

6. Beier, G., Grimm, F., Safety-inherent concepts for model-driven software family engineering in the automotive controller software domain, in Plödereder, E., Keller, H.B., Dencker, P., Tonndorf, M. (Ed.), Automotive — Safety & Security 2006 — Sicherheit und Zuverlässigkeit für automobile Informationstechnik, Stuttgart, Germany, 2006

7. Beier, G., Grimm, F., Approaches to testing UML-based state machines, in Stolzenburg, F. (Ed.), 7. Nachwuchswissenschaftlerkonferenz mitteldeutscher Fachhochschulen, Hochschule Harz, Wernigerode, Germany, 2005

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1. Introduction

This chapter provides introductory background information and a motivation to the research discussed in this thesis. Furthermore, the research aims and objectives are defined. This chapter closes with an outline of the content discussed in the remainder of this thesis.

1.1. Overview and motivation for the research presented in this thesis

Much attention is currently paid to *model-driven software development* (MDSD) [VS06]. In MDSD software models are key software development artefacts. They are first-class development artefacts used by many different stakeholders throughout the whole software development process. They are certainly no longer just marginal tools used only by design or analysis teams.

During the last decade, the *Unified Modeling Language* (UML) [BRJ05] has become a standard language for model-driven software development. It provides a rich set of complementary model types and graphical notations supporting the whole (object-oriented) software development process. UML is largely accepted throughout software industry and academia [DP06].

1.1.1. Pilot study on model-driven development of software for automotive electronic control units

As part of the research presented in this thesis, an industrial software development project realised in a model-driven manner was analysed with respect to the role software models play in this project. While details of this analysis will be presented in Chapter 3, a short introduction to the project is given below in order to allow for a better explanation of the research questions this thesis sets out to provide answers for.

In the analysed project, software for automatic gearbox controllers used in a variety of passenger car models from several car manufacturers is developed. This software is developed by Robert Bosch GmbH, one of the largest suppliers of automotive electronics. About 20 developers from Bosch's "Automatic Transmission Control Units" engineering group started working on this project in 2003. The group is located at two sites, Budapest, Hungary, and Schwieberdingen, Germany. The project is still being developed today (as Bosch is still building automatic gearbox controllers).

In the following, Bosch's approach to implementing electronic control unit software systems is outlined in order to provide an overview of the model-driven development process discussed in this thesis.

In the software design and implementation phase, i. e., after the analysis phase, the initial objective is to produce a software model which describes the *structure* of the software components implementing the automatic gearbox controller software's responsibilities. To do so, the Bosch developers apply an object-oriented modelling approach called *OMOS* (which stands for "object-oriented modelling concept for electronic control unit software systems"). It was developed by Bosch [HN99, HN00, SB05] and will be described in more detail in Section 2.1.3. OMOS is actively used to develop embedded real-time electronic control unit (ECU) software in the automotive domain. It allows to express concepts relevant for software systems for electronic control units directly by means of models. It is purely based on UML (class) models and their graphical notation, i. e. UML class diagrams.

UML class models are the most popular type of UML models [DP06]. They are used to specify the static structure of (object-oriented) software systems [RJB04]. They are often used in the analysis phase of the software development process, i. e., the problem definition phase when the problem to be solved by the to-be-developed software system is first analysed before the system is actually specified and implemented [Lar04]. However, the most popular usage of UML class models is the phase of development when the software system — i. e., one possible solution to the aforementioned problem — is actually designed and implemented [MB02]. Class models — as their name implies — define concepts which also appear in the source code of a software system — such as classes, attributes, and operations (methods).

One of the most important benefits of models — including software models in general and UML class models in particular — is their ability to abstract away from unnecessary detail and allow to focus on a certain level of desired abstraction. UML class models hence provide a formalism to specify software systems without the need to get into too much detail. For instance, when describing the static structure of a software system by means of class models, it is often irrelevant which implementation language is actually used as source code can be automatically generated from models [Fra02, KWB03]. This is possible because the model notation is formally described by means of a meta-model (see Section 2.1.1). Models can thus be interpreted by computer programs to generate other software artefacts like, for instance, source code, database schemas, or software design documentation.

Once a OMOS model is complete with respect to the expected functionality to be realised by the controller software, C source code is automatically generated from the model using a source code generation approach. Every class in a OMOS model is transformed into a C header (*.h*) and a C implementation (*.c*) file that includes the attribute definitions for each attribute and (empty) definitions for each operation defined by the model's classes. Developers then manually add to those empty operations source code which implements their behaviour. A OMOS model provides all information required to generate source code artefacts that could be found in an (object-oriented) implementation, i. e., module definitions (derived from UML packages), classes, operations, attributes, relationships between classes (and instances of these classes), and documentation. The source code that had to be manually added (inside the automatically generated empty operation definitions/bodies) realised the behavioural aspects of the gearbox controller software which could not be derived from the model (since a class model defines mostly static aspects of a software system). For example, algorithms to calculate when to shift up or down had to be manually implemented.

1.2. Aim and objectives of this thesis

In this section, the aims and objectives of the research discussed in this thesis are presented.

1.2.1. The need for parallel working on OMOS models and diagrams

OMOS is an iterative and incremental model-driven approach. Adding new software components, modifying, or deleting them is initiated on model level (and only then source code is automatically generated from the model) for each development cycle (which is described in Section 2.2.2). The modelling tool used by Bosch to create

OMOS models did not support parallel development. Modelling tasks could therefore not be performed in parallel, i. e., by more than one developer at the same time. Only source code files could be modified by more than one developer at the same time. However, since in model-driven development source code gets automatically generated from models, modifications have to be initiated these models. Only then does it make sense to manually adjust the generate source code — for instance, to implement the behavioural parts of the implementation which could not be automatically derived from the model. Modifying the source code *before* the model had been modified is, therefore, not possible.

The sequential (as opposed to parallel) nature of the OMOS model development process has the following drawbacks:

1. Before starting to modify a OMOS model, developers at one development site have to wait until the developers at other site working with this model have finished their modifications.

2. When an issue introduced at the model level is detected later during the implementation phase, developers wanting to fix this issue might no longer be allowed to modify the model because it has already been handed over to the other development site.

3. In order to understand and communicate how a model has been changed at one development site, developers have to manually document model changes (in a purely textual form) or learn about changes by simply looking at the current (possibly modified) version of the model and its previous version to figure out which modifications have been made (no tool support is available for such model comparisons).

One of the motivations of model-driven approaches to software development is the prospect of reducing the complexity of the systems to be developed by abstracting away from detailed, technical implementations to problem-oriented models. Another motivation is to increase the involvement for a range of stakeholders. This inevitably leads to a greater diversity of roles being involved in the production of models.

The aim of the research presented in this thesis, therefore, is to enable parallel working on OMOS models. In the next section, the research objectives originating from this aim are presented.

1.2.2. The objectives of the research presented in this thesis

OMOS models are created in a graphical way (as UML class models usually are in general) using UML class diagrams (see Section 2.1.1), with the attendant benefits of such visual representations, it seems appropriate that the graphical representation has to be taken into account in collaborative development scenarios. The need to take graphical representations of models, i. e. diagrams, into account is demonstrated by the results of a pilot study on the industrial model-driven approach for developing automatic gearbox controller software (introduced above in Section 1.1). The pilot study, which is discussed in Chapter 3, demonstrates the importance of diagram layout for model-based software engineering in generally by analysing a specific approach. Each diagram represents a different part or aspect of a OMOS model. Their manually created layouts are important for modellers as they convey knowledge specific to the domain meaning of the model elements depicted by diagram symbols. They have to be taken into account by approaches to parallel working on models. Therefore, one premise of this thesis is to analyse the state-of-the-art of diagram-based UML class modelling and discuss the advantages and disadvantages of existing approaches that incorporate models and their visual representations into collaborative software development processes. In addition, the findings of the analysis of the pilot study are used to further reason about requirements of and provide implementations for approaches to working collaboratively with OMOS models.

To allow for models being used efficiently, suitable for industrial-scale software development, models need to be accessible by more than one user at a time allowing them to modify them independently from each other in what can be called a *collaborative modelling process* supporting parallel working [MBZR03]. The research presented in this thesis therefore sets out to discuss and provide approaches to parallel working on OMOS models and their diagrams. Therefore existing approaches to collaborative modelling will be analysed and an new approach for working collaboratively and concurrently with UML models and their diagrams will be presented to enable collaborative modelling for the OMOS approach. Based on the need for parallel working on OMOS models, this thesis sets out to provided answers to the following research objectives:

1. Analyse the research literature to . . .

    a) identify the need for parallel working when developing models in general and diagrammatic representations of models in particular;

   b) determine whether diagrams of models contain additional semantic information through their layout;
   c) identify solution methods that others have reported that could potentially resolve the identified problem; and
   d) review available software to verify that there is no existing software that adequately solves the problem.
2. Carry out a pilot study to verify that the problem exists in the pilot study environment and to identify the extent of the problem.
3. Determine the principles on which any software should be based to provide a solution for the problem.
4. Design and implement software to carry out a proof of concept.
5. Test the solution using examples from the pilot study environment.
6. Evaluate the usefulness of the proposed solution from the expert opinion of engineers in the field.

1.3. Thesis layout

This thesis is structured as follows: In Chapter 2, the need for parallel working when developing models in general and diagrammatic representations of models in particular is discussed. An introduction to UML in general, UML class models and diagrams and to the OMOS approach to developing software for automotive electronic control units is provided. The chapter closes with a review of the research literature on concepts of parallel software evolution that could potentially enable parallel modelling and a review on existing tool support for visually merging UML class models and diagrams. Chapter 2 discusses research objectives 1a, 1c and 1d defined in Section 1.2.2.

Chapter 3 provides the findings of a pilot study carried out as part of this research to verify that diagrams of models contain additional semantic information through their layout. It therefore realises research objective 2 (see Section 1.2.2). This chapter also answers research objective 1b (a review of previous research on expressing additional semantic information through diagram layout) and answers parts of objective 1d (review available software to verify that there is no existing software that adequately solves the problem).

Chapter 4 focuses on research objective 3 (see Section 1.2.2): the principles and objectives of an approach for collaboratively working with OMOS models are defined. First, the general principles on which any software should be based to provide a solution for working in parallel with diagrams of models are defined. Then, the principles drawn from the pilot study to provide a software-based solution for working in parallel with OMOS models are defined.

Chapter 5 discusses an approach and its implementation for laying out OMOS diagrams in a semi-automatic manner supporting parallel evolution of diagrams and efficient visual model merging by preserving the meaning conveyed by diagrams. In Chapter 6, an approach (and its implementation) to merging OMOS models and diagrams is discussed. Both chapters realise research objective 4 defined in Section 1.2.2.

An evaluation of the usefulness of the proposed approach to parallel work on OMOS models is provided in Chapter 7 which reports about the results of testing the proposed solution by OMOS modellers using examples from the pilot study as requested by research objectives 5 and 6 (see Section 1.2.2).

The main part of the thesis is concluded in Chapter 8 which summarises the research presented in this thesis.

Appendix A provides further details about the OMOS modelling approach discussed by this research. Appendix B provides the presentation slides of a meeting with Bosch engineers to discuss additional semantic meaning conveyed through the layout of OMOS diagrams as identified in Chapter 3. Additional OMOS diagrams which were analysed as part of the research presented in the thesis are discussed in Appendix C. The design and implementation of the semi-automatic diagram layout and OMOS model merge tool are discussed in Appendix D.

2. Working collaboratively with software models — the need for parallel working when developing models in general and diagrammatic representations of models in particular

This chapter first provides an introduction to the OMOS approach to software development, then the need for parallel working when developing models in general and diagrammatic representations of models in particular is established by providing a review of the research literature on this topic. This review fulfils research objectives 1a (identify the need for parallel working when developing model in general and diagrammatic representations of models in particular), 1c (identify solution methods that others have reported that could potentially resolve the identified problem), and 1d (review available software to verify that there is no existing software that adequately solves the problem) defined in Section 1.2 for this research.

2.1. Introduction to the OMOS software development approach

As discussed in Section 1.2 on the aims and objectives of the research discussed in the thesis, the author of this research has been given the task to enable modellers at different development sites to work collaboratively with OMOS models. As one of the first research activities regarding this research the author looked at the OMOS documentation provided by Bosch ("OMOS User Manual" and "OMOS Primer") to gain an understanding of the OMOS approach to developing automotive software. This software development approach can be summarised as follows: OMOS is used to model the static structure (classes etc.) of electronic control unit (ECU) software systems. It is based on UML. The graphical notation for UML class diagrams is used to create UML class models (called OMOS models).

In the remainder of this section, an introduction to UML in general and the OMOS software development approach in particular is provided. This introduction provides essential information required to better reason about the topics discusses in the remainder of this thesis.

2.1.1. UML class models and diagrams

From a formal point of view, the definition of UML models in general and UML class models in particular is twofold. A model constitutes of instances of UML model elements and their representations. The model elements itself are defined by the *abstract syntax* which describes the features of these elements independently of any particular representation (see Section D.4). UML defines, for example, concepts for modelling the static structure of (object-oriented) software systems using classes, packages, generalization relationships, associations etc. The *concrete syntax* defines how a model is presented to the user, for instance, by UML modelling tools (also called CASE tools).

The UML standard [OMG10b, OMG10a] defines the abstract syntax and provides suggestions for the concrete syntax too. The both together form a modelling *language* (instead of merely a definition of the abstract syntax *or* a notation of the concrete syntax). UML is mainly a graphical modelling language, the concrete syntax used by the UML standard are *diagrams*. They contain *graphical* symbols (concrete syntax) which depict model elements (abstract syntax).

The *UML Superstructure* [OMG10b], which defines UML's user-level constructs, and the UML Notation Guide [OMG03] provide basic style guides roughly specifying the graphical (diagrammatic) representation of UML model elements. With respect to class models, the following representations are defined (not complete): Classes and packages are represented as rectangles. Classes may be further subdivided into compartments. The first compartment, which is mandatory, shows the name of the class and, if defined for the class, stereotype and tagged value definitions. The other two compartments are optional, they are used to display attributes and operations belonging to the class. Relations between classes are represented by lines or polylines. For instance, Fig. 2.1 shows a diagrammatic representation of four classes and three composite aggregations. Class diagrams are typical *line and box diagrams.* A UML (class) model can consist of an arbitrary number of class diagrams. The terms class diagram and class model are often used interchangeably in the literature, in this thesis, the term *diagram* refers to the graphical *view* of a model. A model is then referred to as the *structure* underlying a diagram.

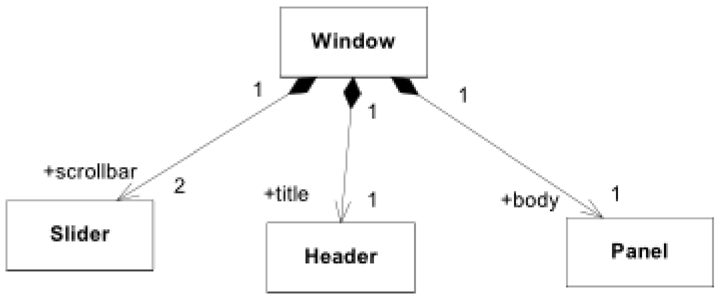
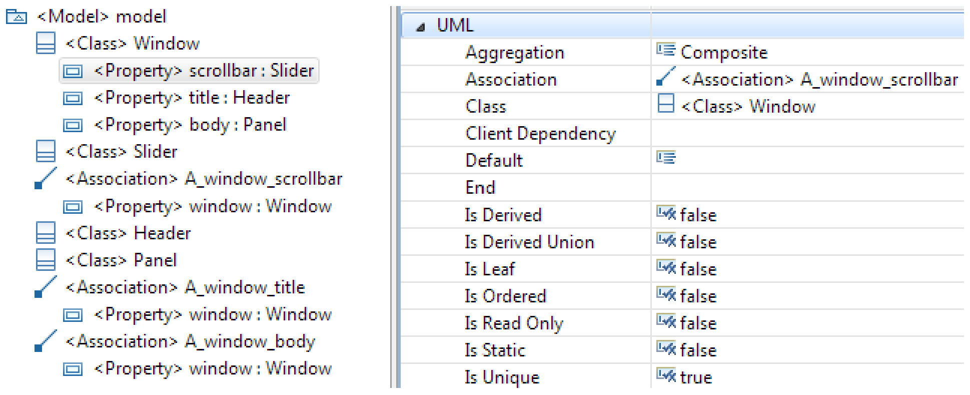


Figure 2.1.: UML class diagram showing classes and composite aggregations [OMG10b, Fig. 7.26, p. 46].

Most UML modelling tools mainly provide editing capabilities to create models by means of diagrams. User hence create or modify the underlying model by creating or modifying diagrams and their symbols.

There exist other forms of concrete syntax for UML, for instance, the *Human-Usable Textual Notation* (HUTN) [OMG04]. It allows defining UML models and other types of models by means of a textual notation. Another type of representation are trees and tables (see Fig. 2.2). They are often provided by modellings tools in addition to the diagrammatic representation of a model since they are better suited for providing an overview of a model and to present details of certain types of model elements. Creating whole models using the tree-table approach is in general not feasible because the trees become large and confusing to work with — especially with respect to dealing with relationships between model elements.



- <Model> model
  - <Class> Window
    - <Property> scrollbar : Slider
    - <Property> title : Header
    - <Property> body : Panel
  - <Class> Slider
  - <Association> A_window_scrollbar
    - <Property> window : Window
  - <Class> Header
  - <Class> Panel
  - <Association> A_window_title
    - <Property> window : Window
  - <Association> A_window_body
    - <Property> window : Window

UML	
Aggregation	Composite
Association	<Association> A_window_scrollbar
Class	<Class> Window
Client Dependency	
Default	
End	
Is Derived	false
Is Derived Union	false
Is Leaf	false
Is Ordered	false
Is Read Only	false
Is Static	false
Is Unique	true

Figure 2.2.: Tree-table representation of the model presented in Fig. 2.1 as a UML class diagram.

2.1.2. The part of UML used for OMOS

In this section, the elements of UML used to define OMOS models are discussed. Since OMOS model are UML models, the UML meta-model defines the available OMOS modelling concepts. The most important elements are: Packages, Profile applications, Element imports, Classes, Stereotypes, Generalizations, Attributes, Associations, and Operations.

Most of these elements are known from (object-oriented) software development. UML uses them to describe the static aspects of the software system being defined in the model [RJB04]. The structure and semantics of these element are defined by the UML meta-model. Chapter 7 of the UML Superstructure Specification [OMG10b, pp. 23] and Section 11.3 of the UML Infrastructure Specification [OMG10a] describe them in detail.

In general, a meta-model formally defines the elements and relationships available in models conforming to this particular meta-model. Hence, a meta-model defines the structure and semantics of model elements that can be created based on this meta-model. A model is an instantiation of the meta-model by which it is defined.

2.1.2.1. The Meta-Object Facility — UML's meta-meta-model

The formal structure defining the UML meta-model is again described by a model called *meta-meta-model*. UML's meta-meta-model is defined by the *Meta-Object*

Facility (MOF) [OMG06a][OMG10a, pp. 11]. At some point this meta-model layering has to stop in order to be able to actually instantiate meta-models, i. e. create models. The meta-meta-model, therefore, describes itself. Such self-descriptions are known as *closed meta-modelling architectures*: '*A specific characteristic about metamodeling is the ability to define languages as being reflective, i.e., languages that can be used to define themselves. The* InfrastructureLibrary *is an example of this, since it contains all the metaclasses required to define itself. MOF is reflective since it is based on the* InfrastructureLibrary*. This allows it to be used to define itself. For this reason, no additional meta-layers above MOF are defined.*' [OMG10a, p. 17].

Based on MOF, a modelling hierarchy consisting of four layers is formed (see Fig. 2.3):

Meta-meta-model (M3) Defines the infrastructure for meta-models, i. e., it defines a language to define (or create) meta-models. A meta-meta-model usually is an instance of itself. MOF represents a meta-meta-model.

Meta-model (M2) Defines a language for creating models. A meta-model is an instance of its meta-meta-model. UML is a meta-model and an instance of MOF.

User model (M1) A language for modelling information (for instance, a UML class model for gearbox controller software, see Section 2.1.3). A model is an instance of its meta-model. For instance, a UML model is an instance of the UML meta-model.

Data model (M0) A concrete instantiation of a user model. For instance, gearbox controller software running in a gearbox embedded in a car.

Additional details about the meta-object facility are provided in Section 6.2.1.

2.1.2.2. Profiles and stereotypes

UML provides mechanisms to extend the elements defined by its meta-model. This is achieved by means of profiles (see chapter 18 of [OMG10b, pp. 669]). As stated in [OMG10b, p. 679], "*[a] profile is a restricted form of a metamodel that must always be related to a reference metamodel [e. g., UML]. A profile cannot be used without its reference metamodel; it defines a limited capability to extend metaclasses of the reference metamodel via stereotypes. Profiles can be made to reference metaclasses*

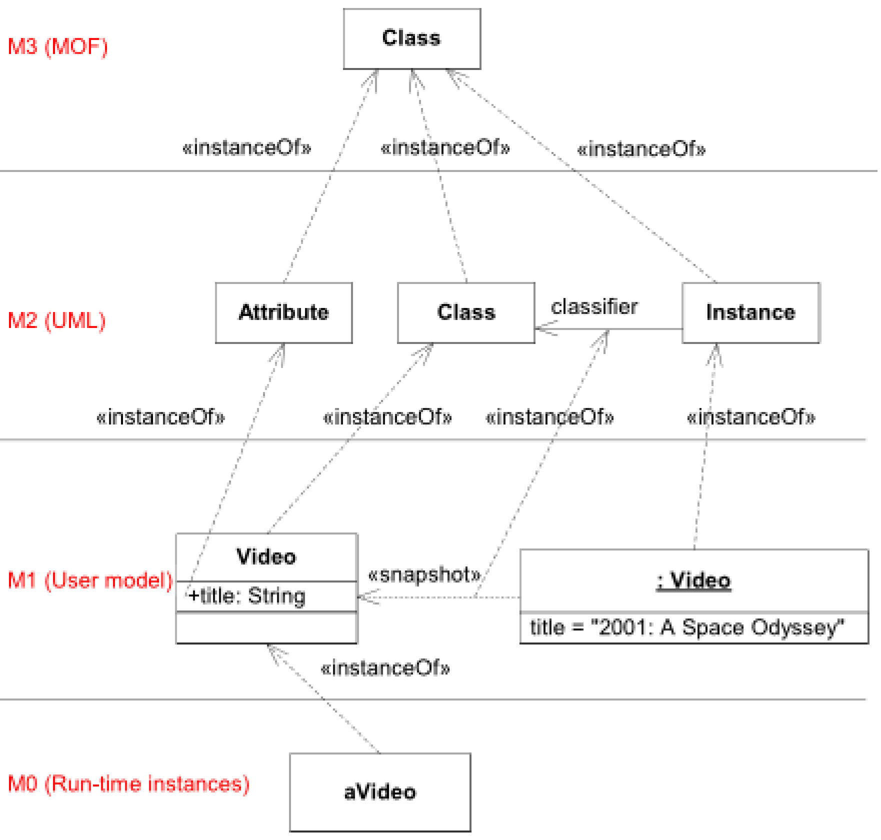


Figure 2.3.: An example of the four-layer meta-model hierarchy [OMG10a, Fig. 7.8, p. 19].

from metamodels by creating an import relationship between the profile and the reference metaclass." The term *metaclass* refers to the classes defined by the UML meta-model (see above).

A profile is not part of a UML model. It is defined outside of such models. Models use the stereotypes provided by a profile. *"A stereotype defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass"* [OMG10b, p. 689] [OMG10a, pp. 177].

For instance, a stereotype called *1-Class* is used by OMOS to annotate classes for which only one instance can exist during the runtime of a controller software system (see Section 2.1.3). The stereotype is *defined* in a UML profile [OMG10a, p. 187] which is imported into a OMOS model and *applied* to classes defined in this model.

Similar to classes, stereotypes can define properties [OMG10b, p. 689]. These properties are called *tagged values*. They are used to define additional features of stereotypes. When a stereotype is applied in a model, its tagged values become available and their values can be assigned.

2.1.3. The OMOS approach to developing software for automotive electronic control units

This section provides an introduction to OMOS (object-oriented modelling concept for electronic control unit software systems). As mentioned in Section 1.1, OMOS is used by Robert Bosch GmbH to develop electronic control unit (ECU) software in the automotive domain [HN99, HN00, SB05]. It is an object-oriented, model-driven approach based on UML.

In model-driven software development (MDSD), models are not only used as a means of visualisation and communication of software structures. In MDSD, the value of models also comes from the fact that other software artefacts, such as source code, can be automatically derived. Models are, for instance, used as input for computer programs called source code generators which, as the name suggests, automatically produce source code from models.

Depending on the information available in a model, the generated source code might be incomplete with respect to the expected behaviour of the software system to be built. This is usually the case when source code is generated from UML class models because only little domain-specific behaviour can be derived from pure class models. Since UML class models only declare operations, but do not contain any definition of their behaviour, the source code generated for operations can only represent their declarations, but not their actual behaviour. An operation's body, which represents this behaviour's implementation on the source code level, is empty and hence has to be implemented by software developers. When the source code is generated again because, for instance, the model was updated, the source code generator takes care of preserving the manually implemented parts.

With the generative, model-driven software development approach outlined above, software developers usually define all the key aspects of the to-be-built software system in the model, then generate the source code and manually fill in the missing parts of the implementation, i. e., the parts that were not defined by the model.

This approach is independent of the development method. No matter if a traditional waterfall approach or a more *en vogue* lean or agile approach is practised, the model acts as the central place to define features of the software under construction.

2.1.3.1. OMOS-based development of automatic gearbox controller software

This section focuses on how the OMOS approach is applied for developing software for automatic gearbox controllers. This software runs on so-called electronic control units (ECU) embedded in cars (shown in Fig. 2.4). These ECUs are connected via communication buses, for instance, Controller Area Network (CAN) or FlexRay, to allow for exchanging information. For example, the gearbox ECU communicates with the car engine's ECU and the car's Electronic Stability Program's (ESP) ECU.

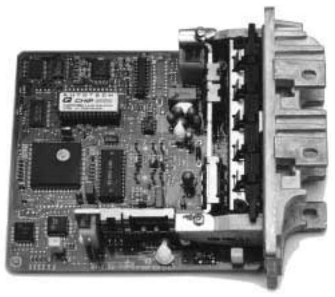


Figure 2.4.: Example of an electronic control unit used in cars.

OMOS is used to create software design and implementation models that describe the static structure of the implementation of ECU software systems. To create these models, UML class models and diagrams are used. ECU software concepts are thus represented by UML classes, operations, attributes, associations, packages, etc. A OMOS model hence provides all information required to automatically generate source code, i. e., module definitions (packages), classes, operations, attributes, relationships between classes (and instances of these classes), and documentation.

C source code is automatically generated from the model using a source code generation approach. Every class in a OMOS model is transformed into a C header (*.h*) and a C (*.c*) file that includes the attribute definitions for each attribute and (empty) definitions for each operation defined by the model's classes. Since a OMOS model defines the static aspects of a software system, source code which realised the behavioural aspects of the gearbox controller software cannot be derived from the model. It has to be manually added inside automatically generated empty operation definitions derived from the operations defined by the model's classes.

In addition, framework and infrastructure code is generated, for example, code helping to realise single inheritance for ANSI C which does not natively support such object-oriented concepts[1]. OMOS allows sub-classes to reuse or override operations inherited from their base classes and thereby enables polymorphism for the procedural C programming language. Another example of infrastructure code that is automatically generated is code that allows for class instances to communicate with associated instances. The information which instances can communicate with each other is taken from the model.

2.1.3.2. OMOS classes

As defined by UML, the UML classes used for OMOS contain operations which represent interfaces to perform a certain behaviour on an instance of the respective class. Classes also contain attributes to store their instances' state. By using aggregations, classes can define other classes to become what is called a part-class of the aggregating class. OMOS does not differentiate between class aggregation and class composition, both represent whole-part relationships between classes.

Since OMOS is used in embedded real-time environments, there are constraints concerning the size of the binaries files compiled from OMOS source code. To allow for runtime- and space-efficient mappings from OMOS models to C code and consequently to ECU binary code, ECU software characteristics and requirements have to be considered in OMOS models. Therefore OMOS defines three stereotypes (see Section 2.1.2) for classes. A class marked by stereotype *1-Class* exists at most once in a OMOS implementation/runtime, i. e., the *1-Class* stereotype marks classes as singletons [GHJV95]. Certain code optimisations can be applied for *1-Classes* when source code is generated from OMOS models. A class marked with stereotype *N-Class* can be instantiated more than once.

The following example in Fig. 2.5 shows how these OMOS concepts are used. The example shows two classes, *Engine* and *Cylinder*. Class *Engine* is a *1-Class* since a car has only one engine. Class *Cylinder* has to be a *N-Class* because an engine has more than one cylinder.

OMOS defines another stereotype for classes called *Root*. This stereotype, demonstrated in Fig. 2.6, defines the root class of a OMOS model. A root class is also a *1-Class* because exactly one root class exists in a OMOS model.

[1]The approach to enabling object-oriented concepts in C used for OMOS is conceptually similar

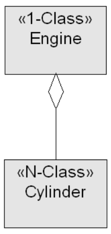


Figure 2.5.: Example of 1-Class and N-Class and whole-part relationships between OMOS classes.

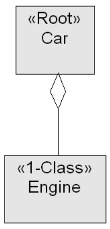


Figure 2.6.: An example root class.

As part of this thesis, the rather *ad hoc* application of stereotypes was formalized by defining a UML profile for OMOS which is discussed in Section A.2.

In OMOS, two types of hierarchies are important: class composition, i. e. whole-part relationships, and class inheritance, i. e. specialisation hierarchies. Single inheritance is used, i. e., every sub-class has one *direct* base class. Regarding class composition, each class, except the root class, is part of at least one other class, either directly or by inheritance, i. e., one of its base classes (not necessarily its direct one) is part of at least one other class. A class can be part of more than one class. Hence, several classes can contain (by means of composition) the same class.

2.1.3.3. Class hierarchies

Classes are used to represent a certain functionality that is needed in an ECU software system. Class inheritance, i. e. generalisation, is used to describe different variants of a certain functionality. A sub-class can specify additional attributes, aggregations, and associations. A sub-class can also overwrite operations defined by its base class. It is possible for a sub-class to provided a different implementation of a certain operation, i. e., functionality. Hence, sub-classes introduce variations (see

to the approach presented in [Sam02]; it is, however, not based on the same implementation.

Section A.1).

In OMOS, all base classes, except the root class, must be connected by aggregation associations (i .e., whole-part relationships) resulting in a class hierarchy that starts at the root class. Thus, every class, except the root class, is a *part class*.

2.1.3.4. Communication between classes

In OMOS, aggregation associations are usually *undirected* which means that instances of both classes (i. e., instances of the aggregating class and instances of the aggregated class) can communicate with each other. An instance can hence call public operations of the other instance. It is possible to define *directed* aggregation associations such that only instances of the aggregating class can call operations of instances of the aggregated class (aggregation associations going in the opposite direction do not make sense because the whole-class of the aggregation has to be able to communicate with the part-class).

Classes can communicate with classes which they do not aggregate, i. e., they can call public methods of instances of non-aggregated classes. This kind of communication is realized by associations. If there is more than one association between the same classes, the navigable association ends need to have distinct names. As with aggregations, associations can be directed or undirected.

2.1.3.5. Addition information on OMOS

More details about OMOS are provided in Appendix A.

2.2. The need for parallel evolution of OMOS models

As discussed in Section 1.2, the aim of the research discussed in this thesis is to provide an approach for working concurrently and collaboratively with OMOS models, i. e., UML models and class diagrams. Nowadays software is no longer developed by individual developers but by groups consisting of many developers. In order to avoid delays during development, the same software should be worked on in parallel by several developers [MBZR03]. This specific form of iterative and incremental software development [BSR03] is called *parallel evolution* [MBZR03].

With respect to the research discussed here, parallel evolution is necessary in order to allow different OMOS engineers (possibly at different development sites) to work in parallel on the same OMOS model, i. e., several engineers or groups of engineers can work on the same model at a time and do not have to wait for another group to finish working on the model before they can start working on it.

After making himself familiar with the OMOS documentation (see Section 2.1.3), the author set out to gain insight into the actual software engineering processes exercised for developing automotive software following the OMOS approach. The main objective was to understand the OMOS modellers' requirements with respect to enabling parallel evolution of OMOS models.

The results of the analysis of the OMOS development process are discussed in this section.

2.2.1. Meeting with domain experts to learn about the OMOS development process

Before the author made himself familiar with the different approaches to model and diagram evolution described in the research literature (discussed in Section 2.4), he analysed the development process of a software development project conducted by Bosch's "Automatic Transmission Control Units" engineering group. In the analysed project, automatic gearbox controller software is being developed using the OMOS approach. The goal of this analysis was to learn about the group's approach to software development and to use these findings to be able to better identify principle solutions of parallel evolution which allow OMOS modellers to pursue parallel modelling.

In order to get detailed information about the development process in general and gain insights for possible types of parallel evolution for OMOS models, the author met with ten developers from the "Automatic Transmission Control Units" engineering group for a two-day meeting at Bosch's Schwieberdingen (Germany) site in September 2005. In this first meeting, the (at this time) current approach to dealing with OMOS models and the envisaged parallel evolution of OMOS models were discussed. The author therefore inquired the Bosch engineers about how they developed OMOS models as a group situated at different development sites. The results of this meeting are presented next in Section 2.2.2.

The meeting described above was the first of a number of meetings which took place over the course of the next three years. In all meetings, the author took notes, i. e., minutes of meeting, by hand. In addition to meeting up with the engineers in person, Bosch allowed the author to contact them by phone and email at any time throughout the entire project.

2.2.2. Bosch's iterative and incremental model-driven development process based on OMOS

Models form the centre of the design and implementation phase of the software development process following the OMOS approach. Adding, modifying, or removing functionality from the software system is achieved by modifying the model first, the respective implementation (source code) modifications then follow automatically when the source code is automatically generated from the model[2]. When, for instance, new functionality (i. e., behaviour) is added to a class, a new UML operation is added to this class in the OMOS model. Then, the source code is generated (again) and the operation's implementation is filled in manually. The model contains all structural information of the actual software system and is a first-class citizen in the software development process. In order to update the software system, the model is modified first and the generated source code is adjusted afterwards. Therefore, in order to update features of the software system like inheritance relationships, associations between classes, attributes or operations, the model is updated before the implementation (source code) — which is updated as a consequence of updating the model.

The behavioural parts of the software are manually added to the automatically generated source code. Hence, the software system's design first manifests itself in the OMOS models and only then in the generated source code. Models thus represent a central part in development cycles of software projects which apply the OMOS approach. Adding new functionalities to the automatic gearbox controller software, removing obsolete, or updating existing[3] ones has to be done through a

[2]A code generator takes care of preserving manually modified source code sections in the generated files. Such source code sections are marked as *protected regions*. They are mainly used to protect the manual implementation of operations (whose signatures are declared in the model), i. e., to protect the operation's body.

[3]Updating existing functionalities requires model updates if structural aspects of the functionality have to be updated. No model changes are required when only the implementation (source code) has to be adjusted. For instance, a model has to be updated if an operation's signature or an attribute's type changes, or a new relation to another class has to be added in order to access operations of this class from within the updated operation's implementation.

model.

For the analysed OMOS project, each development cycle, which is described in greater detail in Section 2.1.3, usually takes about two to four weeks. For each iteration, a *version* is created for the model. A version reflects a certain (with respect to the software system's requirements) stable development stage. Within an iteration, the model can only be modified at one of the two development sites, Schwieberdingen or Budapest, and only by one engineer at a time. Hence, modifications can only be done in a sequential manner, before starting to work on the model and realise their modifications, engineers at one site have to wait for the engineers at the other site to finish their model modifications.

Engineers can work in parallel only on source code level, but *not on model level.* On source code level this is possible because a *shared source code repository* managed by a *version control system*[4] (VCS) is used to manage concurrent access to source files. Such software tools (for instance, *Concurrent Versions System* (CVS) [BP01] or *Subversion* [Sub]) are one of the most beneficial approaches for collaborative software development. They support storing all files comprising the implementation of a software system in a central place and manage access to and modifications of them. They also keep track of all changes done to a software system's implementation files (and any other files) over the course of its development. Concurrent versioning systems play an important role for industrial-scale software development because they enable software developers to work concurrently at the same files. When one developer is modifying a certain source code file, another developer, too, can modify the same file at the same time. When a developer sends modified files back to the shared source code repository managed by a VCS, it takes care of making the developer aware of (potentially) conflicting modifications of the same files.

Bosch's iterative and incremental model-driven development process approach outlined above allows multiple engineers to access and modify the same source code files at the same time. OMOS, however, is a model-driven approach, adding new software components, modifying, or deleting them is initiated on model level (and only then source code is automatically generated from the model). Such modelling tasks cannot be performed in parallel. Furthermore, since source code can only be manually modified within the protected regions generated only for operations (by the code generator), only the implementation of existing operations can be modified

[4]Microsoft Visual SourceSafe (see http://en.wikipedia.org/wiki/Microsoft_Visual_SourceSafe, access date: 4/12/2012).

(at source code level). It does, however, not make sense to start implementing new behaviour (i. e., new operations) on source code level without first declaring them in the model.

2.2.3. Conclusions

As discussed above, OMOS models could only be modified by one engineer at a time. This sequential way of modelling is a limiting factor for the OMOS software development process since models form its centre. It is, therefore, desirable that several engineers can work on the same OMOS model at the same time. Therefore, in order to allow for working on OMOS models in a collaborative manner, parallel evolution of OMOS models has to be supported. This topic is discussed in the next section.

2.3. Literature review on concepts of parallel software evolution

As explained in Section 2.2.2, the software tools used to build OMOS models do not allow for parallel evolution of models. Therefore, after analysing the OMOS software development approach discussed in Section 2.2, the research literature was reviewed regarding the need for parallel working when developing models in general (since OMOS is based on UML class models) and the diagrammatic representations of models in particular (since OMOS uses UML class diagrams). The literature review provided in this section fulfils research objective 1a defined in Section 1.2.

Discussing parallel software evolution inevitably leads to discussing concepts from the domain of *version/revision control* because this domain deals with approaches to enabling a group of people to work on the same electronic documents at the same time. Even though the research presented in this thesis focuses on cooperatively working with UML class models and diagrams, this section provides a general overview of approaches to cooperatively working with electronic documents, i. e., any kind of computer-created documents, be it source code, other text documents, or models. Such an artefact is known as a *configuration item* [HOS90, CW98]. It is often referred to as a file when dealing with source code, documentation, or other text documents in general. For instance, such a document could be a word processor document, a source code file, or a software model.

In the context of this thesis, “working cooperatively on a document” means that the document can be worked on in a parallel manner, i. e., two or more persons can work on the same document or copies of the same document using *different* computers to access this document. As will be discussed below, there is a difference in editing the same document and editing different copies of the same document. The latter approach allows more than one person to edit the same document at the same time to accomplish a certain task. (Each task is usually different from the tasks any other person is working on at the same time.) The advantage of working on the same document (as opposed to accomplishing the different tasks in separate documents) is that existing contents in the document can be referenced, modified, or even deleted — when separate documents are used, only referencing existing elements is be possibly.

2.3.1. Concurrent access for enabling parallel evolution

Parallel evolution means that more than one user can access a certain electronic document (for instance, a model) at the same time. Each user works on its own copy of the model. With this so-called private workspace approach each user holds a private *working copy* of a model. In contrast to the connected approach (which will be explained below), the user can modify the model independently from any other user. At some point, the private working copy is send back to the shared repository to make the updated document available to other stakeholders. A new version of the document is created in the repository. Hence, there exists more than one *version* (also called *revision*) of the same document. That is why the process of managing the integration of different versions of a document and the changes made to it is referred to as *version* (or *revision*) *control* [HOS90, CW98].

2.3.1.1. Optimistic and pessimistic concurrency control/locking

Two approaches for concurrently accessing electronic documents exists: *Optimistic concurrency control* and *pessimistic concurrency control* (or pessimistic locking) [Men02]. With the former approach, access to a certain electronic document is granted to anyone at any time. This means that any number of working copies of a document can be made at any time and there are no restrictions with respect to the possible types of modifications made to any of these working copies.

With the pessimistic approach, a single user is granted exclusive access to a certain document at a time. All other users can create working copies in order to read this document, but are *not* allowed to *modify* it. Conflicts resulting from elements being modified in contradicting ways by several users are prevented by granting *exclusive write access* to only one user at a time.

Given that many software development projects are worked on by teams which are spread across the globe, the optimistic approach seems especially beneficial for this kind of distributed software development. Each team member can work independently on its own local copy of a project (a working copy). Access to the shared repository is not required as long as the user does not want to save modifications from the working copy to the shared repository.

Since the optimistic approach does not restrict the access to shared documents, each team member can access and modify every shared document independently from any other member of the team. On the other hand, restricting the access of a shared document as done by the pessimistic locking approach can become problematic when a user forgets to unlock a document, and thus prevents other users from modifying it. The benefits of parallel work can also be contradicted by locking too many documents by one user at the same time. If a user wants to change a document already accessed by another user, he/she has to wait until it is released again, even if the user wants to modify a different part of the this document and both changes would hence not interfere.

2.3.2. Model merging — changes, conflicts, and conflict resolution

A *change* represents a modification made to a working copy of a certain document in comparison to the original version of the document from which the working copy was created. When more than one person is allowed to modify its working copy of a certain document at the same time (i. e., when the optimistic concurrency control approach is applied), *conflicts* can occur (when concurrently modified document versions are brought back together) if the same document has been modified in contradicting ways at the same time. For instance, a conflict might occur if the same line of a text document has been changed in different ways.

Then, a *merge* (also called *integration*) process is required to reconcile changes made to different working copies (i. e., versions) of the same document. During this process, the user has to manually resolve conflicting changes made to the same document in different working copies. For instance, for a text document, it has to be decided whether to choose one of the conflicting text lines or even define a new one in order to solve the conflict.

The result of the merge process is a new version of the document which contains the changes made to potentially more than one of its working copies at the same time and which has no conflicts any more.

2.3.2.1. Two- and three-way merging

Two approaches to merging electronic documents exist: *two-* and *three-way merging* [CW98]. The former takes two versions of a document into account while the latter is based on three. For the two-way merge, the two versions are the current version of a document in the shared repository and a working copy to be integrated into the repository again. In addition to these two documents, the three-way merge takes into account the document from which both documents originate. This document is also called the parent or common ancestor.

The three-way merge approach has advantages because it allows detecting certain types of changes and conflicts which cannot be detected reliably by the two-way approach [CW98]. These advantages will be explained in Section 6.3.

2.3.3. Alternative approaches to parallel software evolution

To “work cooperatively on a document” could also mean that several persons merely see (and possibly comment on) in real-time how a document evolves that is edited by a single person. However, this kind of approach is not what is of regards for this research. For instance, in order to access the same document from different computers at the same time, a desktop sharing approach could be used which allows multiple users working on different computers to see and work with the desktop of another computer (provided all these computers are connected via some sort of computer network). Hence, the application used to edit the shared document runs on a single computer and its desktop is distributed to several persons. This approach could also be simulated by passing around a to-be-shared document from one user to another (for example, via email) in order to allow users to modify and comment on it (an approach the usually works for nay type of document). Whilst this *shared workspace* [LvO92] approach allows more than one person the work on the same document in real-time, it does not yield any advantages with respect to working on several tasks in parallel on the same document.

2.3.3.1. Connected scenario

Another approach to concurrently working on models (or any other kind of electronic document) is to synchronise every user's modelling tool (or document editing application) in real-time (via computer network-based mechanisms). This synchronisation allows all users working on a shared document to access an always up-to-date version of the document. When working on a shared document using the shared workspace approach, every user's computer has to be connected in order to access the document and create a virtual workspace shared among all users. Therefore the shared workspace approach is also called *connected* or *live approach*.

A drawback of this approach is the required coordination of editing actions for effectively working on a shared document since the involved users are situated at different sites. There is most likely some interaction required between the users — possibly be means of instant messaging or telephone conferencing — in order to orchestrate their actions. For instance, the participants need to be able to communicate with each other to agree on who is the next one allowed to modify the document. A lack of orchestration would most likely override the benefits of shared document editing, i. e., working in parallel, because only one person at a time would have a chance to change the document in a meaningful way without risking these changes to be overridden by other users' changes at the same time. The risk of interfering modifications can be reduced or prevented by locking approaches — at the cost of flexibility and the possible amount of parallelism as will be discussed below.

With the connected scenario, all participants need to be connected at the same time. This might not always be possible because of users living at different time zones or computer networks (intranets) being disconnected from the Internet. The application of connected scenarios might not even be possible because of legal reasons or intellectual property-related issues. For instance, in project collaborations, deliverables are exchanged and made available to other parties participating in the project, however, the very steps required to arrive at a certain deliverable might be tried to be kept "secret" — especially for projects where suppliers provide deliverables to customers and try to protect their intellectual property.

2.3.4. Collaboration based on model element locking

UML class models consist of individual elements (e. g., package, classes, operations etc.). Instead of locking a whole model, only certain model elements could be locked by a modeller in order to get *exclusive access* to the model element for a certain, relatively short (when compared to locking a whole model) amount of time. This approach would still allow other modellers to lock and modify other model elements at the same time. This approach of course requires a connected or live scenario (see above) because every modeller's modelling tool has to be instantly notified when model elements are locked or unlocked.

However, this approach, generally referred to as graph partitioning [LvO92], is not without drawbacks: When a certain model element has to be locked, the decision about which additional model elements have to be locked as well — in order for a modeller to be able to work with the model element — is twofold. On one hand, there are additional elements that definitely have to be locked. For instance, the container (or parent) element has to be locked because the to-be-modified element's existence depends on it, or, when a reference (for instance, an association) is to be edited, the reference's source and target model element have to be locked in order to guarantee that the reference itself and the source or target element cannot be removed. On the other hand, some elements might have to be locked because the modeller wants to ensure that modifications are consistent. For instance, when setting the type of an operation parameter, the referenced type (i. e., the referenced model element) might need to be locked as well. Locking might work better for other types of documents like source code files which tend to be smaller since software system implementations usually consist of many source files. Thus, it would be more feasible to lock a single file without preventing other developers from working on another part of the implementation (which is realised in different files).

The information about which elements have to be locked is primarily taken from the UML meta-model (see Section 2.1.1). The meta-model provides information about, for instance, container/parent-child hierarchies and reference relationships. This information can be used to derive locking rules which are automatically applied when an element has to be locked. However, a challenging issue with element locking is to automatically determine which additional model elements need to be locked. On one hand, unintended modifications, made by or affecting others, have to be avoided by locking all possibly involved and affected child and referenced model elements. On the other hand, locking many elements in order to prevent undesirable effects during the modification of some of these elements leads to a model that is basically usable by only a single modeller. Hence, other modellers cannot modify the model any more and the benefits of collaborative modelling, which should be enabled by a shared modelling approach, are voided.

2.3.5. Summary

Mens [Men02] argues that existing approaches to and tools for distributed software development concentrate on text files, especially on source code since it is a key artefact in software development. These approaches work in a line-oriented fashion, i. e., a text file's lines represent the basic component used to merge different versions of the text document. Models (and the diagrams presenting them, see Section 2.1.1) cannot be handled in a line-oriented manner since they are visualised in a diagrammatic way, i. e., graphical symbols represent the elements of a diagram [OWK03a]. Even if the underlying data and file structures are provided in a textual and line-oriented way, the result of using a text-based merge would not be meaningful with respect to the merged diagrams described by the textual structure. Models, including UML (class) models, represent structured documents. Even though UML models can be persisted as text documents using XMI [OMG05], a XML dialect for persisting models, the structure of the XMI file is automatically generated by modelling tools. However, users deal with UML models at a different level (usually diagrams are used to create and visualise them, see Section 2.1.1). Having to deal with XMI concepts when merging models is not feasible because XMI has nothing to do with the model's visual representation familiar to users. Even if the user was willing to integrate model version based on their XMI representation, there is another difference from ordinary text formates like source files that makes merging XMI files rather hard: the logical structure of the XMI file is based on identifiers, not on a text block's position in a text file produced by a human being. The next section will analyse research that sets out to overcome these issues.

2.4. Literature review on collaboration approaches for visually modelling based on UML class models and diagrams

As discussed by the research literature review provided in Section 2.3 (focusing on research objective 1a, i. e, identify the need for parallel working when developing model in general and diagrammatic representations of models in particular), a collaborative modelling approach has to allow more than one party to modify (OMOS) models in parallel. As discussed in Section 2.1, OMOS models are created in a visual way using UML class diagrams and cannot be modified by more than one modeller at the same time because the available modelling tool does not support concurrent modelling. It follows that a collaborative modelling approach for OMOS has to take the models' diagrams into account. The literature review provided in this section focuses on identifying solution methods that others have reported that could potentially resolve the problem of working in parallel with diagrams of models. This section therefore focuses on research objectives 1c (identify solution methods that others have reported that could potentially resolve the identified problem) and 1d (review available software to verify that there is no existing software that adequately solves the problem) defined in Section 1.2.2.

2.4.1. Review on existing tool support for merging UML class diagrams provided by the research community

This section summarises the literature on visual model merging from a research viewpoint. Section 2.4.2 then provides an analysis of several industrial CASE tools regarding their visual model merging capabilities.

Ohst et al. [OWK03a, OWK03b, OWK04] present an approach and a prototype tool for visually merging UML class diagrams based on the Fujaba CASE tool [Nie04, KWN05, Fuj]. Two models are visually merged using a three-way merging algorithm (see Section 2.3.2) that allows to display fine grained differences between compared versions based on their common ancestor version. A "pre-merged" diagram is automatically generated. As the name suggests, this diagram contains the elements of both versions. Elements unchanged in both versions are included automatically in the diagram. Elements changed in only one version are also included automatically but in a different colour. Deleted elements are crossed out with red, for the first version, and green, for the second version. Elements that are changed simultaneously in both versions need special treatment because of potentially conflicting changes. There exist two kinds of conflict: element properties are conflictingly changed in both versions and deletion-modification conflicts which occur when an element is deleted in one version and the element is modified in the other version. To allow the user to solve conflicts, the conflicting elements are highlighted. Elements deleted in one version and modified in the other version are crossed out with grey lines and marked with a warning symbol. The changed (sub-)elements are underlined in their respective version colour, i.e., red or green. When text elements, i.e., class or package names, or attributes or operations belonging to classes, are changed simultaneously, both versions are displayed side by side using the colour of their respective version, and the user has to decide which of the conflicting versions is used for the merged version. For example, in the case of deletion-modification conflicts this means to either accept to delete the element, and thus discard the change made to the element or its sub-elements in the other version, or to reject the deletion. Similar to conflicting changes, it is possible to undo non-conflicting changes as well since changes made only in one version of the document are highlighted.

The approach described by Ohst et al. has limitations regarding the treatment of diagrams, i. e., the graphical representation of models. It does not take the original diagram layout into account, i. e., layout changes are not considered when models are merged. Instead, Ohst et al. advocate for the use of automatic layout techniques treating UML class models as directed and connected acyclic graphs. The layout algorithm lays out classes hierarchically using inheritance relationships as the hierarchy definition criterion. As stated by Ohst et al. [OWK04], the authors did not focus on a perfect solution to lay out diagrams but to provide tool support for visually merging diagrams. Hence, the automatic layout algorithm is very basic. It uses only inheritance hierarchies to calculate the positions of elements. Base classes are positioned above their sub-classes and associated classes are placed at the left or right hand side of the associating class. Ohst et al. state that only diagrams with a small number of classes and associations are be drawn nicely. Furthermore, this approach lays out a whole merged model in one single diagram, i. e., all elements of a model are depicted in one diagram. Section 2.4.3 discusses further details about approaches to automatic diagram layout and their advantages and disadvantages.

The UML class model merging prototype tool developed by Mehra et al. does not use automatic diagram layout [MGH05]. Their approach takes all the differences between two model versions into account, including changes of graphical elements. Graphical changes, such as resized or repositioned UML class symbols, are detected by the tool and displayed in the merged diagram. The merged diagram uses the layout of *one* of the to-be-merged diagram. Changes made in the other diagram are displayed and highlighted in the merged version. Added or changed elements are displayed in red. Deleted or relocated elements are surrounded by dotted lines. Although preserving the original layout of one version, this approach has major drawbacks. One drawback is that the user has to accept or reject every individual layout change. Another drawback is that diagram symbols may even overlap. The user is then forced to manually disentangle them. The authors of the tool admit that this layout technique can lead to cluttered and unreadable diagrams when complex diagrams are used and many changes are to be displayed and dealt with by users.

2.4.2. Review on existing tool support for merging UML class diagrams provided by commercial software vendors

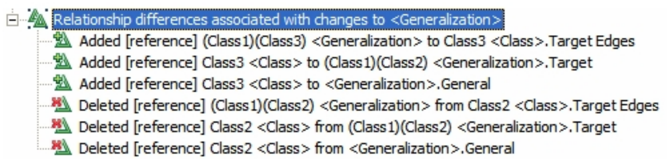
While the literature on tools support for visual model merging from a research viewpoint is summarised in the previous section, the capabilities of several industrial CASE tools regarding visual model merging are analysed in this section.

The following commercial CASE tools were analysed in order to ascertain the abilities of concurrent modelling using UML class models: IBM Rational Software Architect 6 [Let05, IBM06], No Magic MagicDraw UML (Teamwork Server Edition) [NoM06], I-Logix Rhapsody 6.2 [IL06], microTool ObjectIF [mic06], Visual Paradigm for UML (Teamwork Server Edition) [Vis], and Ameos 9.1[5] [Aon06]. While all the tools announce support for concurrent modelling, the actual support ranges from almost none to paying attention to graphical representations. Most of the tools follow an optimistic approach to concurrent modifications (see Section 2.3.1), i. e., parallel changes to models are achieved using local copies of a shared model repository. One tool uses the pessimistic approach, but access management has to be carried out manually. A pessimistic approach allows only one developer at a time to modify models (see Section 2.3.2).

In order to merge a local model copy back into the shared model repository, all the tools using the optimistic approach allow only for merging on the structural level of UML class models. This means that models in order to find conflicts between

[5]Ameos is the UML modelling tool used by Bosch to create OMOS models.

the repository and the local copy, they are compared based on their hierarchical structure. Differences among model elements are displayed in a tree-like manner: UML packages represent outer elements, followed by classes, which act as parent tree nodes of operations, attributes, and association ends. Fig. 2.7 shows an example of such a differences tree.



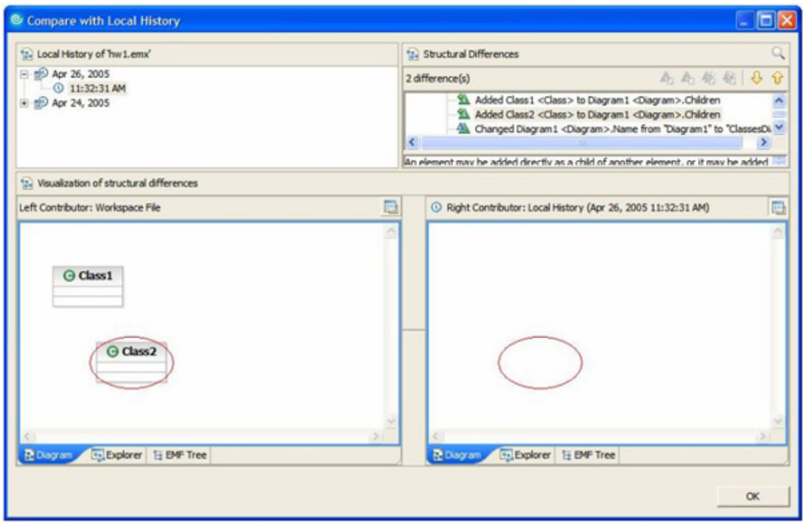
```
Relationship differences associated with changes to <Generalization>
    Added [reference] (Class1)(Class3) <Generalization> to Class3 <Class>.Target Edges
    Added [reference] Class3 <Class> to (Class1)(Class2) <Generalization>.Target
    Added [reference] Class3 <Class> to <Generalization>.General
    Deleted [reference] (Class1)(Class2) <Generalization> from Class2 <Class>.Target Edges
    Deleted [reference] Class2 <Class> from (Class1)(Class2) <Generalization>.Target
    Deleted [reference] Class2 <Class> from <Generalization>.General
```

Figure 2.7.: Example of a model differences tree.

In order to compare the current model version and the local copy to be checked in, a tree based comparison approach is used. Changes between the two versions are represented by two trees that are displayed side by side. Each tree shows one version of the differing model elements and their sub-elements. Changes made to elements in the local copy can be accepted or rejected. Conflicts are highlighted and annotated with details regarding the conflicting changes.

Only two tools, Rational Software Architect and MagicDraw, take the graphical representation of models, i. e., their diagrams, into account. The other tools do not consider at all the diagrams used to create the models. Only changes made to the class models are, therefore, recognised when the models are merged. Graphical changes at the diagram level are not considered during the merge process. Diagram changes are, therefore, lost during the merge process. A merged model version's diagrams have to be manually created.

The tools which take the graphical representation of models into account use the same above-mentioned tree structure to show differences between model versions and, additionally, display graphical changes directly in the respective diagrams. The tools consider only the position and size of elements, and changes to these properties are graphically represented. However, tree-based comparison and merge still dominates the merge process. Comparing and merging models is not done on a graphical level. Even worse, all changes of diagram symbols have to be handled (accepted or rejected) manually in the tree view, they cannot be handled in a diagrammatic way. Fig. 2.8 shows an example differences tree where graphical differences are also highlighted in the corresponding diagram. The visualisation of graphical changes directly in diagrams is a first step to take diagrammatic model representation into


Compare with Local History
Local History of 'hw1.emx'
Apr 26, 2005
11:32:31 AM
Apr 24, 2005
Structural Differences
Visualization of structural differences
Left Contributor: Workspace File
Right Contributor: Local History (Apr 26, 2005 11:32:31 AM)
Class1
Class2
Diagram
Explorer
EMF Tree
OK

2 difference(s)

Added Class1 <Class> to Diagram1 <Diagram>.Children
Added Class2 <Class> to Diagram1 <Diagram>.Children
Changed Diagram1 <Diagram>.Name from "Diagram1" to "ClassesDi

An element may be added directly as a child of another element, or it may be added

Figure 2.8.: Example of a diagram differences tree compared to the actual diagram where the differences appear.

account during the merge process. This, however, is not sufficient. Since the model are created using diagrams, users should be able to merge models in a diagrammatic way. Representing models as trees is very different from the diagrammatic representation used to create them and makes it difficult to understand differences between two model versions [OWK04].

A similar approach for differencing models and diagrams as presented in Fig. 2.8 is taken by EMF Compare[6] and Westfechtel [Wes10]. It allows to compare and merge models which are defined by arbitrary meta-models (see Section D.4). EMF Compare recently added support for comparing, but not for merging diagrams of models[7]. However, the diagram differencing approach has limitations. For instance, it does not take differences between different diagrams and their effect on the common underlying model into account.

To summarise, commercial CASE tools do not provide sufficient support for concurrent modelling of UML class models created in a diagrammatic way because they do not support for *visual* model merging, diagrams have to be merged manually [Sel03, OWK04].

[6]http://wiki.eclipse.org/EMF_Compare (access date: 4/12/2012)
[7]http://wiki.eclipse.org/EMF_Compare/CompareUMLPapyrusAPI (access date: 4/12/2012)

2.4.3. Automatic UML class diagram layout as an enabler for concurrent visual modelling

As described in Section 2.4.1 and Section 2.4.2, existing tools for merging UML class models and diagrams either impose the layout of merged diagrams on users or do not take diagrams into account at all. Since laying out merged diagrams manually is not an option for large-scale projects (due to the large number of diagrams), the approach by Ohst et al. (see Section 2.4.1) appears to be the most promising solution method for enabling parallel work on OMOS models. Their approach takes into account UML class models as well as the class diagrams used to visualise them. The diagrams are merged by applying automatic layout. However, the authors state that their automatic diagram layout has limitations. The approach to automatic diagram layout, therefore, has to be improved to become usable for (laying out merged) OMOS diagrams. This section, therefore, discusses approaches to automatic UML class diagram layout described in the research literature.

2.4.3.1. Hierarchical layout approaches

The graph layout community advocates for automatic layout of UML class diagrams. UML itself states that models can be represented as graphs [OMG10b] but does not say anything about how the graph should be laid out, i. e., how a model's graph should be visualised as a diagram. As stated by Sugiyama in [Sug02], in particular UML class diagram drawing methods are relatively underdeveloped. A first adoption of an existing automatic graph layout algorithm to the particular problems of UML class diagrams was done by Seemann in 1997 [See97] using an *hierarchical approach*. The hierarchical approach used by Seemann was originally developed by Sugiyama and his colleagues [STT81]. It is also known as the Sugiyama approach. The approach is used for drawing directed acyclic graphs [JM03]. It consists of three processing steps: layer assignment, crossing minimisation and node replacement. During the layer assignment step each node v of a directed acyclic graph G is assigned to a horizontal layer $l(v)$ such that all edges extend from a lower layer to a higher layer. The second step computes horizontal permutations of the nodes of each layer so that the number of edge crossings is minimised. Before permuting the nodes of a layer, all nodes are normalized by replacing edges that span more than one layer, i. e., connect nodes that are not placed on consecutive layers, by several consecutive edges that span exactly one layer and connect dummy nodes.

In the third step, the coordinates of the nodes are calculated. All nodes in a layer get the same horizontal, i. e., y, value. The x values of the nodes are assigned according to the node's permutation that was calculated in the second step. Coordinates of dummy nodes become bends in the original edge.

Sugiyama's hierarchical approach was enhanced by Seemann [See97] to cope with UML class diagrams. Seemann's approach is capable of handling undirected edges, for example, symmetric associations or links to association classes and notes, and direct cycles, i.e. self edges, which the original approach by Sugiyama cannot deal with. The algorithm works as follows: Nodes that are not adjacent to generalization edges are removed temporarily from the graph. Then, the first two phases of the Sugiyama algorithm are executed on the sub-graph formed by the generalisation edges. The removed nodes are then inserted iteratively in the layers. The last step calculates the node positions and routes the edges. Generalization edges are drawn as direct lines while association edges are drawn as orthogonal lines.

First analyses and experiments that examined UML class diagrams in terms of readability, comprehension and user performance were published by Purchase and her colleagues in 2001 [PCM$^+$01, WPCM02]. Based on the results of this research, aesthetic criteria for the layout of UML class diagrams were inferred [PMCC01]. Eichelberger conducted additional research on aesthetics of UML class diagram layout and developed another algorithm, called "SugiBib" [Eic02c], to automatically lay out UML class diagrams. This layout algorithm creates diagrams according to a large number of aesthetic criteria for UML class diagrams [Eic02a, EvG03a, EvG03b, Eic05, Eic06]. These aesthetic criteria are discussed in Section 4.1.1.1.

Eiglsperger et al. come to the conclusion that the hierarchical approach is often unusable in practice because class diagrams may not contain any hierarchical structure at all. For instance, class diagrams can contain only associations (but no compositions/aggregations or inheritance relations). There can even be problems with diagrams containing a lot of hierarchical information, since these hierarchies are usually not very deep. Thus, diagrams tend to get wider but not higher since many nodes are assigned to the same horizontal layer. As stated by Gutwenger et al. [GJK$^+$03], the problem with solely using inheritance trees as the hierarchy criterion used to layout a graph is that there may be other hierarchy-defining relations like associations or compositions. Eichelberger's SugiBib algorithm is a generalisation of the Seemann algorithm. It is able to cope with further layout constraints that are based on aesthetic criteria. According to Eiglsperger et al., it is the "most sophisticated hierarchic layout algorithm for class diagrams at the moment" [EKS03]. This algorithm is no longer restricted to generalisations as the hierarchy-defining edge type. Any edge type, i. e., any UML class relationship type, can be used as the hierarchy-defining edge type. In addition, it supports UML packages, also called *clustering* or *clustered (sub-) graphs* [JM03] since packages can contain nested elements like classes, notes, or packages.

2.4.3.2. The topology-shape-metrics layout approach

Eiglsperger and his colleagues [WEK02, EKS03, EGK$^+$04] suggest an alternative approach for UML class diagram layout based on the *topology-shape-metrics approach* [BDPP99]. It is called "GoVisual" [EGK$^+$04] and part of the "yFiles" graph layout tool. When compared to the hierarchical layout approach discussed above, this algorithm produces good results even for models with few or no hierarchy information, i.e., when inheritance hierarchies are missing. However, since the algorithms especially focuses on hyper edges, i. e., multiple generalisations relationships to the same base class are joined into one edge, the default algorithm is based on the directed acyclic graph that results from the generalisation hierarchy. The algorithm can work on other types of relationships, like class aggregation, composition, or association, if no generalisation hierarchies are defined in a model. As with the hierarchy approach, three phases are performed for the topology-shape-metrics approach: planarisation, orthogonalisation, and compaction. In the planarisation step, the topology of the drawing is determined. The topology is described by a *planar embedding.* A graph is planar, if it can be drawn in the plane without edge crossings. The resulting drawing divides the plane into regions, called faces. A planar embedding is a combinatorial description of the faces. It contains for each face the sequence of edges that contour it. A planar embedding implicitly defines a cyclic ordering of the edges around a vertex. To produce a planar graph from a non-planar one, dummy nodes are inserted which represent crossings to make the resulting graph planar. In this step, the mixed upward planarisation approach [EKE03] is used to assure that edges belonging to the same hierarchy criterion are drawn in the same direction. The orthogonalisation step determines the angles and bends in the drawing. To assure that a drawing is orthogonal, only angles with multiples of 90 degrees are applied. In the final compaction step, coordinates are assigned to the nodes and to the edge bends. The dummy nodes that were introduced in the planarisation step are removed.

As described by Eiglsperger & Kaufmann [EKE03] for their mixed upward planarisation approach to UML class diagram layout, it is assumed that the input graph is connected. Since UML models can contain elements that are not connected with any other element, this assumption does not have to hold for every UML class diagram and its underlying model.

2.5. Conclusions on the envisaged approach for parallel working with OMOS models

In the following, the findings from the literature on concepts of parallel software evolution discussed in Section 2.3 are combined with the findings from the analysis on parallel evolution of OMOS models presented in Section 2.2 to define objectives for an approach to enabling collaborative working with OMOS models.

OMOS models are build in a visual way by means of UML class diagrams. The approach to enabling collaborative modelling for OMOS based on UML class diagrams developed as part of this research thus has to provide solutions to the following problems:

1. In order to enable parallel working on OMOS models, it has to be possible to modify the same model in parallel. Therefore, multiple parties (at different sites) have to be allowed to work in parallel on the same OMOS model at a time allowing them to modify it independently from each other. Working in parallel on the same model will result in different versions of this model.
2. An efficient and automatic way of combining (i. e., merging) different versions of a model (modified in parallel) back into one model has to be provided.
3. The diagrams of models have to be taken into account by any approach for parallel working on OMOS models. It is not sufficient to take only models into account without considering their graphical representations (i. e., the diagrams used to create and visualise the models).
    a) Versioning of diagrams is currently not sufficiently supported by existing version management systems. Existing CASE tools offer little or no support for versioning and working in parallel [Sel03]. This is mainly because the format of stored diagrams differs from their actual visual presentation format in CASE tools [Ohs02]. Existing visual approaches to and tools for merging UML class models and diagrams either force the user to

manually lay out merged diagrams or do not take diagrams into account at all during the merge process.

4. Tool support for *visual* differencing and merging must be available to OMOS modeller in order to allow for parallel modelling: Since OMOS models are created in a diagrammatic fashion, the UML class diagrams representing the models have to be taken into account too (in addition to the underlying models). Therefore the merge approach has to take diagrams into account too. As discussed in Section 2.4, existing tools for visually merging UML class models and diagrams either did not take diagrams into account at all, burdened the modellers with manually resolving diagram layout conflicts (i. e., diagrams have to be manually uncluttered), or used automatic diagram layout to enable efficient merging of diagrams. Only the latter approach (by Ohst et al., see Section 2.4.1) appears to be a viable solution method for parallel working with OMOS models. The former two approaches are not efficient as they require modellers to merge diagrams manually.

2.6. Chapter summary and outlook

In this chapter, an introduction to the OMOS approach to software development was provided and the need for parallel working when developing models in general and diagrammatic representations of models in particular was established from the research literature to fulfil research objectives 1a (identify the need for parallel working when developing model in general and diagrammatic representations of models in particular), 1c (identify solution methods that others have reported that could potentially resolve the identified problem), and 1d (review available software to verify that there is no existing software that adequately solves the problem) defined in Section 1.2.

In the next chapter, a pilot study on an industrial software development project which used the OMOS approach is presented.

3. Pilot study on conveying additional semantic information in diagrams of models through their layout

This chapter reports about a pilot study conducted as part of this research to determine whether diagrams of models convey additional (inherent) semantic information through their layout.

In Section 3.1, the motivation for conducting the pilot study is provided. In Section 3.2, the research literature related to the study is discussed, i. e., a review of the research literature on conveying additional semantic information in diagrams of models through their layout is provided. Section 3.3 provides an outline of the actual study as well as an introduction to the scientific foundations of pilot study research. The study's findings are discussed in Section 3.4 and Section 3.5 and summarised in Section 3.6.

This chapter focuses on research objective 1b (determine from the research literature whether diagrams of models contain additional semantic information through their layout) and research objective 2 (carry out a pilot study to verify that diagrams actually convey semantic information in the pilot study environment and identify the extent of the problem) defined in Section 1.2.2.

3.1. Motivation of the pilot study on semantic information conveyed by diagram layout

As discussed in Section 2.1.3, OMOS models are created in a diagrammatic way. Diagrams therefore need to be taken into account by any approach providing a solution for enabling parallel work on OMOS models (which is one the main objectives of this research). As stated by the research literature review in Section 2.4, a popular approach for enabling parallel evolution of visual models is automatic diagram layout. With this approach, there is no longer a need for merging diagrams because they are laid out automatically according to the underlying (merged) model (resulting from merging two versions of a model which evolved in parallel). Based on the findings from the research literature, an approach based on automatic layout as an enabler for parallel work on OMOS models was discussed with Bosch engineers. Their reaction is presented in the following section.

3.1.1. Domain experts reject automatic diagram layout

After the author made himself familiar with the software development process followed by Bosch's "Automatic Transmission Control Units" engineering group (described in Section 2.2) and the different approaches to model and diagram evolution described in the research literature (discussed in Section 2.4), another meeting with OMOS modellers took place in March 2006. The main objective of this meeting was to discuss possible ways forward with respect to enabling parallel work on OMOS models. The same ten Bosch engineers (from the "Automatic Transmission Control Units" engineering group) who participated in the first meeting (which took place on September 2005, see Section 2.2.1) participated in this meeting (not all of them participated all the time, though). It took again place at Bosch's development site at Schwieberdingen.

Based on the findings from the research literature regarding support for parallel evolution of models and diagrams provided in Section 2.4, the author presented the idea of using automatic diagram layout to enable parallel work on OMOS models to the Bosch engineers. Much to his surprise, they objected to this idea!

Asked why they oppose automatic diagram layout, the modellers argued that, in their experience, automatic layout would not allow them to lay out diagrams the way they intended, it would destroy the effort they put into manually laying out diagrams. The engineers stated that the layout is important for them: according to the engineers, their OMOS diagrams contain semantic information which is solely conveyed through their layout. In order to demonstrate their understanding of domain-specific meaning inherently conveyed through the layout of their diagrams, modellers used the diagram shown in Fig. 3.1 for explanation.

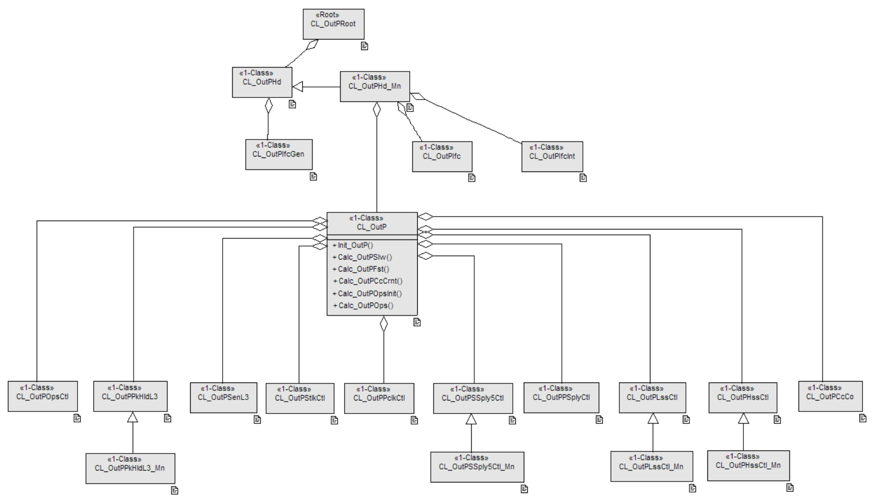


Figure 3.1.: OMOS diagram “OutP_ObjectModel” used by Bosch engineers to demonstrate domain-specific knowledge conveyed through diagram layout.

The modellers explained that, in this diagram, classes representing the high-side current sensor controller (class CL_OutPHssCtl) and its CL_OutPHssCtl_Mn sub-class (providing monitoring capabilities) are positioned next to the classes representing the low-side current sensor controller (CL_OutPLssCtl) and the respective monitoring-enabled sub-class (CL_OutPLssCtl_Mn). According to the OMOS modellers, these classes are semantically (very closely) related in the gearbox controller software domain and thus should be positioned in close visual proximity.

The modellers hence use secondary notation called *semantic grouping* (see Section 3.2.3) to express the classes' close semantic relationship through diagram layout features (even though they were not aware of the fact that this is described as semantic grouping by the research literature).

The fact that the classes are semantically related is not expressed in the respective OMOS model itself. However, the classes are documented (using free-form text). The documentation of class CL_OutPLssCtl and CL_OutPHssCtl reads:

- *“Description of the class CL_OutPLssCtl: With this class the other functions of this system are able to calculate and update the low side information. For example it is possible to get information if selected low side is available. [...]”*
- *“Description of the class CL_OutPHssCtl: This class provides the high side control information (calculation and updating of high side control). [...]”*

The modellers argued that the documented domain knowledge is expressed in the diagrams by visually grouping semantically related class symbols according to the domain meaning of the classes they depict.

The minutes of the March 2006 meeting, prepared by Bosch engineer Uwe Maienberg, highlight the OMOS modellers' issues with automatic layout and semantic information conveyed through the layout of diagrams: *"Predefined layout schema [as used by automatic layout algorithms] might destroy the consciously chosen relation of [diagram] elements to an extend that these relations are no longer visually apparent. Therefore, a suitable approach should be defined as part of the dissertation that eliminates these drawbacks. The approach has to preserve the character [or nature] of the [manually laid out] diagrams."*[1]

3.1.1.1. Consequences of the domain experts' rejection of automatic diagram layout

With respect to the research presented here, the OMOS modellers' rejection of automatic diagram layout as an approach for enabling parallel work on OMOS models, had two important consequences. Firstly, another review of the research literature was conducted in order to identify previous research on conveying additional semantic meaning through the layout of diagrams. As a second consequence, a pilot study was conducted in order to analyse existing OMOS projects to test OMOS diagrams against the layout phenomena conveying semantic meaning as pointed out by Bosch engineers and as discussed by other research.

Existing research on conveying additional semantic information in diagrams of models through their layout is identified by the literature review provided in Section 3.2. The pilot study is discussed in Section 3.3 and following sections.

[1]The original minutes of meeting by Uwe Maienberg read (in German): *"Durch ein vorgegebenes Layout-Schema kann die Relationen der Elemente, die der Entwickler bewusst gewählt hat, zerstört werden und damit nicht mehr visuell ersichtlich sein. Die Dissertation soll sich u.a. damit beschäftigen, diese Nachteile durch einen geeigneten Ansatz zu beseitigen. Dieser Ansatz soll den Charakter eines Diagramms erhalten."*

3.2. Literature review on conveying additional semantic information in diagrams of models through their layout

This section provides a review of the research literature on diagram layout and expressing additional semantic information in diagrams of models through their layout. As will be shown by the literature review, previous research identifies and acknowledges that modellers use layout of diagrams (including diagrams of UML models) to convey additional semantic information. The research presented in this literature review serves as the basis for the pilot study on expressing additional semantic information in diagrams of OMOS models presented in Section 3.3 and following sections.

3.2.1. Semantic meaning conveyed through the layout of diagrams of models

As argued by Hendrickson et al. [HJvdH06], creating diagrams and thereby building the underlying software models can be regarded as a creative process. Sugiyama describes the combination of individual elements of a diagram and rules for arranging elements as a *diagram language*, i. e., the syntactic grammar [Sug02].

As stated by Koschke [Kos03], semantic layout, i. e., the meaning of diagram elements in the application domain, plays an important role for software engineers. Batin et al. [BFN85] argue that users apply their implicit and intimate knowledge about the semantics of the application domain when laying out diagrams. The human perception of diagrams thus depends on a diagram's application domain. This findings is further supported by research on UML class diagrams conducted by Tilley & Huang [TH03]. According to them, the modellers' domain knowledge has an important impact on a diagram's spatial layout.

Eichelberger [Eic03] states that in UML, the size of a node, for instance, class or packages symbol, does not say anything about the magnitude of this element. UML introduces elements that can contain nested elements, e. g., packages can contain other packages, and classes can contain so-called inner classes. According to Eichelberger [Eic05], this introduces semantics that can be dishevelling. Furthermore, UML defines neither the meaning of proximity nor spatial ordering of model elements. Thus, depending on the user's interpretation, both meanings can vary widely.

3.2.2. The UML class diagram layout guidelines and diagramming as a creative process

UML defines a set of layout guidelines for its diagrammatic notation [OMG03, OMG10b]:

1. The layout should expose hierarchical structures if such structures exist;
2. Hierarchical structures should be drawn horizontally;
3. Inheritance relationships should be drawn in vertical direction with base classes above sub-classes;
4. Symbols should appear close to connected symbols like association classes, constraints, or comments;
5. Different edges belonging to the same relationship hierarchy should go into the same direction; and
6. Symbols should not overlap.

However, the UML notation specification does not impose any rules on how to structure a diagram, i. e., no rules exist on where diagram elements have to be positioned. As long as the UML class model syntax rules are followed, elements can be arranged in any way in a diagram. A diagram, as one of the structural elements of the UML notation, can be seen as a canvas that holds the graphical elements used to depict a certain aspect of a software system. For instance, when defining a class inheritance hierarchy, a subclass can be placed at any position (above, below, on the right, on the left etc.) in a class diagram, their is no rule to place it, for example, below its base class. This kind of layout is suggested, but not enforced by UML's guidelines. Elements may even overlap.

It follows that a certain semantic fact can be expressed in a UML class diagram in many different ways. Since there are no rules for a standardised layout, layouts may vary significantly between different users [PCM+01]. There is no layout and notation that can be seen as the standard one that should be favoured. As stated by Eichelberger [Eic05], the concrete layout of a UML class diagram is one degree of freedom when creating class diagrams. The UML Notation Guide [OMG03] confirms that the freedom of layout that is inherent to the graphical notation of the UML is intentional: *"Dynamic tools need the freedom to present information in various ways and we do not want to restrict this excessively. In some sense, we are defining* *the 'canonical notation' that printed documents show, rather than the 'screen notation'. [...] We have not sought to eliminate all the ambiguity that some of these presentation options may introduce, because the presence of the underlying model in a dynamic tool serves to easily disambiguate things."*

Even when following the UML layout guidelines (defined in [OMG03] and [OMG10b]), the diagram language (as defined by Moody [Moo10]) for UML class diagrams provides a high degree of freedom when creating class diagrams. The graphical notation of class diagrams allows to express the same semantic fact in different ways. Results of diagram layouts evaluations from a comprehension point of view show that layouts of class diagrams may differ depending on the task or problem that is described by a certain diagram [PCM+01]. Thus, a certain element type could be visualised in different ways to realise different tasks. Even though Moody [Moo10] does discuss about how (UML) diagrams can be effectively visualised, he criticises the UML layout guidelines because they do not provide any design rationale for any of its graphical conventions, it simply defines symbols and provides examples on how to lay them out. According to Moody, those guidelines are debatable: *"Just putting information in a graphical form does not guarantee that it will be worth a thousand of any set of words."* In addition, he criticises that UML diagrams do not provide any ways of dealing with the complexity of the models they depict. Moody argues that UML does not define rationales for the chosen notation and states that especially the UML class diagram notation has flaws regarding the meaning of certain layout constellation.

This finding is in line with earlier research conducted by Purchase et al. [PCM+01] who argue that little research has been performed on the readability and understandability of diagram layout. Eichelberger [Eic02a] argues that the diagram layout guidelines provided by the UML standard do not take diagram readability aspects into account. As a result, there is no common agreement on the criteria of readable and understandable layout of class diagrams.

3.2.3. Secondary notation

Secondary notation is defined as *"layout or graphical cues which are not part of the formal notation"* [Pet95], for instance, adjacency, clustering, white space, position, symmetry, or colour [PCA02]. In the context of this research the formal notation is UML's class diagram notation (see Section 3.2.2).

Purchase et al. [PMCC01] discuss the results of experiments on UML class and composition diagram comprehension. These results show that the understanding of diagrams is not merely related to aesthetic diagram layouts but to the *semantic grouping* of related objects. For example, it could be desirable to position semantically related subclasses in an inheritance hierarchy close together even if this means to disregard some aesthetics criteria. The grouping of semantically related elements, therefore, has to be recognised as an important layout aspect. The absolute position of graphical elements is not as important as their adjacency to other, related elements. Eichelberger [Eic02a] concludes that a lot of diagrams seem to follow inherent rules when positioning classes and relations.

As stated by Tamassi et al., a diagram is readable if its meaning is easily captured by the way it is drawn [TBB88] and supports the capability to clearly communicate information about the conceptual structure [Tam85, Sug02]. Eiglsperger et al. [EKS03] looked at another aspect of the layout aesthetics of class diagrams, the user performance. User performance is the users' ranking of aesthetic criteria of class diagrams. It is only indirectly linked to the readability of a diagram, i. e., a user may prefer a certain visualisation even though it decreases readability. Thus, a diagram is more readable for a user when it matches with her or his preferences.

Larkin & Simon [LS87, pp. 2] state that the position, grouping, and proximity of elements in node-link diagrams play an important role. These finding are backed by the *gestalt theory* [MF93] that was one the first theories on the human perception of visual representations. The *law of proximity* in gestalt theory says that regional closeness of elements are grouped by our mind and seen as belonging together. The *law of similarity* says that our mind groups similar elements to an entity. Form, colour, size, and brightness of elements constitutes this similarity. On the other hand, when examining the visual proximity of objects, Petre [Pet95] found that placing unrelated elements close to each other leads to the misinterpretation that they are semantically related.

Ware et al. [WHF93] consider the problem of semantic clustering as the most crucial issue in graph layout, and the meaning of nodes and edges is regarded as the most important criterion for laying out information by means of diagrams.

3.2.4. The mental map of a diagram

When users create or read a diagram, they learn about its structure and understand its meaning. The effort of becoming familiar with a diagram or drawing has been termed "building a mental map" by Eades et al. [ELMS91]. They suggest that the orthogonal ordering has an important influence on preserving the mental map and should thus be kept stable. Misue et al. [MELS95] explain that the most important concepts for the mental map are orthogonal ordering, proximity relations, and topology. According to them, the mental map of a diagram refers to the model of this diagrams that a modeller has in mind. They introduce three mathematical models for what is important in order to preserve the mental map of a drawing when it is modified. Theses models are used to evaluate to which extend automatic graph layout algorithms preserve the mental map of a graph drawing.

The most basic classification of the mental map is the orthogonal ordering of diagram elements, i. e., up, down, left and right of a diagram. Thus, orthogonal ordering refers to the concept of positioning graphical elements on a diagram's canvas. Preserving the directional relations between model elements when a diagram is modified is desirable because it preserves the user's mental map of a diagram.

Another classification of the mental map is the topology of a drawing. The topological structure of a drawing is represented by the dual graph that results from the drawing's curve set. This curve set is represented by the edges of a diagram. Curves divide the plane into subregions, called *faces*. If the graph is connected, then subregions do not contain vertices. If the graph drawing is not planar, i. e., if it contains edge crossings, a planar graph drawing can by created by temporarily replacing edge crossings with pseudo-vertices. A face is defined by the circular list of the edges and vertices that results from the clockwise traversal of the boundary of the face. Then, the dual graph of a graph drawing D is the graph that represents the faces of D as its vertices where subregions that share a common border in D, i. e. adjacent faces, result in an edge between the faces corresponding to these subregions. Two visual representations have the same topology if they have the same dual graph. Preserving its topological structure when changing a drawing helps to preserve the mental map.

Bridegman & Tamassia [BT00] suggest to compare the angles between straight connections between all pairs of elements on the old and new diagram layout since it is a more gradual measure than just taking the ordering of elements into account. Furthermore, it considers the intuition that elements that are further apart can be allowed larger absolute movements relative to each other but will still result in the same angular move. The authors suggest that the change of angles between elements is more significant for the user's mental map if the positioning is diagonal rather than if two elements are horizontally or vertically on the same level, i. e., side by side or on top of each other. This suggestion contradicts with the orthogonal ordering argument, because changes that influence the orthogonal ordering are considered less significant for destroying the user's mental map.

Mandelin et al. [MKY06] present an approach to detect similar elements in different, unrelated diagrams. Beside using semi-semantic information, for example, vertex and edge types and vertex names, they use visual information, for example, position, shape, and colour of diagram elements, to reason about similar elements in different diagrams. The decision to use visual information is based on the finding that similar elements in different diagrams were placed at similar locations.

3.2.5. Summary

The findings from the research literature reviewed in this section confirm the importance of diagram layout because modellers use diagrams to convey additional semantic information through their layout and because the layout constitutes their mental map of a diagram. Based on these research findings, the pilot study discussed in this chapter sets out to determine whether for real-world OMOS diagrams convey additional information through their layout.

3.3. Introduction to the pilot study

In this section, the foundations and prerequisites for and the approach to conducting the pilot study (on additional semantic information conveyed through the layout of diagrams of OMOS models) are discussed. First, an introduction to the scientific method of pilot study research in general is provided. Then, the actual research plan for the pilot study conducted as part of this research (i. e., the study's "blueprint") is presented. This section will discuss the objectives of this pilot study, how the analysed OMOS diagrams were chosen, and which diagrams were provided by Bosch.

3.3.1. Pilot study research as a strategy of inquiry

The research technique chosen for conducting the research presented in this chapter is a pilot study approach. Pilot (and case) study research is an accepted research strategy in information systems [Cav96] for studying the use of information technology in organisational contexts [DSB98]. It is used to collect *“data with which to develop grounded theory”* [Lub03] and to identify themes particular to the analysed case. *“As a research strategy the case [and pilot] study research method is a technique for answering who, why and how questions”* [Lub03]. It is therefore particularly suited for the type of qualitative information empirically analysed by this research. However, Yin argues that gathering multiple evidence does not necessarily prove (or disprove) a theorem [Yin03]. This issue is discussed in more detail in Section 3.3.3 on the study’s validity and threats to it.

In order to successfully conduct empirical research such as a pilot study, a *research design* has to be defined. Yin describes it as *“the logical sequence that connects the empirical data to a study’s initial research questions and, ultimately, to its conclusions”* [Yin03]. The components of research designs for pilot (and case) studies are: the study’s questions and propositions (i. e., a definition of what questions to study), its units of analysis (i. e., a definition of what data is relevant), the logic linking the data to the propositions, and the criteria for interpreting the findings (i. e., a definition of what data to collect and how to analyse the results) [Yin03]. According to Yin, the prior development of theoretical questions and propositions helps to guide data collection and analysis. Each research question and proposition therefore directs attention to something that should be examined within the scope of the study.

The unit of analysis is related to the fundamental problem of defining what the pilot study is actually going to analyse. It is therefore related to the research questions and propositions, selection of the appropriate unit of analysis will occur when the research questions and propositions are accurately defined [Yin03].

Yin argues that previous research described in the research literature needs to be taken into account and compared when defining the study’s research questions and propositions and its units of analysis [Yin03].

For the actual pilot study presented here, these components are discussed in Section 3.3.2.

Another important component of pilot (or case) studies is *validity.* It is used to ensure that the interpretations of the data collected by the pilot study will be both reliable and valid. In order to construct validity, it is important to use multiple sources of evidence and create chains of evidence during a study's data collection phase. Furthermore, experts in the field should review the study's outcome [Yin03].

There are mainly two types of validity: *internal* and *external*. Internal validity *"reflects the extent to which a causal conclusion based on a study is warranted. Such warrant is constituted by the extent to which a study minimizes systematic error (or 'bias')"* [Wikb]. According to Yin, internal validity is concerned with *"whether there is sufficient evidence to support the [study's] claim"* [Yin03]. External validity *"is the extent to which the results of a study can be generalized to other situations and to other people"* [Wika]. According to Yin, it *"deals with the problem of knowing whether a study's findings are generalizable beyond the immediate study"*; theoretical findings presented in the research literature should be used to prove external validity [Yin03].

The validity of the pilot study conducted as part of the research presented here is discussed in Section 3.3.3.

3.3.2. The pilot study's research plan

After the theoretical foundations of pilot study research have been discussed in Section 3.3.1, the components of the actual pilot study conducted as part of this research are discussed in this section.

As discussed in Section 3.1, when automatic diagram layout was suggested as an approach for enabling parallel work on OMOS models, the OMOS modellers objected to this suggestion. They argued that, in their experience, automatic layout destroys their manually created diagrams. According to the modeller, the diagrams' layout, however, conveys important semantic information, and, therefore, must not be destroyed.

Using a OMOS diagram (discussed in Section 3.1 and shown in Fig. 3.1) to illustrate their issues with automatic layout, modellers explained to the author examples of what they considered important inherent information conveyed through the layout of diagrams. In the example diagram, modellers visually grouped class symbols whose classes are semantically related in close visual proximity.

As reported in this chapter, a study was conducted to analyse OMOS models to determine whether and, if so, to which extend OMOS diagrams convey additional meaning through their layout. The overall goal of the study was to determine whether OMOS diagrams do convey inherent domain-specific information through their layout. Therefore, rather than developing a new theory on meaning conveyed through diagram layout, the study tests the theory that OMOS models convey additional semantic information through the layout of diagrams.

In contrast to an explanatory approach, an exploratory approach is used to answer who, where, and when questions, an explanatory approach is taken by this study, i. e., what and why questions are answered. The exploratory approach would not help to answer this study's questions (defined in the next section).

3.3.2.1. The study's questions and propositions

To conduct the study, the research literature was consulted in a first step to review previous research on semantic meaning only expressed through the layout of diagrams. The findings of this literature review are presented in Section 3.2. They confirm that additional semantic information is conveyed through the layout of diagrams. Then, real-world OMOS models and their diagrams were analysed in order to learn how OMOS models are build by means of diagrams and to learn about the diagrams' content and layout. (Details about the analysed models and diagrams will be discussed in the following sections.)

In order to direct attention to the nature of the analysed diagrams' layouts which should be examined within the scope of the study, propositions and questions are defined. The study answers those questions and prove (or disprove) the propositions.

Based on the information provided by OMOS modellers regarding the need for grouping symbols of semantically related classes (discussed in Section 3.1) and based on the findings from the research literature, detailed study questions and propositions are defined (see below).

The study's propositions and questions are answered based on the findings of the diagram analysis. The findings and conclusions are provided in Section 3.4, Section 3.5 and Section 3.6.

3.3.2.1.1. Propositions This study considers the following propositions:

- Proposition 1: Modellers convey semantic information through the layout of diagrams of OMOS models.

- Proposition 2: The importance of layout can be observed by the fact that layout structures will be preserved in consecutive diagram versions.

These propositions will be investigated in the following sections.

3.3.2.1.2. Questions The study is going to investigate the following questions:

- Question 1: How are diagrams used to visually construct OMOS models?
- Question 2: Which, if any, particular semantic information (specific to the model's domain) are implicitly conveyed by means of diagram layout?
- Question 3: Which kinds of diagram symbols convey additional domain-specific knowledge through their of layout?
- Question 4: Which layout features are considered important by OMOS modellers in order to express implicit domain-specific knowledge through the layout of diagrams?
- Question 5: To what extent do diagrams convey domain-specific knowledge through their layout?
- Question 6: Is the mental map of diagrams important? What drives diagram layouts changes in consecutive diagram versions?

Questions 1 to 5 focus on proposition 1, and question 6 focuses on proposition 2. A summary of the answers to theses questions and the evidence gathered by the study in order to answer them is provided in Section 3.6. Details of the evidence gathered as part of this study to investigate the above questions is provided in Section 3.4 and 3.5. The evidence to answer question 1 is provided in Section 3.4. The evidence for answering the other questions (2 to 6) is covered in 3.5.

3.3.2.2. The study's units of analysis

In this section, the elements on which the study focuses are defined based on the study's questions and propositions (see Section 3.3.2.1). As will be explained in detail below, OMOS is a visual modelling approach. Therefore, the main elements the study focuses on are OMOS models and especially their diagrams, namely UML class diagrams. The remainder of this section explains how evidence was gathered for this pilot study.

3.3.2.2.1. The analysed models: Bosch provided two OMOS models used to build automatic gearbox controller software[2]. These models represent the study's units of analysis. The OMOS models are part of model-driven software development projects dealing with software for automatic gearbox controllers (see Section 2.1.3). As explained in Section 2.1, OMOS models are visual UML class models, UML class diagrams are used to define the static parts of the modelled software systems. The models are part of the solution space (rather than problem-space models like, for instance, requirement models). With respect to the software development cycle, they are of course created *before* the implementation (i. e., source code). Large parts of the implementation are then automatically derived from the OMOS models which are turned into source code by means of automatic source code generation.[3]

Bosch provided the author of this research with a licence for the Ameos modelling tool used to create to models. It was, therefore, possible to access the models in the same way the OMOS engineers did and without any limitations.

The first OMOS model provided by Bosch is called "Power Supply Input/Output and Gearbox Operation Modes." It deals with managing the gearbox's mode of operation (for example, standby, cruise, or manual) and mode switching. Furthermore, this model deals with accessing various information of a car, for instance, transmission input speed recognition, the transmission input speed signal and gradient, automatic transmission fluid temperature, transmission output speed signal and gradient, (de)activating the power supply output, and monitoring of selected power supply output.

The second OMOS model provided by Bosch is called "Adaptive Shifting Strategy System" (ASIS). It deals with adapting the automatic gearbox's shifting schema (i. e., calculations determining when and how to shift gears) to the driving habits of the driver, i. e., to adapt shift points and so on according to the selected driving style (for example, economic or sportive).

[2]From the minutes of the March 2006 meeting with Bosch engineers at Schwieberdingen: *"To analyse the OMOS development process (models and diagrams), it is required to analyse different version of a model. RB [Robert Bosch GmbH] will send those versions to FH Zwickau [University of Applied Sciences Zwickau, the author's employer at that time]."*
Original note (in German): *"Zur Analyse des OMOS-Entwicklungsprozesses (Modelle und Diagramme) ist es notwendig, Modellversionen in größeren Zeitabständen zu haben. RB sendet entsprechende Versionen an die FH Zwickau."*

[3]The systems are implemented in C and use a framework which allows to realise object-oriented concepts (e. g., polymorphism) in native C. In order to automatically generate syntactically correct source code, the OMOS models have to be complete with respect to class definitions including operations, attributes, and (inheritance, composition, and association) relationships between classes.

The two models are independent from each other, i. e., they are not connected. However, on source code level, the ASIS part of the software uses functionality provided by the "Power Supply Input/Output and Gearbox Operation Modes" to get access to various information about a car's state.

Both OMOS models are created and consumed by software engineers who are experts in the domain of automotive controller software with a strong background in embedded software development for automatic gearbox controllers. Regarding the creation of OMOS diagrams, their layout and secondary notation (discussed in Section 3.2.3), the modellers share the *"same perceptual and cognitive hardware and software"* [Moo10]. Following practices from the extreme programming and agile software development community [BA04], all developers "own" all the source code [Gre02][SG09, pp. 183] *and* all parts of the models they work with. In order to increase the awareness of the team with respect to models and source code, an effort is made to make all the team's members familiar with and responsible for all parts of the software on both model- and implementation-level.

While source code could potentially be modified in parallel, the analysed models were *not worked on in parallel* — enabling parallel modification (i. e., collaboration) on the model level is one of the main objectives of the research this thesis reports about.

3.3.2.2.2. Modelling by means of diagrams: The analysed OMOS models represent the static structure of the automatic gearbox controller software. OMOS is based solely on UML. The analysed OMOS models were entirely created by means of UML class diagrams. This UML diagram type defines UML's visual notation for modelling the static structure of software systems [OMG10b, p. 23]. The UML modelling tool (CASE tool) used to create the models, called *Ameos*, provides only tooling for creating UML models by means of diagrams[4]. This feature is emphasised by the fact that Ameos was formerly called *Software through Pictures*.

Since the study is mainly dealing with diagram layout, it is important to note that the analysed OMOS diagrams were manually created by modellers who were also involved with the validation of the study's findings (as will be explained later).

[4]A free and open-source version of Ameos, called *OpenAmeos*, is available at http://www.openameos.org (access date: 4/12/2012). It is similar, but not identical to the Ameos modeller which Bosch uses; for instance, the OMOS extension (UML profile, see Section 2.1.2.2) and the OMOS source code generation templates are not part of the OpenAmeos distribution.

Diagrams can be freely laid out with respect to positioning class nodes and routing edges. The Ameos modelling tool used to create the OMOS models and diagrams does not impose any layout constraints on users. Class diagram symbols can be freely positioned on an (with respect to its width and height) unlimited 2-dimentional canvas. A class diagram's canvas hence provides a free-form layout. For instance, the rectangle representing a UML class could be placed anywhere in a diagram and it could be of any size — of course, it would be at least large enough to show all its contained information like the class's attributes and operations —, or a poly-line depicting an association could form any sort of path — it can even cross or overlap with other symbols.

3.3.2.2.3. The analysed model versions: In order to be able to analyse how models and diagrams evolve over time, Bosch provided three consecutive versions of each model. The first model, "Adaptive Shifting Strategy System", had been in development for about one and a half year until the first analysed version was created. The second model, "Power Supply Input/Output and Gearbox Operation Modes", had been developed for about a year until the first analysed version was created.

The following versions of the ASIS model were analysed:

Analysed model version	Date
1	End of December 2005
2	Beginning of February 2006
3	End of March 2006

Table 3.1.: Overview of the analysed versions of the "Adaptive Shifting Strategy System" model.

Analysed model version	Date
1	End of February 2006
2	End of March 2006
3	End of April 2006

Table 3.2.: Overview of the analysed versions of the "Power Supply Input/Output and Gearbox Operation Modes" model.

As explained above, the models were developed for some time before the versions of the models analysed here were created. It follows that "analysed version 1" does not refer to the very first version of the model, but to the first model version actually analysed as part of this research.

As will be explained in Section 3.3.2.3.1, the possibility to analyse several evolutions of the same models and diagrams allows to better understand how modellers use diagram layouts.

3.3.2.2.4. Summary: In this section, the elements on which the study focuses are defined. Based on the study's questions and propositions (see Section 3.3.2.1), the main elements the study focuses on are OMOS models and their diagrams. All diagrams constituting each of the three versions of both OMOS models were analysed as part of this research to answer the study's questions and prove (or disprove) its propositions. The modellers who created them were available for answering questions about the diagrams and were involved in the evaluation of the findings of the diagram analysis.

Regarding the extent of control over the study's environment, the author of this research had none at all since the OMOS models were created before they were analysed as part of this research, they were created before it was even known to the modellers that there will be a diagram analysis.

3.3.2.3. Map of evidence: the process of and the criteria for interpreting the findings and the logic for linking them to the propositions

The examples on semantic information conveyed by the layout of diagrams provided by the OMOS modellers (see Section 3.1), the definition of secondary notation phenomena, and the findings on the mental map taken from research literature (see Section 3.2) served as a starting point for defining criteria for conducting the study's model and diagram analysis and interpreting its findings.

Besides grouping symbols of semantically related classes in a diagram as pointed out by the OMOS modellers, the following diagram layout features were identified in the research literature domain-specific diagramming (see Section 3.2.2) and on secondary notation (see Section 3.2.3) as playing an important role when modellers apply their implicit and intimate knowledge about the semantics of the application domain when laying out diagrams (Batin et al. [BFN85]):

- Semantic clustering and ordering (as identified by Ware et al. [WHF93], Purchase et al. [PMCC01] and Eichelberger [Eic02a])
- Position, grouping, and proximity (as stated by Larkin & Simon [LS87])

In order the answer the study's questions and prove (or disprove) its propositions, an important part of the diagram analysis conducted as part of the study is to detect diagram layout phenomena described as secondary notation, especially semantic grouping, clustering and ordering as described in the research literature and by the OMOS modellers. In order to be able to determine whether a collection of diagram symbols actually represents a semantic group and/or is semantically ordered, all available textual documentation of the model elements (packages, classes and, where available, associations) depicted in diagrams was analysed. The meaning of model elements was understandable for the author because model elements were documented directly in the respective model by means of free-form text. Based on the documentation, a list of (according to the documentation) semantically related elements was compiled.

In addition to detecting semantic grouping and ordering in diagrams, deviations from the standard UML layout guidelines (as discussed in Section 3.2.2) were identified in the analysed OMOS diagrams, i. e., layout constellations which differ from the layout suggested by UML. Such deviations could be an indicator for the application of secondary notation because the modellers might have consciously chosen not to use UML's suggested layout in order to convey domain-specific meaning through the domain-specific layout. UML class diagrams can be regarded as a free-form "canvases" on which diagram symbols (for instance, rectangles representing classes and lines depicting associations and generalisations) can be laid out freely [OMG10a]. Modellers can therefore use this freedom to create layouts which do not always follow the layout style proposed by the UML standard, but which better depict their understanding of certain model elements depicted in a diagram. This finding is further supported by research on secondary notation phenomena discussed in Section 3.2.3.

As explained in Section 3.3.2.2, Bosch provided three consecutive versions of two real-world OMOS models (and all their diagrams). All diagrams of the three versions of both OMOS models were manually analysed as part of the pilot study. The diagram analysis was done as follows: First, (using the same OMOS modelling tool as used by Bosch to create the models in the first place) all diagrams of all three versions of both analysed OMOS models were printed on paper (this allowed for a more convenient visual comparison of different versions of a diagram than comparing them on-screen). Each diagram was then manually analysed in a visual way, i. e., by "looking at it", to identify diagram symbols in close proximity as well as deviations from the suggested UML layout guidelines.

Diagram symbols in close proximity were compared with the semantically related elements identified from the documentation. If the model elements depicted by the closely positioned diagram symbols were indeed semantically related, the diagram symbols then became a candidate for a semantic group. Also, the OMOS diagrams were manually searched for layout constellations that were uncommon in the sense that following UML's notation guide would lead to layouts that differ from the actual diagrams' layouts. These uncommon diagram symbol constellations then became secondary notation candidates.

3.3.2.3.1. Evolving diagrams help to identify secondary notation phenomena and to reason about the mental map: Since Bosch provided three consecutive versions of each model, three different versions of each diagram were available for the diagram analysis conducted as part of the pilot study.

The possibility to analyse different versions of the same diagrams allows gaining insights into the evolution of diagrams. For instance, by analysing three consecutive versions of a diagram, it becomes possible to see how the diagram evolved over the period of several weeks or months. The evolution of diagrams provides valuable information about where, in a diagram, symbols are added, deleted, or modified. Insights drawn from evolving diagrams are thus especially valuable for learning about how OMOS modellers create and modify diagrams. These insights can thus provide strong and compelling indications regarding secondary notation phenomena which become more apparent due to being able to see how a diagram evolves over time. For instance, as will be shown in Section 3.5, for symbols added or relocated in later versions of a diagram it can be determined in which visual context they are positioned, for instance, whether they are positioned in close visual proximity to symbols depicting semantically related model elements, i. e., it could be determined whether the added symbol's position (visually) conveys semantic meaning. The comparison of different versions of a diagram also helps with respect to identifying deviations from the standard UML layout guidelines: Symbols added or relocated in later versions of a diagram can be compared with (in terms the UML model element type) similar symbols already belonging to a diagram. If added or relocated symbols are positioned in different ways than the existing ones (even if the done in accordance with UML's layout guidelines) or than advertised by the UML layout guidelines, there is a possibility that the modellers do so in order to consciously convey additional semantic meaning through a diagram's layout.

3.3.2.3.2. Summary: As will be explained in the next section, all identified candidate semantic groups (which were identified during the diagram analysis) were discussed with OMOS modellers to verify whether the grouped diagram symbols do indeed depict groups of semantically related model elements. Furthermore, secondary notation candidates resulting from diagram symbol constellations which did not follow UML's diagram layout guidelines were discussed with the OMOS modellers.

With respect to pilot study research (see Section 3.3.1), the study presented here is a contemporary one — as opposed to a historic study which deals with events from the "dead past." It is a contemporary study because the OMOS modellers who created the analysed diagrams were still available and took part in the investigation by reviewing and providing feedback on the study's findings.

3.3.3. Validity and threats to validity

As discussed in Section 3.3.1, the validity of a pilot study has to be taken into account. In order to ensure its internal validity, which refers *"specifically to whether there is sufficient evidence to support [a] claim [i. e., proposition]"* [Yin03], multiple sources of evidence should be used and a chain of evidence should be established during the study's data collection phase. In addition, experts in the field should review the study's findings. In order to ensure external validity — which deals with *"knowing whether a study's findings are generalizable beyond the immediate study"* [Yin03] —, it should be tried to generalize the study's findings to the theory described by the respective research literature.

3.3.3.1. Ensuring the study's internal and external validity

The following approach is taken to ensure the study's internal validity with respect to collecting sufficient evidence to support the study's findings: In order to validate the findings of the pilot study, noticeable diagrams identified (by the author of this research) during the diagram analysis were discussed with the Bosch engineers who actually created them. This discussion took place in a two-day meeting in September 2006 and via phone.

The following findings help to ensure the study's internal validity with respect to using multiple sources of evidence and establishing a chain of evidence during the study's data collection phase: All the semantic grouping and secondary notation candidates identified during the OMOS diagram analysis (as defined in Section 3.3.2.3) were discussed with the Bosch engineers who created the analysed OMOS models and diagrams. The respective diagrams (whose symbols were believed to express semantic grouping or other secondary notation phenomena) were presented to the modellers (in the meeting) and it was discussed whether the layout constellations were important for the modellers and if they did actually convey semantic meaning. The engineers then verified whether a diagram's layout indeed conveyed additional semantic meaning and, if so, explained what kind of meaning is intended to be conveyed.

A selection of the slides used to discuss diagram layouts with the OMOS modellers during the meeting is provided in Appendix B. The slides were used to present the findings of this pilot study to the modellers and to validate these findings by discussing them with the modellers who created the analysed diagrams.

To summarize, the following measures were taken to ensure the study's internal validity:

- Two different OMOS models were analysed. All (100 per cent of the) diagrams of all three versions of both OMOS models were analysed as part of the pilot study. Hence, all available data (as provided by Bosch) was taken into account.
- Experts in the field (i. e., the OMOS modellers) were involved when the study's findings were analysed.
- As will be explained in more detail when the findings of the case are discussed in Section 3.5, diagrams dealing with similar domain concepts had similar layouts. These diagrams contained also similar secondary notation phenomena. Hence, a chain of evidence can be established because if only a few diagrams contained secondary notation phenomena, these phenomena could be *visual noise* rather than semantic information conveyed through the layout of the diagrams or these phenomena might not be representative for the analysed models. However, when diagrams representing different, but related concepts expose similar layouts, this is a strong indicator that the diagrams do indeed convey semantic information through the layout modellers had deliberately chosen for several diagrams. This finding is further supported by the fact that new diagrams could not simply be copies of existing diagrams since the diagrams depicted different (but related) classes. Hence, a chain of validity can be established based on the layout similarity between several diagrams.

- The analysed OMOS models and diagrams were created *before* the decision to conduct this study was taken. At the time the modellers created the diagrams they were unaware that their diagrams might become subject of an analysis on meaning-conveying layouts.

The following findings help to ensure the study's external validity by generalising the study's findings:

- Another software development group at Bosch — which used the OMOS approach to develop embedded software for car air bag controllers — used similar secondary notation phenomena (i. e., similar diagram layout patterns) in the layouts of their diagrams to express domain-specific meaning. The author of this research got to know about this group's diagram layouts when he discussed the topic of OMOS models and meaning-conveying diagram layout with one of the group's engineers (during a coffee break — the air bag controller software group's offices were next to the automated gearbox controller software team's). The author had access to some of the OMOS diagrams. The diagram layout patterns identified (by this research) for the "Adaptive Shifting Strategy System" and "Power Supply Input/Output and Gearbox Operation Modes" models are, therefore, not unique. The analysed project is hence a typical representative regarding visual modelling based on the OMOS approach.
- The author worked on a space systems engineering project for EADS/Astrium and the European Space Agency. In this project [EMdK09], UML class diagrams were used to define meta-models for model-based space systems engineering. These diagrams shows similar layout pattern as identified in OMOS diagrams by this pilot study, additional semantic meaning is expressed through the layout of diagrams.
- The research literature confirms that layouts of diagrams are used to convey inherent semantic information. The research literature on secondary notation phenomena in diagram layouts and on the mental map of diagrams discussed in Section 3.2 acknowledges that grouping, ordering and, proximity of diagram symbols is important when users apply their implicit and intimate knowledge about the semantics of the application domain when laying out diagrams (see, Larkin & Simon [LS87], Batin et al. [BFN85], Ware et al. [WHF93] and Eichelberger [Eic03]).
- It is argued that the findings of the pilot study are generalisable to modelling contexts...

– in which models are created by means of diagrams which are created manually, and
– in which models are constituted from of a large number of diagrams, and
– which, instead of applying domain-specific visual languages, use general purpose diagram notations like UML which do not allow formally defining domain-specific meaning in diagrams.

While acknowledging the limitations of pilot studies, the above findings are considered to ensure the study's internal and external validity.

3.3.3.2. Threats to validity

Care has to be taken in order to not mistake diagrammatic noise for layout constellations conveying semantic meaning through layout of diagrams. A possible threat to the study's validity thus is that, during the diagram analysis, secondary notation phenomena are wrongly identified to convey meaning, which, however, are rather diagrammatic noise not actually conveying any additional semantic meaning. In order to avoid such false positives, i. e., to decide whether the layout is consciously chosen and indeed conveys meaning, OMOS modellers involved with creating or modifying the respective diagrams reviewed the findings (as explained in Section 3.3.3.1).

In addition, the respective research literature on secondary notation phenomena in diagram layouts and the mental map of diagrams was reviewed (see Section 3.2) before the study was conducted in order to ensure that the diagram analysis took into account the findings on additional meaning conveyed through the layout of diagrams identified by other research.

As discussed in Section 3.1, OMOS modellers had also explained some of the secondary notation phenomena they applied in their diagrams to convey additional semantic meaning before the study was conducted. The phenomena described by the OMOS modellers were, therefore, taken into account during the diagram analysis.

However, when asked about the candidates of secondary notation phenomena (as identified during the diagram analysis, see Section 3.3.2.3), OMOS modellers might falsely state that they do indeed convey semantic information while in reality they do not. Since the findings of the diagram analysis were discussed in a meeting with several modellers, the threat that modellers accept the author's false findings can be considered a minor threat since more than one modeller is usually familiar with a certain diagram.

3.4. Findings regarding modelling by means of diagrams

Before the pilot study's findings regarding meaning-conveying diagram layout are discussed in detail in Section 3.5, the findings regarding the analysed OMOS models in general are discussed in this section. This section, therefore, focuses on the pilot study's first research question defined in Section 3.3.2.1: How are diagrams used to visually construct OMOS models?

The two analysed models, "Adaptive Shifting Strategy System" and "Power Supply Input/Output and Gearbox Operation Modes", consists of several dozens to more than 130 class diagrams. Table 3.3 and Table 3.4 show the number of classes and diagrams for each version of both analysed models.

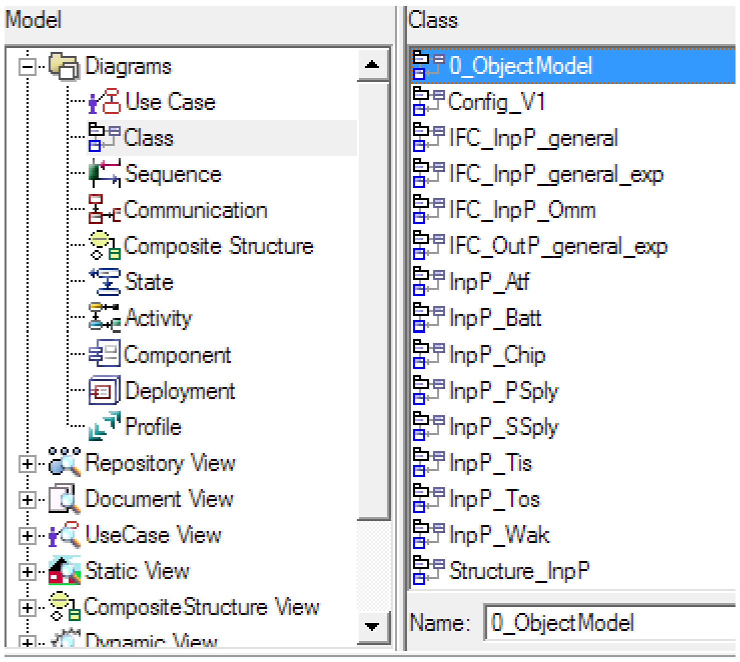
Analysed model version	№ classes	№ class diagrams
1	272	138
2	272	137
3	272	137

Table 3.3.: Overview of the numbers of classes and class diagrams of the analysed "Adaptive Shifting Strategy System" model versions.

Analysed model version	№ classes	№ class diagrams
1	186	43
2	223	47
3	231	49

Table 3.4.: Overview of the numbers of classes and class diagrams of the analysed "Power Supply Input/Output and Gearbox Operation Modes" model versions.

Each diagram represents a different part or aspect of the underlying OMOS model. On the structural level, the classes comprising a OMOS model (and which represent the concepts realised by the software system defined by the model) are organised in a hierarchical manner. In order to access the diagrams, the UML modelling tool presents the user with a list of the names of all class diagrams (see Fig. 3.2). This list provides the entry point into a model. A model's class diagram listing, shown in Fig. 3.2, reflects this hierarchical organisation of the model.


Model

- Diagrams
  - Use Case
  - Class
  - Sequence
  - Communication
  - Composite Structure
  - State
  - Activity
  - Component
  - Deployment
  - Profile
- Repository View
- Document View
- UseCase View
- Static View
- CompositeStructure View
- Dynamic View

Class

- 0_ObjectModel
- Config_V1
- IFC_InpP_general
- IFC_InpP_general_exp
- IFC_InpP_Omm
- IFC_OutP_general_exp
- InpP_Atf
- InpP_Batt
- InpP_Chip
- InpP_PSply
- InpP_SSply
- InpP_Tis
- InpP_Tos
- InpP_Wak
- Structure_InpP

Name: 0_ObjectModel

Figure 3.2.: List of UML class diagrams from the Ameos modelling tool.

The name of each diagram reflects its purpose. It reflects the part of the software system's domain which is represented by the symbols in the respective diagram. The names of the diagrams show that the model is decomposed into smaller components. For each component in the system there is a diagram named like the component. For instance, a diagram called "ESP" (short for Electronic Stability Program) contains classes dealing with the car's electronic stability control. OMOS diagrams contain classes from different packages. In the following, the features of the model and its constituting diagrams are explained in detail.

Both models also have a OMOS diagram called "Object model" which represents the system's top-most classes. For the "Adaptive Shifting Strategy System" model, the "Object model" root diagram contains the root class of the model, called "CL_FAPR" (which stands for "Fahrprogramm", which is German for "driving program"). This class defines composition associations to the top-level domain classes contained in the model, i. e., those classes are defined in this diagram as part classes of the root class. For instance, some of the top-level concepts define driver type detection, power train, vehicle status, electronic stability program (ESP), and drive situation detection. These classes are top-level classes, i. e., they do not have super classes. The root diagram does not contain any other classes than the root class and the top-level classes. For these classes, no details like attributes, operations, or associations are shown in this diagram. These details are defined in other diagrams. For each of these top-level classes, a diagram with the same name exists. Such diagrams contain the respective top-level class again, but this time attributes, operations, sub- and part classes, and associations to other classes are defined.

For instance, the “FZGG” diagram (which stands for “Fahrzeuggrößen” or “vehicle status values” in English) defines these details for two classes (see Fig. 3.3), “CL_-FZGG” and “CL_FZGG_KOMP.” Class “CL_FZGG_KOMP” is a sub-class or variant of class “CL_FZGG.” In the diagram, one attribute is defined for class “CL_FZGG” and 13 attributes are defined for class “CL_FZGG_KOMP”, for instace, “slope”, plus 18 operations, for instance, “getSlope” and “getState.” For class “CL_FZGG”, four associations to associated classes are defined. For instance, class “CL_FZGG” associates class “CL_GETR” (“Getriebe”, which is German for gearbox) and “CL_MOT” (for “Motor”, or engine in English). Class “CL_FZGG_-KOMP” associates two other classes, for instance, “CL_FZFWB” (“Fahrtwiderstandsberechnung”, German for tractive resistance calculation). All associations are directed, i. e., they point from class “CL_FZGG” and “CL_FZGG_KOMP” to the associated classes. No details (such as attributes, operations, or associations) of the associated classes are shown in this diagram.

The diagrams discussed so far serve two purposes: (1) define classes and their details and (2) define the inheritance hierarchy, i. e., define OMOS variants (see Section 2.1.3). This approach is applied for a number of *main classes* in a certain diagram. In case main classes associate classes other than the diagram’s main classes, the associated classes are also part of this diagram together with associations pointing from the main classes to these associated classes. Except for public operations, which are sometimes displayed, no further details are shown for associated classes — those details are defined in other diagrams. A certain main class is always defined in one particular diagram, but there can be several diagrams in which this class is displayed as an associated class.

If a main class has further part-classes, i. e., if the domain concept represented by the class is further divided into sub-concepts, the diagram that shows the class as a main class also contains its part-classes. Depending on the complexity of the sub-classes and part-classes their details — attributes, operations, sub-classes, part-classes, and associated classes — are either defined in the same diagram or in new diagrams dedicated in particular to the details of the part-classes. If the part-classes

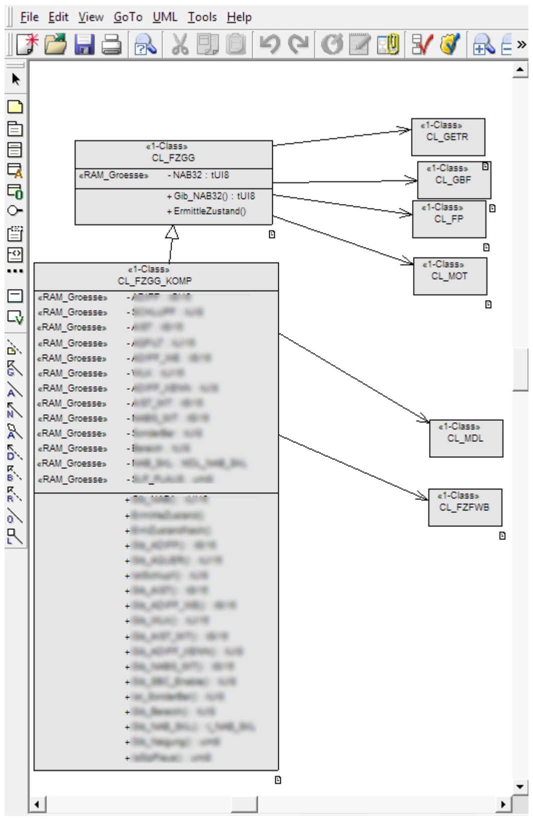


Figure 3.3.: OMOS diagram “FZGG” shown in the Ameos class diagram editor.

have themselves sub-classes and/or part-classes or relationships to many associated classes, new diagrams are created for the part-classes. Then, the details of the part-classes are not defined in the same diagram; they are defined in other diagrams which are named like the part-classes.

For example, class “CL_FAPR” (which means “Fahrprogramm” and is German for driving program) defined in diagram “Fahrprogramm” (driving program) has a part-class called “CL_KRITV” (which means “Kriterienverwalter”, [gear change] criteria manager in English). Diagram “Fahrprogramm” does not define any details of this class. This is done in another diagram called “Kriterienverwaltung” (German for [gear change] criteria management) where attributes, operations, sub-classes, and associated classes of class “CL_KRITV” are defined.

Another example for the creation of new diagrams is class “CL_FZGG_KOMP.” As explained above, this class is defined in diagram “FZGG.” However, another diagram, called “Fahrzeuggroessen” (meaning “vehicle status values” in English), exists which defines the seven part-classes of class “CL_FZGG_KOMP.” Three of those part-classes are “CL_FZRAD”, “CL_FZAQ”, and “CL_FZSLP” which represent classes for accessing a vehicle’s wheel, lateral acceleration, and slip state respectively. The details of these classes are not defined in diagram “Fahrzeuggroessen.” For each of the seven part-classes another diagram exists in which the details of each class are defined. Even though the seven part-classes have no part-classes themselves — because they represented the states of vehicle components which were not further dividable — each part-class has one to three sub-classes (i. e., variants) which in turn have one or two sub-classes.

3.4.1. OMOS diagram Tiptronic

An example demonstrating the OMOS modellers’ approach to partition models by means of diagrams is shown in Fig. 3.4. This figure shows the “Tiptronic” diagram from the “Adaptive Shifting Strategy System” model. This diagram defines classes which deal with information from the car’s *tiptronic*[5] for deciding and calculating when and how to shift gears. Tiptronic allows the driver (of a car with automatic transmission) to manually control the gear selection. The main classes defined in this diagram are

- Class “CL_UKTIP”, which represents a whole-class; and
- Classes “CL_TIPIM”, “CL_TIPZA”, and “CL_TIPAN”, which represent part-classes of class “CL_UKTIP”; and
- Classes “CL_UKTIP_ASIS”, “CL_UKTIP_ADVW”, “CL_UKTIP_PORS”, and “CL_UKTIP_PAG”, which represent sub-classes of class “CL_UKTIP.”

The other three classes, “CL_GETR”, “CL_GBF”, and “CL_STAT”, in diagram “Tiptronic” are associated classes.

3.4.2. Adding new diagrams to a OMOS model

New diagrams are created when new functionality, represented by one or more new classes, is added to a model. Then, part-classes belonging to different classes that are defined in different diagrams are grouped into a single new diagram to again define parts-classes of these part-classes. The classes are grouped together into this

[5]See *http://en.wikipedia.org/wiki/Manumatic* for an explanation of tiptronic (access date: 4/12/2012).

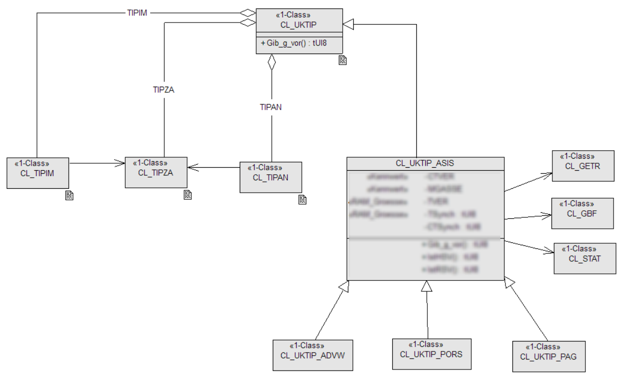


Figure 3.4.: OMOS diagram "Tiptronic".

diagram to realise a certain problem in terms of the functionality that a model is expected to provide. Part-classes of each of these (in terms of the diagram) top-level classes of the whole-part hierarchy represent further sub-components of the functionality.

Even if the functionality realised by a new class conceptually belonged to an existing diagrams, adding new classes will result in new diagrams if an existing diagram becomes too complex in terms of the layout of its whole-part and/or inheritance hierarchies. Usually the elements of a diagrams fit on a single screen. If they do not fit on a screen, they are, if possible, arranged in a way which requires scrolling in only one direction (usually this is the vertical axis).

3.4.3. Summary

This section's findings can be summarised as follows: Given a OMOS diagram D, the classes presented in this diagram play different roles. The classes can act as whole- or part-classes and/or super/sub-classes or associated classes.

1. Whole- and super classes are those classes for which no super or whole-class is depicted in diagram D.
2. Sub- or part-classes are those classes which are sub- or part-classes of a super- or whole-class or another sub- or part-class in diagram D.

As the names of these class types suggest, the class symbols in diagram D are connected by inheritance or composition relationship symbols. Like the classes themselves, their relationships are defined in diagram D.

For the purpose of the pilot study discussed here, these two types of classes are called a diagram's *main classes.* Then*, associated classes* are those classes which main classes connect to only by means of associations. Associations are usually, but not always, directed from the main to the associated class.

For associated classes, none of their attributes or associations, inheritance, and whole-part relationships are displayed in diagram *D*. This is different for an associated class's operations: the modelling tool allows the user to select the attributes and operations that are shown for a class. Hence, the user can show or hide some or all of a class's attributes and operations. Sometimes some or all of an associated class's *public* operations are shown (because the object-oriented approach followed by OMOS allows only publicly visible operations to be called by (instances of) classes associating (instances of) other classes). Visualising the public operations makes the behaviour/functionality provided by the associated class and used by the associating main class explicitly visible in the diagram. If only some of the public operations are shown, it is indicated that the associating main class does only use these particular operations.

The details of classes acting as associated classes in one digram are of course defined in other diagrams for which these associated classes act as main classes.

There is exactly one diagram for which a certain class acts as a main class. However, there can be many diagrams for which this class acts as an associated class. To support navigation between diagrams, the modelling tool provides a list of diagrams in which a certain model element (for instance, a class represented in a displayed diagram) is shown as well. The user can then open these diagrams too, and the diagram's representation of the model element from which the selection started gets selected in these diagrams.

A OMOS diagram displays a section of the whole-part (i. e., containment) class hierarchy. Classes of two or three consecutive levels of the whole-part hierarchy are shown in a single diagram. If part-classes belonging to the lowest class containment level shown in this diagram have again part-classes, they are defined in another diagram. Then, the whole-class whose parts are defined is also shown in this new diagram. Thus, a part-class shown on the lowest level of a diagram is included in another diagram used to define its part-classes. So, a part-class on the lowest level of the section of the whole-part hierarchy defined in a diagram can result in a (in terms of class containment) next-lower-level diagram on the next lower level in the diagram hierarchy. Since there can be more than one part-class on this lowest level, several other diagrams may be use to define the details of these classes depending on the complexity of these part-classes.

However, not every part-class will result in a sub-diagram defining this class's details (i. e., part-classes, sub-classes, and associated classes). Diagrams include more than one class representing a whole-class on the topmost level of the whole-part hierarchy shown in this diagram. The documentation of these classes (shown at the topmost level in the same diagram) indicates that they are related in terms of the functionality they represent. Furthermore, the name of a diagram is related to a certain part/functionality of the domain defined in the respective model.

3.5. Findings regarding conveying additional semantic information in diagrams through their layout

While the analysis findings regarding the analysed OMOS models in general are discussed in the previous section, the findings of the diagram layout analysis conducted as part of this research are discussed in detail in this section. In this section, actual diagrams are discussed which have been identified (by Bosch engineers who created and worked with them) to convey additional semantic information through their layout. The objective of this discussion is to investigate if and how modellers use layout to convey domain-specific meaning which is informal and not part of underlying model. This section therefore focuses on the pilot study's first proposition defined in Section 3.3.2.1.

3.5.1. OMOS diagrams OutP_IFC_General and OutP_IFC_General_Exp as an example for semantic grouping

Diagram "OutP_IFC_General" demonstrates an important layout finding regarding the presence of secondary notation in OMOS diagrams: *semantic grouping* — symbols of semantically related classes are grouped in close visual proximity.

In version three of diagram "OutP_IFC_General" (see Fig. 3.6) and "OutP_IFC-_General_Exp" (see Fig. 3.8), classes "CL_OutPIfcHss", "CL_OutPIfcHss_Hss", and "CL_OutPHssCtl" were added to the "Power Supply Input/Output and Gearbox Operation Modes" model. These classes realise the high-level functionalities of the high-side sensors (HSS) control. The (lower-level) sub-components of the

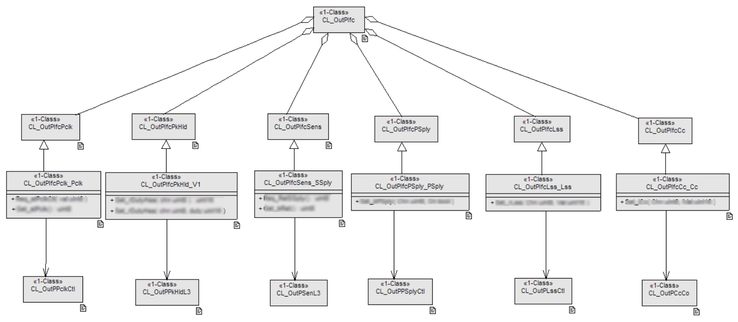

«1-Class» CL_OutPIfc
«1-Class» CL_OutPIfcPclk
«1-Class» CL_OutPIfcPkHld
«1-Class» CL_OutPIfcSens
«1-Class» CL_OutPIfcPSply
«1-Class» CL_OutPIfcLss
«1-Class» CL_OutPIfcCc
«1-Class» CL_OutPIfcPclk_Pclk
«1-Class» CL_OutPIfcPkHld_V1
«1-Class» CL_OutPIfcSens_SSply
«1-Class» CL_OutPIfcPSply_PSply
«1-Class» CL_OutPIfcLss_Lss
«1-Class» CL_OutPIfcCc_Cc
«1-Class» CL_OutPPclkCtl
«1-Class» CL_OutPPkHldL3
«1-Class» CL_OutPSenL3
«1-Class» CL_OutPPSplyCtl
«1-Class» CL_OutPLssCtl
«1-Class» CL_OutPCcCo

Figure 3.5.: OMOS diagram “OutP_IFC_General” (version 1 and 2).

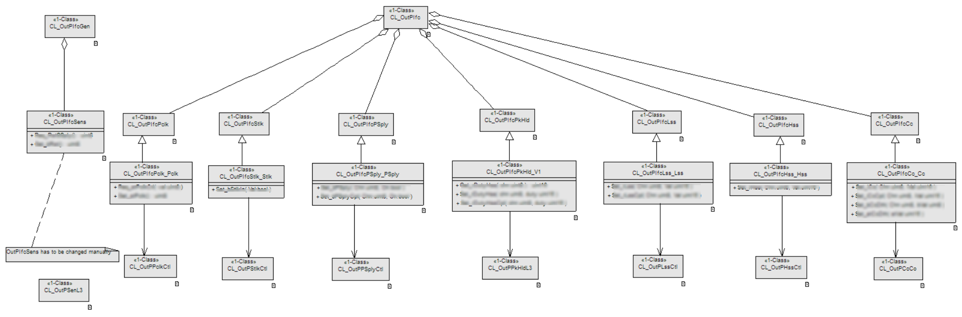


Figure 3.6.: OMOS diagram “OutP_IFC_General” (version 3).

high-side sensors control (software function) are defined in another OMOS diagram, “OutP_Hss” (see Section 3.5.3), which also has been added in analysed model version three.

An interesting aspect of both diagrams’ layout is that the newly added classes (“CL_OutPIfcHss”, “CL_OutPIfcHss”, and “CL_OutPHssCtl”) are placed next to the classes realising the low-side sensor control (LSS). It would have been easier for the modellers to simply add the new classes to the far left or right of the diagram (see version one and two of the diagrams in Fig. 3.5 and Fig. 3.7 respectively) where no rearrangements of existing diagram symbols would have been required. However, the modellers decided to place the new classes for the high-side sensor control in visual proximity of the low-side sensor control classes — taking into account that existing diagram symbols had to be rearranged.

When the diagrams were discussed with the OMOS modellers who created them, they explained that the extra effort of rearranging existing diagram symbols was made in order to add the new class symbols for the high-side sensor control next to the low-side sensor control classes. Positioning low- and high-side high-side sensor control classes next to each other is important for the modellers because these classes are semantically closely related. The modellers hence visually convey this semantic relationship in the diagrams by applying the secondary notation phenomenon known as semantic grouping.

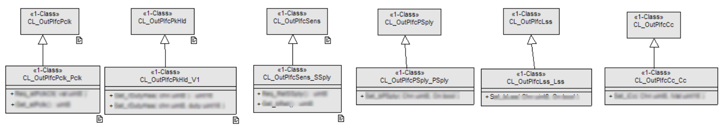

«1-Class» CL_OutPIfcPclk
«1-Class» CL_OutPIfcPkHld
«1-Class» CL_OutPIfcSens
«1-Class» CL_OutPIfcPSply
«1-Class» CL_OutPIfcLss
«1-Class» CL_OutPIfcCc
«1-Class» CL_OutPIfcPclk_Pclk
«1-Class» CL_OutPIfcPkHld_V1
«1-Class» CL_OutPIfcSens_SSply
«1-Class» CL_OutPIfcPSply_PSply
«1-Class» CL_OutPIfcLss_Lss
«1-Class» CL_OutPIfcCc_Cc

Figure 3.7.: OMOS diagram “OutP_IFC_General_Exp” (version 1 and 2).

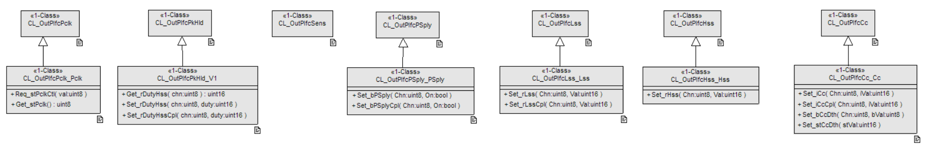

«1-Class» CL_OutPIfcPclk
«1-Class» CL_OutPIfcPkHld
«1-Class» CL_OutPIfcSens
«1-Class» CL_OutPIfcPSply
«1-Class» CL_OutPIfcLss
«1-Class» CL_OutPIfcHss
«1-Class» CL_OutPIfcCc
«1-Class» CL_OutPIfcPclk_Pclk
+Req_stPclkCtl(val:uint8)
+Get_stPclk() : uint8
«1-Class» CL_OutPIfcPkHld_V1
+Get_rDutyHss(chn:uint8) : uint16
+Set_rDutyHss(chn:uint8, duty:uint16)
+Set_rDutyHssCpl(chn:uint8, duty:uint16)
«1-Class» CL_OutPIfcPSply_PSply
+Set_bPSply(Chn:uint8, On:bool)
+Set_bPSplyCpl(Chn:uint8, On:bool)
«1-Class» CL_OutPIfcLss_Lss
+Set_rLss(Chn:uint8, Val:uint16)
+Set_rLssCpl(Chn:uint8, Val:uint16)
«1-Class» CL_OutPIfcHss_Hss
+Set_rHss(Chn:uint8, Val:uint16)
«1-Class» CL_OutPIfcCc_Cc
+Set_iCc(Chn:uint8, iVal:uint16)
+Set_iCcCpl(Chn:uint8, iVal:uint16)
+Set_bCcDth(Chn:uint8, bVal:uint8)
+Set_stCcDth(stVal:uint16)

Figure 3.8.: OMOS diagram “OutP_IFC_General_Exp” (version 3).

The finding of semantic grouping as an important means to convey semantic information through the layout of OMOS diagrams is further supported by the fact that classes semantically related to the concepts shown in diagram “OutP_IFC_-General_Exp” (discussed in this section) are placed in the identical order (as in this diagram) in diagram “OutP_ObjectModel” (see Section C.4). When the high-side control classes were added in version three, the “OutP_ObjectModel” diagram was modified in a similar way as described above: Instead of simply positioning the new classes at the left or right side of the diagrams, they were added next to the low-side sensor controller classes — this, again, required rearranging other diagram elements. This layout was consciously chosen because the modellers want to emphasise that the new classes (added to diagram “OutP_ObjectModel”) are semantically related to the existing low-side sensor controller classes and are, therefore, positioned in close proximity to them.

3.5.2. OMOS diagram OutPLss as an example for semantic ordering

The classes defined in diagram “OutPLss” deal with the inner components of the low-side sensor (LSS) control. The main class, “CL_OutPLssCtl”, is defined as a part-class of class “CL_OutP” in OMOS diagram “OutP_ObjectModel” (discussed in Section C.4).

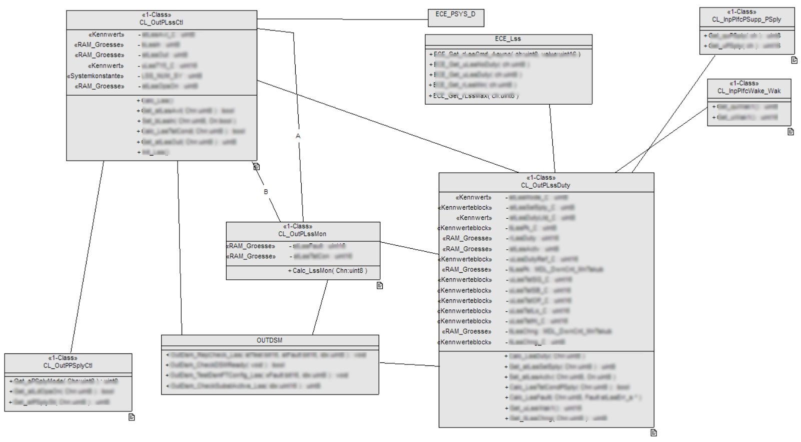


Figure 3.9.: OMOS diagram “OutPLss” (version 1).

In version 2, class “CL_OutPLssCtl_Mn” was added to the diagram (and to the model). It is placed at the same horizontal position as its base class “CL_OutPLss-Ctl” and its part-class “CL_OutPLssDuty.” This way of positioning class “CL_-OutPLssCtl_Mn” is in contrast with the UML class diagram layout suggestions (see Section 3.2.2) and also in contrast to the layout patterns applied in other OMOS diagrams belonging to the same OMOS model, for instance, diagram “OutP_IFC_-General_Exp” and “OutP_IFC_General” discussed in Section 3.5.1. If this diagram would follow the layout approach applied in these diagrams, class “CL_OutPLss-Ctl” would be placed in the top centre of the diagram and sub-class “CL_OutP-LssCtl_Mn” would be positioned directly below. Part-classes “CL_OutPLssMon” and “CL_OutPLssDuty” would be positioned side-by-side below their whole-class “CL_OutPLssCtl_Mn.”

The OMOS modellers who created this diagram explained that the reason for the deviation from the UML layout suggestions is that the software functionalities realised by these classes belong to different software (and hardware) layers in the software system. As can be seen in version 3 of diagram “OutPLss” (Fig. 3.11), the classes depicted in this diagram belong to different packages, some classes belong to package “OutPL3” (which represents software layer 3) and some classes belong to package “OutPL2” (representing “layer 2”)[6].

The OMOS modellers further explained that because class “CL_OutPLssMon” belongs to the same domain layer as its whole-class “CL_OutPLssCtl_Mn”, it is

[6]Each class’s fully-qualified name including the package it belongs to is shown in the diagram below the class’s name to demonstrate that the classes indeed belong to different packages.

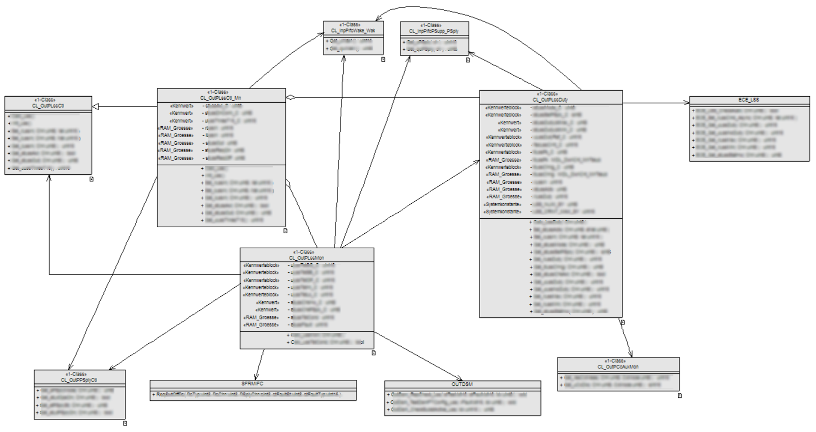


Figure 3.10.: OMOS diagram “OutPLss” (version 2).

positioned below its whole-class, as suggested by the UML layout guidelines for part-classes. However, class “CL_OutPLssDuty”, the other part-class of whole-class “CL_OutPLssCtl_Mn”, belongs to a different domain layer, it is, thus, positioned on the right side of the whole-class — not below it as UML’s layout guidelines suggest.

The diagram layout approach which positions classes according to the domain layers they belong to is used for all three versions of the diagram (shown in Fig. 3.9, Fig. 3.11 and Fig. 3.11). The diagram hence demonstrates an important layout finding regarding the presence of secondary notation in OMOS diagrams: *semantic ordering* — class symbols are ordered from left to right and top to bottom according to the order of the domain layers their respective classes belong to.

Other OMOS diagrams exist whose layouts are similar with respect to classes being positioned according to their domain layers, for instance, diagram “OutP_Hss” discussed in Section 3.5.3, diagram “InpP_Chip” discussed in Section C.2, diagram “InpP_PSply” discussed in Section C.3, and diagram “OutPStaLck” discussed in Section C.8. In those diagrams, classes are defined for which the name of the layer is part of the class name, for instance, classes “CL_OutPRstL2”, “CL_OutPSenL3” and “CL_OutPRstL3” belonged to software layer “L2” and “L3” of the software system, respectively. As explained above, this layering is also visualised in the diagrams.

The layout similarities of all these diagrams further support the finding of semantic ordering as an important means to convey semantic information through the layout of OMOS diagrams.

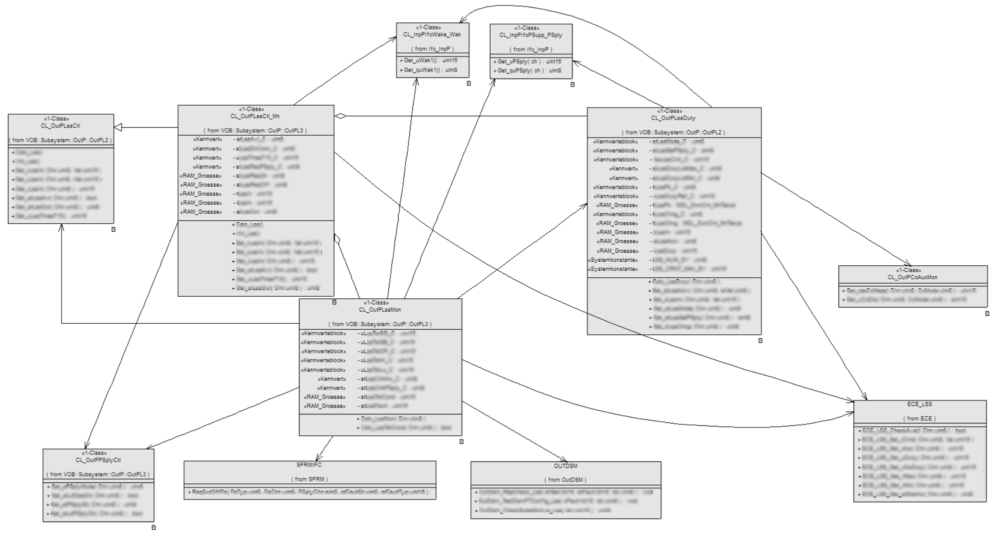


Figure 3.11.: OMOS diagram “OutPLss” (version 3).

An example for diagram symbols for which ordering is not important (according to the creators of the diagram) is associated class “SFRMIFC.” Even though it was added (in version 2 of the diagram shown in Fig. 3.10) *between* associated class “CL_OutPPSplyCtl” and “OUTDSM” — not to the left or right —, the particular order of the three classes is not important with respect to their domain meaning, i. e., according to the OMOS modellers the class symbol’s horizontally ordered layout does not convey additional domain-specific information.

3.5.3. OMOS diagram OutP_Hss as an example for semantic ordering

Diagram “OutP_Hss” (shown in Fig. 3.12) was added in version 3 of the “Power Supply Input/Output and Gearbox Operation Modes” model. In this diagram, classes realising the high-side sensor (HSS) control are defined.

The diagram’s layout is similar to OMOS diagram “OutP_Lss” which defines classes dealing with the low-side sensor (LSS) control (diagram “OutP_Lss” is discussed in Section 3.5.2). Its layout is also similar to OMOS diagram “OutPStaLck” (discussed in Section C.8) which defines the software functionalities for controlling and monitoring the electronic starter and steering locking device.

Following layout patterns similar to those diagrams, the class symbols in diagram “OutP_HSS” are laid out in a left-right and top-down manner. Classes belonging

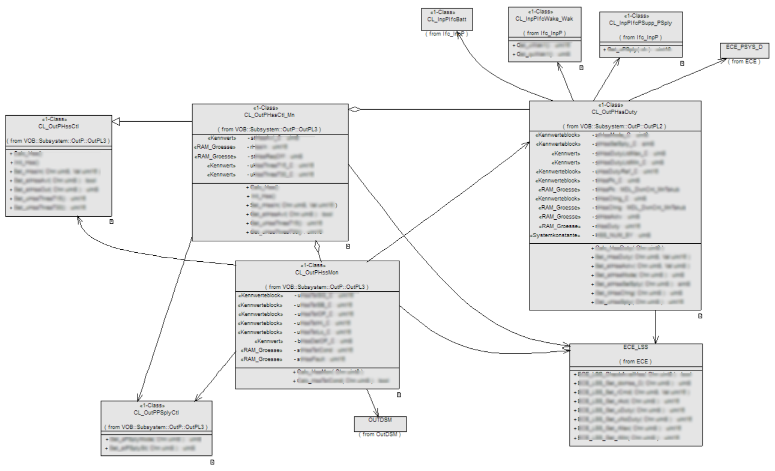

«1-Class» CL_InpPIfoBatt
(from Ifc_InpP)
«1-Class» CL_InpPIfoWake_Wak
(from Ifc_InpP)
«1-Class» CL_InpPIfoPSupp_PSply
(from Ifc_InpP)
ECE_PSYS_D
(from ECE)
«1-Class» CL_OutPHssCtl
(from VOB::Subsystem::OutP::OutPL3)
«1-Class» CL_OutPHssCtl_Mn
(from VOB::Subsystem::OutP::OutPL3)
«1-Class» CL_OutPHssDuty
(from VOB::Subsystem::OutP::OutPL2)
«1-Class» CL_OutPHssMon
(from VOB::Subsystem::OutP::OutPL3)
ECE_LSS
(from ECE)
«1-Class» CL_OutPPSplyCtl
(from VOB::Subsystem::OutP::OutPL3)
OUTDSM
(from OUTDSM)

Figure 3.12.: OMOS diagram "OutP_Hss" (version 3).

to layer 3 (see the package names displayed in Fig. 3.12) are positioned on the left of the diagram while class "CL_OutPHssDuty", which belongs to layer 2, is positioned further to the right. Class "CL_OutPHssMon", which is a part-class of class "CL_-OutPHssCtl_Mn", is positioned below its whole-class (i. e., top-down layout is used as suggested by the UML layout guidelines), and associated class "CL_OutPPSply-Ctl", which also belongs to layer 3, is positioned below the whole-part class hierarchy.

This diagram, too, demonstrates the importance of semantic ordering used by OMOS modellers to convey semantic information through the layout of diagrams (also discussed in Section 3.5.2 and Section 3.5.4).

3.5.4. OMOS diagram OutPCcCo as an example for semantic grouping and ordering

The findings of the analysis of diagram "OutPCcCo"'s layout properties are provided in this section. The classes defined in this diagram deal with controlling and managing the communication of the gearbox controller's electronic control unit and other control units. The three versions of this diagram are shown in Fig. 3.13, Fig. 3.14, and Fig. 3.15, respectively.

A general layout finding is that the sub-classes of class "CL_OutPCcCo", i. e. classes "CL_OutPCcCo07", "CL_OutPCcCo0708", and "CL_OutPCcCo08", and their part-classes "CL_OutPCc07Ctl", "CL_OutPCc0708", and "CL_OutPCc08Ctl", are horizontally ordered from left to right.

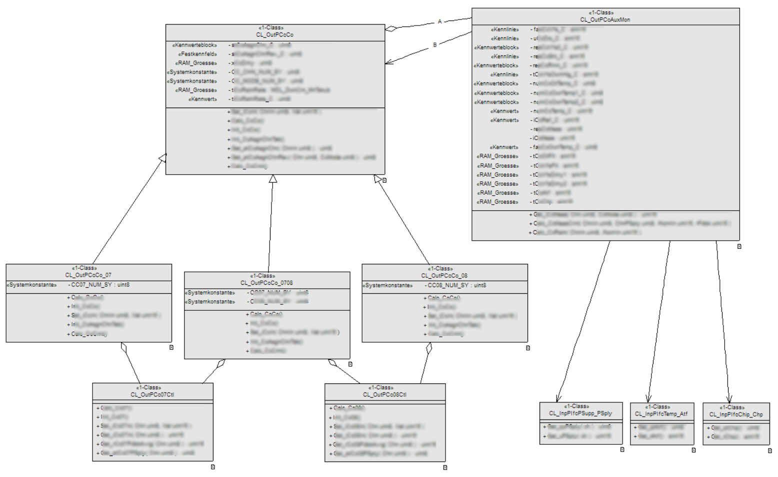


Figure 3.13.: OMOS diagram “OutPCcCo” (version 1).

The OMOS modellers explained that these sub- and part-classes represent different communication modes and that class “CL_OutPCcCo” coordinates the flow between the different modes — from mode “07” to mode “07+08” to mode “08.” This flow is expressed in the diagram by ordering the respective class symbols from left to right — mode “07” left of mode “07+08” left of mode “08.”

This diagram again demonstrates the importance of semantic ordering used by modellers to convey semantic information through the layout of diagrams. However, the ordering found in diagram “OutPCcCo” is in contrast to diagram “OutP_Lss” and “OutP_Hss” (discussed in Section 3.5.2 and Section 3.5.3, respectively) where the semantic ordering of class symbols is motivated by the modellers’ intent to visually order class symbols according to the domain layers the depicted classes belong to. In terms of diagram “OutPCcCo”, the modellers’ intention for ordering class symbols is to visually convey the flow of information coordinated by class “CL_OutPCcCo.”

Furthermore, when class “CL_OutPLssCtl” was added in version 2 of diagram “OutPCcCo” (see Fig. 3.14), it was placed next to and at the same horizontal position as “CL_OutPCcCo”’s three sub-classes even though it is not a sub-, but an associated class of “CL_OutPCcCo.” In contrast, in version 3 of this diagram, class “CL_OutPSys” was added at the diagram’s top even though it is associated by class “CL_OutPCcCo” in the same way class “CL_OutPLssCtl” is.

The OMOS modellers explained that class “CL_OutPLssCtl” plays an active part in controlling and managing the communication controlled by class “CL_OutPCcCo.” That is why class “CL_OutPLssCtl” is positioned in close proximity to “CL_-

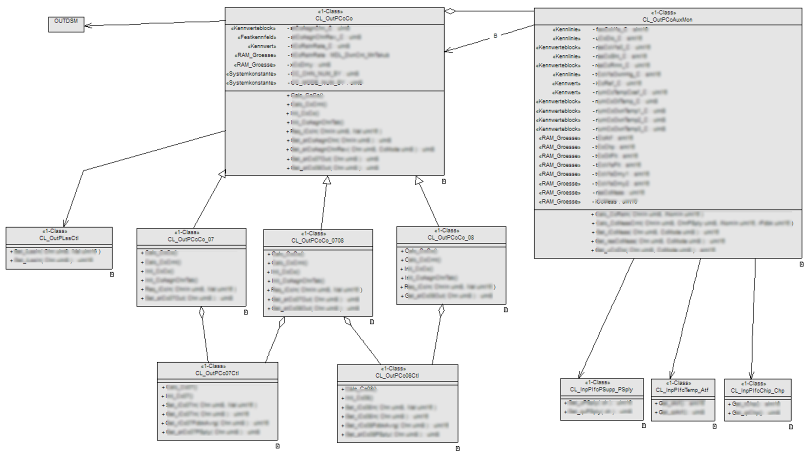


Figure 3.14.: OMOS diagram "OutPCcCo" (version 2).

OutPCcCo"'s three sub-classes instead of positioning it next to the other associated classes "OUTDSM" and "ECE_PSYS_D." Furthermore, class "CL_OutPLssCtl"'s public methods are displayed in the diagram which is not the case for all other associated classes. This further demonstrates the different role (compared to all other associated classes) of class "CL_OutPLssCtl" in this diagram.

This last finding again demonstrates that, for OMOS modellers, grouping class symbols according to their domain meaning is an important means to convey semantic information. (Semantic grouping is also discussed in Section 3.5.1 and Section 3.5.5.) This finding also shows that relationship symbols (i. e., connections between pairs of class symbols) are not necessarily a key factor for defining the position of class symbols as suggested by UML's layout guidelines.

3.5.5. OMOS diagram Fahrertyp as an example for semantic grouping

Diagram "Fahrertyp" (shown in Fig. 3.16), meaning "driver type" in English, defines classes which deal with the concept of determining and assessing various driver types in order to adjust the style of gear shifting to the car driver's driving habits. Class "CL_FTYP" (meaning "Fahrertyp" or "driver type") represents the whole-class of all other classes of this diagram. There are eight classes for assessing/weighting different driver types, for instance, kick-down (of the accelerator pedal) and sportiness, they are positioned below their whole-class class "CL_FTYP." The three other part-classes of class "CL_FTYP" do not present driver types, they provide functionalities to manage and access information from the driver types classes. Class

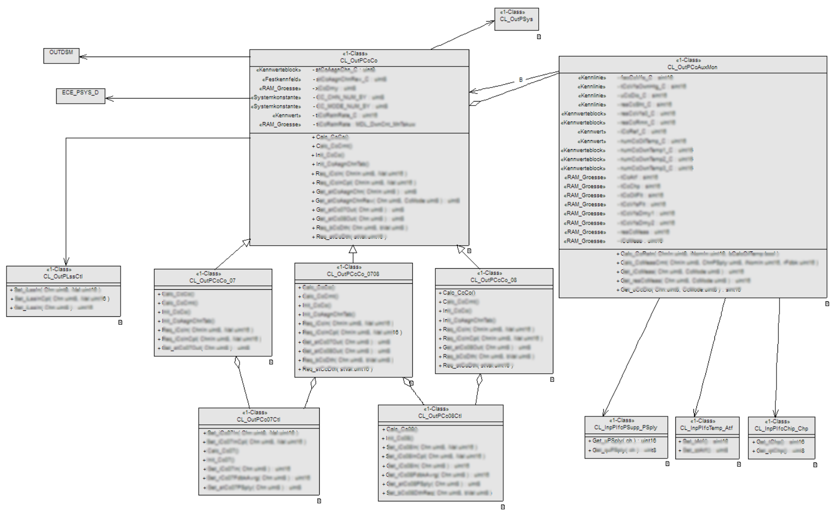


Figure 3.15.: OMOS diagram “OutPCcCo” (version 3).

“CL_TYERM” (meaning “Fahrertypermittler” or “driver type detector”) provides an interface to the driver type detection used by other classes (which are not shown in this diagram). Class “CL_TYZAL” (meaning “Fahrertypbewertungszähler”, or driver type assessment/weighting counter in English) is responsible for counting and weighting the information provided by the different driver type classes. Class “CL_TYVER” (short for “Fahrertypbewertungsverwalter” or “driver type assessment/weighting manager”) is responsible for internally managing and coordinating the driver type assessments.

Instead of following the UML layout guidelines suggesting to position part-classes below their whole-class, the layout of this diagram demonstrates the grouping of classes according to their domain meaning. For instance, class “CL_TYERM” is a part-class of class “CL_FTYP” (as are all the other classes in the diagram). However, in contrast to the other part-classes, class “CL_TYERM” is positioned next to the whole-class, not below it. This position is intentionally chosen by the OMOS modellers because class “CL_TYERM” presents the interface to the driver type detection functionality and is used by other parts of the ASIS software. It therefore belongs to a different domain level than the other part-classes of class “CL_FTYP.”

Class “CL_TYVER” and “CL_TYZAL” represent different semantic functionalities than the eight driver type assessment classes. To visually differentiate them from these classes, they are placed at different horizontal layers. Even though the

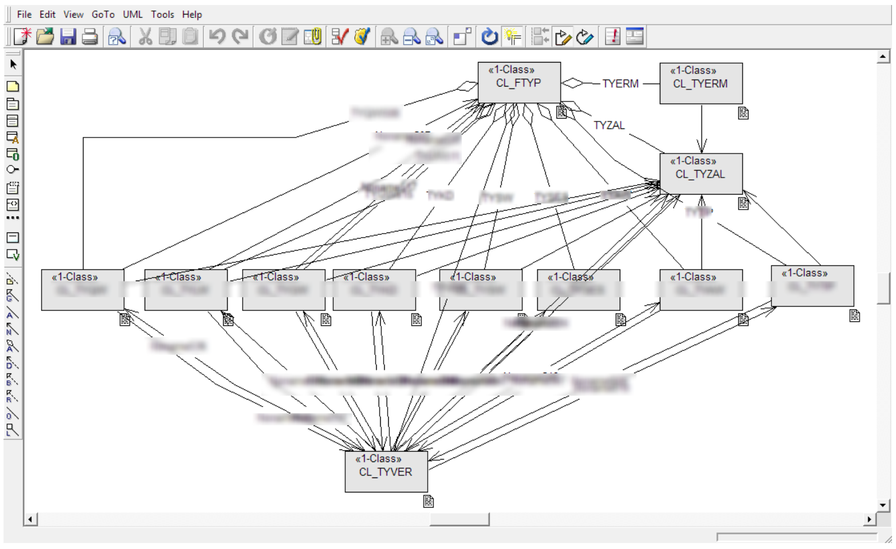


Figure 3.16.: OMOS diagram “Fahrertyp” (driver type).

diagram’s layout — with respect to the hierarchical arrangement of the diagram’s class symbols — visually exposes the whole-part class hierarchy as the classes are positioned at different horizontal layers, the whole-part hierarchy is not the most important criteria for positioning the part-classes in this diagram (as would have been the case when UML’s standard layout guidelines were followed).

The layout of this diagram again demonstrates that visually grouping class symbols according to their domain meaning is an important means to convey semantic information through the layout of a diagram. (Semantic grouping is also discussed in 3.5.1 and Section 3.5.4.) The layout of this diagram also shows that relationship symbols are not necessarily a key factor for defining the position of class symbols.

Furthermore, in order to avoid edge crossings class “CL_TYVER” is positioned below the eight driver type assessment classes, and class “CL_TYZAL” above them. Also, a balanced layout is tried to achieve for class “CL_FTYP” and “CL_TYVER” by horizontally centring them below or above the other classes.

3.5.6. Additional OMOS diagrams

As discussed in Section 3.3.2.2, all diagrams of both OMOS models were analysed as part of this pilot study. Appendix C provides the analysis findings of additional OMOS diagrams which could not be included in this chapter due to space limitations.

In Section C.1, package assignment diagrams are discussed. OMOS modellers use this kind of diagrams to assign each class of a OMOS model to a UML package. As identified by the diagram analysis conducted as part of the pilot study and as confirmed by the OMOS modellers who created them, package assignment diagrams do not convey domain-specific meaning through their layout.

3.6. Summary of the findings of the model analysis regarding expressing additional semantic information in diagrams of models through their layout

A summary of the results of the analysis of the OMOS models discussed in Section 3.4 and Section 3.5 is provided in this section. Following research objective 2 defined in Section 1.2.2, this analysis set out to determine how OMOS models are created and if and how additional semantic information are conveyed through the layout of diagrams of OMOS models.

3.6.1. Modelling by means of diagrams is a creative process

In this section, the study's first research question (How are diagrams used to visually construct OMOS models?) is discussed and linked to the study's findings.

The analysed OMOS models are partitioned into a large number of diagrams. As the number of diagrams in Table 3.3 and Table 3.4 (in Section 3.4) indicate, both analysed OMOS models consist of many (in total about 200) diagrams. Each diagram represents a different part of the software system's domain which is depicted by the symbols in the respective diagram. The model elements belonging to a certain diagram realise a certain part of the overall functionality the respective model provides.

The partitioning of the model into diagrams is often, but not always organised along the lines of the model's whole-part hierarchy and the inheritance hierarchy. A single diagram focuses on defining the classes which represent a certain domain functionality; it "encapsulates" the classes representing a certain domain-specific part of a OMOS model. Depending on the number of classes realising this functionality, the classes could be defined in several diagrams if there are too many classes to fit into one diagram. The name of a diagram reflects which part of the software system (represented by the model) the classes contained in this diagram realise. The diagrams' names thus represent domain knowledge.

New diagrams are created when new functionality, represented by one or more new classes, is added to a model. A OMOS model is partitioned into diagrams according to the model's semantic building blocks, i. e., the automatic gearbox controller domain concepts it represents. Even if the functionality realised by new classes conceptually belongs to an existing diagram, adding new classes results in new diagrams if an existing diagram becomes too complex in terms of the layout of its whole-part and/or inheritance hierarchies. Usually, all elements of a diagrams fit on a single screen. If they do not fit on a screen, they are arranged in a way which requires scrolling in only one direction (usually the vertical axis).

The findings are confirmed by other research too. Moody [Moo10] argues that using smaller, less complex diagrams instead of creating just one big, complex diagram helps to cope with complexity. Hahn & Kim [HK99] state that decomposition and layout organisation help to decrease the number of analysis and design errors. As discussed in the literature review on diagram layout in Section 3.2.2, assigning classes to diagrams is a creative act as it focuses on the domain meaning of the classes rather than its hierarchical features defined by means of UML elements (when following UML hierarchies this would mean to, for instance, include all part-classes and subclasses for each whole-class in a diagram). This finding is also supported by other research: Complexity has a major effect on cognitive effectiveness as the amount of information that can be effectively conveyed by a single diagram is limited by human perceptual and cognitive abilities. Moody states that the number of "bubbles" per diagram should not exceed seven plus/minus two because the human working memory is limited to dealing with this number of things at the same time [Moo10].

3.6.2. Summary of the pilot study findings regrading semantic information conveyed through the layout of diagrams and the mental map

The research presented in this chapter set out to investigate the importance of diagram layout for model-based software engineering in generally by analysing a specific approach by means of a pilot study. The most important finding of the pilot study presented here is that diagram layout is important for OMOS as diagram layout conveys additional domain-specific meaning which is important for modellers.

The study's findings on semantic information conveyed through the layout of diagrams provided in Section 3.5 confirm that modellers convey additional semantic information through the layout of OMOS diagrams. In regards to the study's first proposition (see Section 3.3.2.1), the pilot study confirms that this information is only defined by means of secondary notation, but is not formally defined in the model itself (i.e., there are no model elements defining the conveyed meaning). Modellers use grouping and ordering of class symbols to visually indicate that the underlying class symbols are semantically related, that a certain domain order or layering exists (in the modelled software system), and that information flows in a certain order from one depicted class to another.

As the pilot study shows, the position of class symbols (i. e., diagram symbols representing classes) depends on the class's domain meaning, for instance, its meaning in the domain of automatic gearbox controller software. Another finding is that the position of class symbols also depends on the meaning of the *specific* class diagram to which they belong, i. e., the way class symbols are laid out depends on the visual context in which they are visualised. Another finding made by the pilot study is that even in diagrams whose symbols are laid out according to UML's guidelines certain secondary notation, like the order of adjacent class symbols, is used by modellers to convey domain-specific meaning through the diagram's layout.

In the following, the study's propositions and (remaining) research questions (the first question has been discussed in Section 3.6.1) are connected with the study's findings and the results from other research identified in the respective research literature on meaning conveyed through the layout of diagrams.

3.6.2.1. Question 2: Which particular semantic information are implicitly conveyed by means of diagram layout?

The additional semantic information modellers intent to convey through the layout of OMOS diagrams are close semantic (domain) relations, domain layering, and domain flow. The layout features used to convey this semantic information are discussed next.

3.6.2.1.1. Domain layering: In order to express a certain domain layering of classes, the class symbols belonging to a layer are visually separated from class symbols whose classes belong to different software layers. The classes of each layer are grouped together, and the groups of classes are ordered from left to right. Modellers applied layering to convey information on how the (respective underlying) classes (of the depicted and visually ordered class symbols) are assigned to the different domain layers of the software system. For instance, in diagram “OutPLss” (see Section 3.5.2) and diagram “OutPHss” (see Section 3.5.3), classes are assigned to layers “L2” and “L3.” This layering is depicted by ordering the respective classes from left to right (with classes belonging to layer “L3” positioned on the left and layer “L2”’s classes positioned further to the right). Class symbols depicting classes belonging to the same layer are grouped, but not ordered, in close visual proximity.

3.6.2.1.2. Domain flow: Modellers express the flow of domain information in a similar layout manner as used for expressing layering discussed above. For instance, modellers ordered classes in diagram “OutPCcCo” (see Section 3.5.4) from left to right to emphasise that the information provided by (instance of) the ordered classes flow from left to right.

3.6.2.2. Question 3: Which kinds of diagram symbols convey additional domain-specific knowledge through their of layout?

The pilot study’s findings identified that domain-specific knowledge is conveyed by *class* symbols. This finding is not surprising since the diagrams used to build OMOS models (and analysed by the study) are UML *class* diagrams and the most important symbol type of this diagram notation are class symbols.

Classes (and the symbols depicting them in diagrams) represent the main components of the software design models realised by both analysed OMOS models. Each class represents a certain software functionality in the domain of gearbox controller software.

3.6.2.3. Question 4: Which layout features are considered important by OMOS modellers in order to express implicit domain-specific knowledge through the layout of diagrams?

The study’s findings identified that OMOS modellers use visual grouping and ordering of class diagram symbols as means to convey domain-specific knowledge through the layout of diagrams.

When the results of the diagram analysis conducted as part of this study were discussed with the OMOS modellers, they stated that they use visual grouping and ordering of (symbols of) semantically related classes. However, they do not consider the grouping of (possibly semantically related) relationship symbols depicting association and inheritance relations and symbols depicting packages important.

3.6.2.3.1. Visual grouping of class symbols: Visual grouping is expressed by means of visual proximity. For expressing grouping of class symbols, the particular order, i. e., left-right or top-down, of these symbols is not important to modellers, only their visual proximity is. Diagrams "OutP_IFC_General" and "OutP_IFC__General_Exp" discussed in Section 3.5.1 demonstrate the secondary notation for the visual grouping of class symbols.

Visually grouping semantically related class symbols is often used when UML inheritance or whole-part relationships are visualised in a diagram too. Following the UML relationships, the diagram is then hierarchically laid out: base/whole classes above sub-/part-classes. Class symbols of semantically related classes are then adjacent *within* such layout hierarchy. Therefore, when UML inheritance and hierarchies are depicted in a diagram, semantically related base classes are grouped on the same horizontal layer above semantically related sub-classes which are also grouped on another horizontal layer (below the base classes). For instance, if the (underlying UML) classes of grouped class symbols form an inheritance hierarchy shown in the diagram, the base classes are positioned above sub-classes. Nevertheless, semantically related base and/or sub-classes are positioned in close visual proximity.

3.6.2.3.2. Visual ordering of class symbols: Similar to visual grouping, visual ordering is expressed by means of visual proximity of class symbols. In contrast to visually grouped class symbols, when OMOS modellers use the ordering of class symbols to express domain meaning, the symbol's particular left-to-right order is indeed important. In the analysed OMOS diagrams, modellers use visual ordering to express domain layering and flow (see above). No cases were found during the diagram analyses where class symbols were ordered vertically (top-down) and conveyed additional semantic meaning apart from the meaning express by UML relations (i. e., inheritance and whole-part or association relations).

3.6.2.3.3. Summary: According to the modellers who created the OMOS diagrams analysed as part of this study, the decision of where elements are placed is (often) made by the modellers based on their knowledge and understanding of semantic relationships between elements and how they intent this knowledge to be represented in a diagram. When diagrams are created (or modified), software engineers, therefore, apply layout rules that are often informal but intentional for the domain to which the elements depicted in the diagram belong. The analysis of OMOS models shows that the position of an element in a diagram depends on an element's semantic meaning, i. e., the element's domain-specific meaning. For example, modellers often visually group and order symbols of UML classes based on their semantic meaning in the application domain. The domain-specific ordering of class symbols can result in layouts which are in contrast with UML's layout guidelines. Often, the UML class diagram layout guidelines are not followed in the analysed OMOS diagrams, for instance, base classes or whole-classes are *not* positioned above sub-classes or part-classes, respectively. The domain meaning expressed by the layout hence takes precedence over the UML guidelines; modellers consider the domain meaning more important and it thus overrules the suggested UML layout guidelines. This finding is further supported by other research regarding diagram layout discussed in Section 3.2.2 and Section 3.2.3. Modellers working with OMOS diagrams are actually asked (but not rigorously enforced) by Bosch-internal OMOS diagram layout guidelines to adhere to the UML class diagram layout guidelines defined by UML. Moody argues that the layout guidelines given by the UML standard are flawed in several ways [Moo10]. As the results of the diagram analysis show, those guidelines are not followed rigorously for OMOS diagrams. Hence, diagram layouts can differ and are merely subject to the interpretation of the modellers who create or modify them. The layout that modellers choose for a diagram is intentional and can follow informal, unspecified rules. The position where class symbols are placed is chosen according to the symbol's semantic (i. e., domain) meaning and the modeller's understanding of this meaning. Elements that are closely related in terms of their domain semantics are likely to be positioned in close visual proximity in a digram.

The principle layout of class symbols is (often but not always) determined by the "model facts" to be expressed in a diagram, i. e., UML model hierarchies (inheritance, compositions and associations) influenced the visual context a class symbol belongs to. For example, the proximity of class symbols is determined by their connections to other class symbols. Class symbols close to each other are often connected by inheritance or association connections. For instance, base classes are positioned above sub-classes which are positioned in close proximity below their base classes (top-down layout). On the other hand, the visual relations of classes, i. e. their layout, of the classes is not always as advertised by the UML layout guidelines (see Section 3.2.2). In such cases, domain-specific facts “overruled” the UML layout suggestions. For instance, the position and grouping of several class symbols connected with each other by means of UML relationships (inheritance, association or composition) does not necessarily follow the layout suggested by UML. For instance, when two or more sub-classes are connected with the same base class, not all of them may be positioned below the base class (as suggested by the UML layout guidelines). A similar finding was made for composition: Given several part-classes, connected to the same whole-class, not all of them are positioned below this whole-class. For instance, the diagrams discussed in Section 3.5.4 demonstrate such layouts. Hence, even if the UML class diagram layout guidelines are followed, class symbols are often ordered according to their domain meaning. Symbols of closely related classes are then positioned in close visual proximity to each other. This is, for instance, the case for containment (whole-part) and inheritance hierarchies. The visual grouping and ordering of class symbols within such a hierarchy is then defined by the domain meaning of the depicted classes.

3.6.2.4. Question 5: To what extent do diagrams convey domain-specific knowledge through their layout?

Not all class symbols of every diagram convey additional semantic meaning (solely expressed by their layout through secondary notation). However, for many diagrams at least a number of class symbols do. About two thirds of the analysed OMOS diagrams convey domain-specific knowledge through their layout. This holds for all three versions of the both analysed OMOS models.

Most (about 95 per cent) of the analysed diagrams are used to define the classes constituting the modelled software system[7]. Classes belonging to a diagram are often semantically related since these classes together realise a certain part of the model’s functionality. Because a diagram depicts semantically related classes, there is a high possibility that diagrams convey additional meaning through their layout if such meaning exists and is important for modellers.

[7]The remaining less than 5 per cent of the diagrams are package assignment diagrams which do not convey additional semantic meaning through their layout (see Section C.1).

3.6.2.5. Question 6: Is the mental map of diagrams important and what drives diagram layouts changes in consecutive diagram versions?

Diagrams dealing with similar domain concepts, i. e., representing classes whose semantics are closely related, often expose a similar layout structure (see, for instance, Fig. 3.12, Fig. C.5, and Fig. C.6). Together with the finding that certain groups of classes appear in similar groupings/have similar ordering in several diagrams, this finding confirms the importance of the diagrams' mental map (see Section 3.2.4) for OMOS modellers. The overall layout of the diagrams of both analysed OMOS models does not change significantly over the course of the three analysed versions. This finding confirms the study's second proposition, even though diagrams provide a high degree of freedom with respect to layout, the mental map of a diagram and, therefore, stable layout is important to modellers — even in consecutive versions of a diagram.

When diagram symbols were added in the version 2 or 3 of the analysed models, the overall layout of the respective diagrams did not change. When class symbols were added to a diagram, they were positioned in close proximity to existing class symbols depicting classes semantically related to the new classes. Then, existing diagram symbols were often rearranged to make space for the new symbol. Such rearrangements were done in a way that preserves the overall relationship of a diagram's class symbols. Class symbols that were positioned in close proximity in the old layout were also close in the new layout. Hence, the visual context of class symbols did not change.

Several new diagrams were added in later version of the analysed models, for instance, the diagram shown in Fig. 3.12 in Section 3.5.3. The new diagrams define concepts similar (but not identical) to those already defined in earlier versions of the models, for example the diagram depicted in Fig. 3.12 is similar to the one depicted in Fig. 3.9 in Section 3.5.2 which already exists in the first analysed model version. Newly defined diagrams are laid out in a similar way as diagrams created in earlier versions.

In the analysed diagrams, symbols were only rearranged when new class symbols were added to a diagram. The analysis could not identify other reasons for diagram layout changes. A possible reason for rearranging class symbols may, for instance, be that the domain meaning of the depicted class, i. e., the modellers' understanding of this class, changed as the understanding of the domain grew. However, such considerable layout rearrangements could not be observed by the diagram analysis.

One reason for this could be that too few model versions were taken into account by the pilot study.

3.6.2.5.1. Visual context: With respect to the mental map, the *visual context* of class symbol is, therefore, defined by its adjacent class symbols, but also by the semantic meaning conveyed by this symbol. The visually closest adjacent symbols might therefore not always fully define a symbol's visual context, but symbols in close, but not immediate visual proximity can be important, too, for the symbol's visual context as perceived by modellers.

3.6.2.5.2. Summary: Regarding the study's findings on the importance of the mental map of diagrams, it can be summarised that the mental map is important to OMOS modellers since the principle layout of the analysed diagrams is stable. Furthermore, as discussed in Section 3.1, OMOS modellers insisted that the consciously chosen relations of diagram symbols are important to them. It, therefore, follows that this statement, too, indicates that the mental map of the diagrams is important for the OMOS modellers.

After the pilot study's research questions and propositions have been discussed in this section, the next section will provide an overview of additional, more general findings regarding the layout of the OMOS diagrams analysed as part of the pilot study.

3.6.3. General findings regarding the layout of OMOS diagrams

The following general findings regarding the layout of the OMOS diagrams were made by the OMOS diagram analysis conducted as part of the pilot study presented in this chapter:

1. OMOS diagrams are laid out following the UML class diagram layout guidelines (see Section 3.2.2) and implicit layout rules conveying additional domain knowledge. The latter rules are more important to OMOS modellers than UML's layout guidelines.
2. Even though the same model element (class, package, inheritance, association, composition etc.) could be depicted more than once in the same diagram using the Ameos modelling tool, none of the analysed diagrams contains a certain model element more than once, a certain *model* element is only depicted once

in a diagram. However, a model element can appear in many diagrams. This is because the model is partitioned into many diagrams and because classes are used in different contexts in different diagrams. That is why the same association and inheritance relations could also appear in many diagrams.

3. Packages are only depicted in the package assignment diagrams (see Section C.1), they are not used in the majority (about 95 per cent) of the diagrams comprising the analysed OMOS models (see Section 3.6.1).
4. All diagram symbols are drawn in black colour (the default colour of the Ameos modelling tool).
5. Regarding the diagram canvas size used to position diagram symbols, the OMOS modellers try to avoid scrolling in two directions. If a diagram exceeded the screen's size, symbols are positioned in a way that requires only scrolling on one axis (usually to vertical one). The width of a diagram usually does not exceed the width of a standard desktop PC's screen resolution. The hight of a diagram could exceed the screen's height.

The following findings regarding the layout of *class symbols* and the expression of additional semantic information in diagrams through their layout were made:

1. The absolute position of a class symbol is meaningless, the symbol's proximity (visual context) and relation the other class symbols is important for the modellers' mental map of a diagram (see Section 3.2.4) and for conveying additional semantic information through the diagram's layout.
2. Classes are often laid out in a top-down, left-right manner. Thus, class symbols are often arranged in horizontal layers.
3. For top-down layouts, classes on a upper layer are often centred above classes on the next lower layer. With respect to layout aesthetics, some degree of layout symmetry is thus tried to achieve if UML class hierarchies are depicted in a diagram.
4. Class symbols do not overlap (which is a fundamental requirement for creating readable diagrams).

The following findings regarding the layout of *connection symbols* and the expression of additional semantic information in diagrams through their layout were made:

1. Connections between adjacent class symbols are often drawn as straight lines. OMOS modellers stated that (with the Ameos modelling tool) straight connections are easier to draw than bended ones.

2. Where possible, the crossing of straight lines is tried to be avoided since such lines usually connect adjacent class symbols. However, many line crossings can be found in the analysed symbols, often for longer, bended lines.
3. Orthogonal routing of the line segments of connections is not overly important.
4. The visual context of class symbols is more important to OMOS modellers than the possibility to draw connections as straight lines. Positioning class symbols in their preferred visual context overweighted the class symbol's connections to other class symbols in this diagram.
5. There is no preferred direction of connections. If a diagram's layout mainly focuses on depicting a UML hierarchy (with class symbols being arranged in horizontal layers), then a top-down direction of connections is preferred.
6. Unlike class symbols, connections are *not* ordered according to their domain meaning. The OMOS modellers stated that, for them, associations and inheritance relations have no additional domain-specific meaning.
7. Regarding their type, connections are usually not mixed, i. e., in a certain diagram, the same two connected class symbols are either connected by an association or generalisation edge, but not by both. The reason is that with the OMOS approach is based on whole-part and inheritance hierarchies. There hence is usually no reason for a whole- or part-class to also define a generalization from the opposite part- or whole-class and vice versa for generation relations between the two classes.

3.7. Chapter summary and outlook

This chapter discussed research objective 1b (determine from the research literature whether diagrams of models contain additional semantic information through their layout) and 2 (carry out a pilot study to verify that diagrams actually convey semantic information in the pilot study environment and identify the extent of the problem), defined in Section 2.1.1.

The results of the pilot study presented in this chapter show that the analysed OMOS diagrams have layout features which do not always follow the layout conventions suggested for UML class diagrams (see Section 3.2.2). These layout features are intentionally used by the OMOS modellers to visually express semantic meanings and relations between model elements (by means of secondary notation, see

Section 3.2.3). The semantic meaning conveyed by a diagram's layout is neither part of the model nor is it formally defined in the diagram, it is only known to the modellers (domain experts) who create the diagrams. For instance, class symbols depicting semantically related classes are ordered according to their meanings in the automotive controller domain (see Section 3.5.1), and classes are laid out according to the domain layers these classes belonged to (see Section 3.5.2 and Section 3.5.3).

While acknowledging the limited scope of the pilot study, its results confirm the general importance of diagram layout for users of model-based software development approaches by analysing a specific case. The most important finding of the pilot study is that OMOS modellers use layout to convey domain-specific meaning which is informal and not part of the model.

The findings on semantics information conveyed through diagram layout are not unique to the analysed OMOS models. The author had the opportunity to informally look at OMOS diagrams created by another engineering group at Bosch developing software for anti-lock breaking systems. These diagrams reveal similar layout patterns as identified in the diagrams analysed as part of the pilot study discussed in this chapter. The author later had access to diagrams from the space systems engineering domain [EMdK09]. Again, these diagrams show similar layout pattern. In all those cases, the additional semantic meaning expressed through the layout of diagrams is not defined by the model itself, it is only manifested in the diagram layout by means of secondary notation. Such phenomena are well known and described by other research as the literature review provided in Section 3.2 confirms.

Based on the findings summarized above, the principles on which any software should be based to provide a solution for working in parallel with diagrams of models will be defined in the next chapter.

4. Principles and objectives of an approach for collaboratively working with OMOS models

As discussed in Section 1.2.2, an important objective of the research presented in this thesis is to overcome the limitation of only one modeller being able to modify a OMOS model at a time and enable a collaborative approach to OMOS modelling. This chapter focuses on research objective 3 defined in Section 1.2.2: the general principles are established which any software should be based on to provide a solution for working in parallel with OMOS models and their diagrams. Furthermore, principles are established which are specifically drawn from the findings of the pilot study on semantic meaning conveyed through the layout of diagrams (discussed in Section 3.6).

4.1. Automatic diagram layout and additional semantic information conveyed through the layout of diagrams

As stated by other research discussed in Section 2.4, when diagrammatic models are merged manually, laying out their merged diagrams is not an option for models consisting of more than a few diagrams. The same conclusion can be drawn from the pilot study discussed in the previous chapter as the OMOS models analysed as part the study consist of almost 200 diagrams (see Section 3.4). Therefore, automatic layout of UML class diagrams appears to be a feasible solution for enabling parallel work on OMOS model. Ohst et al. [OWK03b, OWK03a, OWK04] advocate for the usage of automatic layout for merging UML class diagrams (see Section 2.4.1)[1]. This section, therefore, validates the suitability of automatic diagram layout for enabling

[1]The foundations of automatic layout approaches and an overview of the concepts and tools for the automatic layout of UML class diagrams are discussed in Section 2.4.3.

parallel modelling with OMOS taking into account the findings from the pilot study and from the research literature on conveying additional semantic information in diagrams of models through their layout provided in Chapter 3. That is, having established that layout is important to modellers, it has to be preserved when models are merged. However, as the following section describes, current automatic layout approaches are not suitable for this purpose. Automatic diagram layout approaches are, therefore, evaluated regarding their support for parallel modelling with OMOS in Section 4.1.

As discussed in Section 3.1, OMOS modellers rejected the usage of automatic layout approaches. They claimed that semantic information conveyed solely through the layout of OMOS diagrams is lost when the diagrams are laid out automatically. Based on the pilot study's findings regarding which and how additional semantic information is actually conveyed through the layout of OMOS diagrams (see Section 3.4 and Section 3.5), automatic layout approaches are evaluated in this section to determine whether such approaches are able to take into account the modellers' intention to convey additional semantic information through the layout of diagrams.

4.1.1. On the usefulness of automatic graph layout for UML class diagrams when domain semantics of visualised model elements are important to modellers

Automatic UML layout techniques (see Section 2.4.3) lay out diagrams by applying layout rules which are solely based on information from the UML model. For laying out UML class diagrams, these information are taken from the UML (class) model. For example, classes and packages represent (graph) vertices and association and inheritance relationship between classes represent (graph) edges. Fully automatic layout hence cannot/does not take the layout of existing, manually created diagrams into account, furthermore, the diagrams do not yet exist because they are automatically created. To produce readable and aesthetically pleasing looking layouts, *aesthetic criteria* are applied (to arrange the (graph's) vertices and edges), for instance, rules for creating layouts with few edge bends, orthogonal edges, and symmetrically distributed nodes.

Automatic layout algorithms group diagram symbols based on the semantics of the UML diagram notation (see Section 3.2.2), for example, the class symbols depicting a certain inheritance hierarchy or symbols depicting classes involved in an association relation are positioned in close visual proximity. However, even though it is called *semantic clustering* [Eic05], automatic layout algorithms do not and cannot group diagram symbols in a way that conveys semantic meaning inherent to the application domain, i. e., a layout that modellers would create if these elements were placed by humans, since their manually created layouts convey semantic information inherent to the application domain (as the study shows, this is important for OMOS modeller too). By semantic/domain meaning it is referred to an element's meaning in its domain, not the UML semantics (for instance, classes belonging to the same UML inheritance hierarchy). Automatic diagram layout algorithms produce layouts that are based on a set of aesthetic criteria. Hence, diagram symbols are placed at the best position according to theses criteria, not according to their semantic meaning in the application domain. These criteria are based on aspects such as placing a certain set of vertices near to each other or specific vertices on top of others or in the centre of a drawing with respect to the meaning of these elements regarding the elements' UML meaning, not their meaning in the application domain since these information are not known to automatic diagram layout algorithms.

Automatic diagram layout algorithms for UML diagrams are often used for re-engineering [BETT99, OWK03a], i. e., when existing source code artefacts are used to create UML diagrams, such as class, collaboration, or activity diagrams. Automatically generating diagrams from source code is useful since it would be very tedious and error-prone to create them manually — especially because the model that is underlying to and represented by the diagram has an already existing pendant: the source code. However, it is argued that when diagrams are created from scratch, developers have to be able to create diagrams in a way that does not inhibit their creativity and freedom to create diagrams according to their understanding of the domain problem or solution depicted in the diagram.

Batin et al. [BFN85] and Fleicher & Hirsch [FH01] state that automatic layout tools are often based on fixed presumptions concerning layout aesthetics and trade-offs between conflicting layout rules. Whereas modellers apply their implicit and intimate knowledge about the semantics of the application domain to solve such trade-offs. Thus, the human perception of diagrams depends on the application domain of the diagrams. Eichelberger [Eic05] suggests to define a rule set that follows the behaviour of modellers drawing UML class diagrams. Batin et al. [BFN85] suggests to enumerate as many layout criteria as possible and rate them in terms of modeller preferences for solving conflicts between these criteria. Sugiyama [Sug02] states that, for automatic layout approaches, semantic rules, i. e., rules depending on the semantics of diagram elements and on the application domain, should be rated with a higher priority than structural rules, i. e., common rules that are independent from the meaning of elements and that are common to all diagram layouts.

Dengler et al. [DFM93] suggest that perceptual organisation has to be taken into account in addition to pure layout aesthetics. The authors argue that in order to automatically generate high-quality diagrams, for instance, certain groups of symbols must form perceptual groupings by visual proximity. Whilst their study does not consider UML class diagrams in particular, the finding that diagram symbol proximity is important to modellers applies to UML class diagrams too as the findings of the pilot study show (see Chapter 3).

When a model changes, because, for example, elements are added, deleted, or modified, the new diagram layout produced by automatic layout algorithms can be very different from the previous layout. Even small changes to the model can result in significant layout changes since automatic layout algorithms often apply incremental layout and layout heuristics since optimising layout criteria (like minimising the number of edge crossings and minimising edge bends) which are often NP-hard computational problems [Nor95, See97, Eic06]. The mental map (discussed in Section 3.2.4) that modellers have established for an existing diagram, therefore, gets destroyed/becomes meaningless. This drawback can be diminished when *dynamic graph drawing* algorithms are used, also called *interactive graph drawing* algorithms since graphs are modified interactively by the user [BETT99]. The goal of these algorithms is to preserve the mental map of a graph when it is changed [BT00] such that modellers are not forced to spend considerable effort relearning the new diagram, i. e., to visually parse [Moo10] and understand it.

Approaches to preserving the mental map often try to minimise the changes between diagrams. This is achieved by allowing only limited modifications of an already existing layout regarding the position of vertices and edge bends in the diagram [MELS95]. However, to the best of the author's knowledge there appears to be no CASE tool that supports the layout of evolving models (i. e., dynamic graphs). Eichelberger explicitly emphasises that the "SugiBib" diagram layout algorithm (see Section 2.4.3) does not consider dynamic graphs [Eic05]. Eiglsperger and his colleagues admit that incremental editing can be easier especially when diagrams are complex [EGK^{+}04]. Their "yFiles" graph layout tool which is based on the "GoVisual" automatic graph layout algorithm (see Section 2.4.3) supports dynamic graphs but provides only very basic support for elements as they are used in UML class models/diagrams since it is a general purpose graph layout tool and by no means a CASE tool.

The proximity of elements is another property that is considered important for the user's mental map [KW01]. Miuse et al. ask for dynamic graph layout algorithms that honour the proximity of elements when existing layouts are adjusted due to modifications of the underlying graph [MELS95]. When diagrams are modified, proximity relations should be preserved in order to preserve the user's mental map. Hence, diagram symbols which are close together should stay close together. Elements that are close in the old layout should also be close in the new layout. North argues that the position of elements is more important than the routing of edges because elements are remembered as locations, while edges are traced "on the fly" [Nor95].

Sugiyama states that UML class diagram layout methods in particular are relatively underdeveloped [Sug02], and Eichelberger [Eic02b] compared forty-two CASE tools regarding their ability to automatically create UML class diagram layouts and concludes that none of the tools is able to create readable layouts. This situation might have changed today, since the two new graph layout algorithms, "SugiBib" and "GoVisual", can produce better UML class diagram layouts according to certain layout aesthetics. However, as stated by Lee et al. [LLY06], the concept of the mental map is largely ignored by many graph drawing techniques. This finding appears to hold for automatic UML class diagram layout algorithms as well because only one approach exists that pays attention to the mental map by applying dynamic graph layout techniques as described in this section.

4.1.1.1. Diagram layout aesthetics criteria, diagram comprehension and semantic information embedded in the layout of diagrams

Several studies evaluate different types of class diagram layouts to reason about which layout styles result in diagrams that are easy to understand. Purchase et al. conducted usability studies on software engineering documents, for instance, UML diagrams and entity-relationship diagrams [PCM$^+$01, PCA02], and in [WPCM02] and [Pur04] they present the results of experiments with computer science students on the perception of diagrams as well as a number of metrics for diagram layout aesthetic criteria originating from graph drawing. The studies aim to identify the aesthetic criteria that are most important for the visualisation of diagrams with respect to the extent to which the aesthetics produce diagrams that are easy for human readers to understand. Subjects were asked to perform comprehension tasks on the same UML class diagram laid out with different aesthetic emphases. The results reveal that modellers preferred diagrams with fewer bends and crosses, shorter edge lengths and an orthogonal structure. However, the authors of these studies state that the semantics of the diagram are not taken into account by automatic diagram layout algorithms. They state that there are additional semantic aspects that have to be considered when layout algorithms are used for domain-specific approaches [PMCC01]. Ware et al. report that most of the layout aesthetics criteria have not been validated by empirical studies. Many criteria discussed in the graph drawing community refer to general graphs where edges are lines and vertices are points which do not bear any domain/semantic meaning [WPCM02].

As discussed in Section 2.4.3, Eichelberger developed an approach that automatically lays out UML class diagrams according to certain layout aesthetics [Eic05]. He states that there are no commonly agreed criteria to describe the features necessary to support for diagram readability. Eiglsperger et al. [EKS03] conclude that it is not clear which is the most important aesthetic criterion for class diagram layouts. Aesthetic criteria regarding edge crossing and the number of bends of an edge seem to play an important role. They state that the importance of aesthetic criteria that lay out (model element) hierarchies in a certain direction, for instance, class inheritance hierarchies in vertical direction, and position diagram symbols in an orthogonal manner is not backed by the above studies, partially because they have not been investigated. Because the situation seems to be unclear they advocate for an automatic layout algorithm for UML class diagrams that is flexible enough to let the modeller choose the aesthetic criterion to optimize diagram layouts for.

4.1.1.2. Automatic layout of multiple diagrams

Different UML class diagrams are usually used to either display different parts of a model at the same level of abstraction, for instance, several diagrams represent the functional elements of a system, or to display the system at different abstraction levels, for example, diagrams representing high-level views on the domain analysis level and low-level diagrams presenting functional and non-functional elements on the implementation level. UML class models *per se* do not contain enough information to automatically decide which model elements should be displayed in separate diagrams. Dividing UML class models into several diagrams when they are generated using an automated approach can only be based on model information. For instance, a diagram could be generated for each UML package and its classes. However, automatic approaches cannot decide when new diagrams should be created for a certain group of elements that should be visualised in a separate diagram. This information would only be known to humans that are aware of the semantic of these elements in the application domain and who can also reason about a diagram's layout complexity. As reported by Ware et al., studies on diagram layout aesthetics and comprehension focus only on one diagram per evaluation presuming that one diagram is sufficient to represent the static structure of a software system. Also, automatic UML class diagram layout algorithms are developed with only one diagram in mind which then visualises all the model elements [WPCM02]. However, as the pilot study confirms, OMOS models consist of a (large) number of diagrams (see 3.6). Therefore, this drawback of existing automatic layout approaches has to be taken into account when deciding about its suitability for the envisaged approach to collaborative modelling with OMOS models.

4.1.2. Conclusions on visual UML class modelling and parallel software development

Ohst et al. [OWK03a, OWK03b] argue that state-of-the-art software development processes are often based on models, but concepts and tools that allow to efficiently include models into distributed development processes are not as evolved as required. The problem is that models are usually represented using diagrammatic notations and that the respective diagrams are created manually. Most tools that support distributed software development only consider textual software artefacts like source code. As discussed in Section 2.4, two research approaches support visually merging UML class models. However, both approaches do not pay the required attention to UML class diagrams when merging them. One approach creates a kind of overlay of the two diagram versions to be merged and, therefore, imposes the effort of untangling overlapping diagram symbols on the user. The other approach uses an automatic diagram layout approach. When this automatic approach is applied, the elements of a merged UML class model are laid out according to diagram layout aesthetics as discussed in Section 4.1.1.1. The original diagram layouts of both manually created to-be-merged diagram versions are not considered at all when the new layout is generated. Automatic diagram layout algorithms produces nice layouts according to aesthetic criteria, but they do not allow to preserve the mental map of a diagram when semantic information from the application domain is not explicitly described/modelled in the model but only expressed by the position and grouping of model elements, i. e., expressed by the diagram's layout (see secondary notation in Section 3.2.3). Thus, when diagram versions that are to be merged were created manually, automatic diagram layout algorithms are likely to destroy the meaning of the diagram that users intentionally and implicitly conveyed through the layout of this diagram. Thus, the user's mental map (see Section 3.2.4) is destroyed. Destroying the semantic information conveyed by means of secondary notation and, thus, destroying the mental map is not desirable because diagrams have to be relearned after there have been merged.

Using automatic layout algorithms for UML class diagrams has the advantage, that every diagram of a given model is laid out in the same way. This it the case because automatic layout algorithms apply rules to decide where diagram symbols (i. e., vertices for, for instance, classes and packages, and edges and edge bends for, for instance, association and inheritance relationships) are positioned. Fully automatic layout algorithms focus on laying out (UML class) diagrams in an aesthetically pleasing way based on the elements of the underlying UML model. They cannot take semantic information from the application domain into account since this information is not defined in the models. When dealing with evolving models, a major disadvantage of automatic layout algorithms is that the layout of the revised model's diagram can significantly differ from the diagram of the previous model version. Thus, the modeller's mental map of the previous version is destroyed. Hence, modellers are forced to spend a lot of effort relearning the newly laid out diagrams. Even worse, the semantic information conveyed by the diagram's layout is lost since it is not formally defined in the model itself and thus cannot be taken into account by automatic layout approaches.

These drawbacks caused the initial decline of automatic diagram layout by the Bosch engineers (as discussed in Section 3.1). The additional semantic information expressed through the layout of their OMOS diagrams could not be taken into account by automatic layout approaches and was lost when the models changed and, as a consequence, diagrams were automatically laid out again.

4.1.2.1. Summary

To summarise, the following conclusions regarding automatic diagram layout are drawn from the findings of pilot study presented in the previous chapter and the research literature discussed in this section:

1. Currently available (UML) model merge approaches and tools do not provide viable solutions for enabling parallel work on visual models which convey additional semantic information through the layout of their diagrams [Men02, OWK03a, OWK03b].
2. Conventional approaches to automatic (UML class) diagram layout are not a viable solution to parallel evolution of OMOS models because they cannot take into account the additional information conveyed only through the layout of the models' diagrams. Automatic diagram layout algorithms are based on layout aesthetics and information solely defined by the model underlying the diagrams. However, the additional information conveyed by OMOS diagrams is not described by means of model elements but only by means of secondary notation. As stated by Larkin & Simon [LS87, pp. 2] and, as confirmed by the pilot study, the visually grouping and proximity of diagram symbols play important roles for the modellers' understanding of diagrams. When diagrams are laid out fully automatically, this (solely visually conveyed) information is likely to be lost and the mental map is likely to be destroyed. Besides the drawback of losing additional semantic meaning conveyed through the diagram's layout, the mental map would have to be relearned after OMOS models have been merged. Therefore, laying out (evolving) diagrams (depicting evolving models) in a fully automatic manner will not work.

4.2. General principles on which any software should be based to provide a solution for working in parallel with OMOS models

This sections defines the principles and objectives on which any software should be based to provide a solution for the problem of working in parallel with OMOS models.

4.2.1. Motivation: the old sequential, non-parallel approach to working with OMOS models

As discussed in 1.2.1 and Section 2.2.2, each iteration of Bosch's sequential, i. e. non-parallel, software development approach works as follows:

1. Firstly, the objectives of the iteration are defined, i. e., the requirements (functionalities) to be realised by the OMOS-based software system under development are defined.
2. Secondly, in order to implement the requirements, the (already existing) OMOS model (or models) belonging to this software system has to be developed further, i. e., functionality is added to, modified in, or removed from this model. Each iteration is based on the previous iteration's model (or models). This model provides the basis for implementing the new iteration's requirements.
3. Thirdly, since the OMOS software development approach is a model-driven one, models are modified first, source code is then automatically generated from these models and implementation details which could not be derived from the model are manually added to the generated source code.

The existing OMOS tooling does not allow for collaborative modelling; only one team member can work on a OMOS model at a time. Hence, the OMOS tools do not support parallel modelling, only sequential, i. e. one modeller at a time, modelling is supported. One of the main objectives of this research therefore is to enable parallel work on OMOS models. Since the OMOS development team was located at different sites (in Hungary and Germany), this will allow the two groups of the team to work in parallel on the same models.

4.2.2. Optimistic, merge-based version control supporting parallel work on OMOS models

As discussed in Section 3.3.2.2, Bosch's projects for electronic control unit software realised with OMOS tend to consist of only few, but large models. Therefore, a pessimistic collaboration approach (introduced in Section 2.3.1) is not an option for enabling collaborative work with OMOS models. With the pessimistic approach model-based collaboration is enabled by granting exclusive access to a certain model to only one modeller at a time, locking the model for all other modellers. With the pessimistic approach, the need for merging different versions of a model and for dealing with conflicts (during the merge) does not exist since models can only be modified by one modeller at a time. However, locking a whole model to prevent modifications by more than one modeller does not offer any advantages regarding collaborative OMOS modelling. Hence, it does not provide any benefits with respect to more than one modeller working (on several tasks) *in parallel* on the same model.

As discussed in Section 2.3.4, model partitioning is another approach supporting parallel work (on the same model). It is related to the pessimistic approach. With this approach exclusive access to parts of a model is granted to only one modeller at a time by partitioning models on a per-element basis and granting exclusive modification (i. e., write) access to each model element to only one modeller at a time — locking these elements to prevent concurrent modifications by other modellers.

However, as explained in Section 2.3.4, locking single model elements is in general not very useful since it does not allow a modeller to accomplish a reasonable modelling task — such a task usually involves more than one model element at a time. For instance, the element's container has to be locked, too, in order to prevent other modellers from deleting the container element and thus its sub-elements. The need for locking additional model elements to allow a modeller to modify some model element is likely to result in large parts of a model being locked by a certain modeller. However, granting exclusive access to a large part of a model to only a single modeller contradicts the benefits of collaborative modelling — even more so because the same OMOS modeller usually works on different parts of a OMOS model and thus needs to access and modify various model elements at a time.

As explained above and as further discussed in Section 2.3.4, pessimistic (locking-based) collaboration approaches do not provide any advantages for the envisaged approach for working in parallel with OMOS models. As explained further in Chapter 2, each modeller usually works only on a subset of the new requirements to be fulfilled by a OMOS model. Each of these evolved models, therefore, represents only a subset of the (desired) model which fulfils *all* the requirements. More than one modeller should hence be able to modify the same model at the same time. The advantage of working on the same model — as opposed to accomplishing the different tasks in separate models — is that existing model elements can be referenced, modified, or even deleted — whereas with separate models only referencing existing elements would be possible. Each task is usually different from the tasks of other modellers working on the same model at the same time. It follows, that, to accomplish a certain modelling task, each modeller should, therefore, be able to work independently from any other modellers working on the same OMOS model (i. e., be able to independently modify his/her version of this model). Furthermore, the collaboration approach should not require modelling tools to be connected via some kind of computer network. The necessity for software tools to have access to company networks is often difficult to justify (at large companies like Bosch) because of computer security issues and firewall restrictions. It follows that the most suitable approach for collaboratively working with OMOS models is, therefore, the *optimistic* collaboration approach discussed in Section 2.3.1.

Based on the findings from Chapter 2 and Chapter 3, the following principles and objectives of a parallel and, because of teams being split across different sites, distributed (also called multi-site) modelling approach for OMOS are defined: Following the established OMOS (model-based) development approach, each development iteration should be based on the OMOS model (or models) created in the previous iteration since this model provides the basis for realising new requirements. In order to allow several modellers to work in parallel on the same OMOS model, each modeller should be able to work on its own copy of this model. Since it provides the basis for the updated models (being worked on in parallel), it is called the *common ancestor model*. The updated models are called *evolved models*. Each evolved model realises a part of the requirements defined for the current development iteration. All evolved models combined realise all the iteration's requirements.

After the desired requirements have been realised, the evolved models have to be combined into one model again. This is necessary because the model is only complete, i. e., fulfils all requirements, after the parallel work of all modellers has been brought back together into a single model. This model then serves as the basis (i. e. common ancestor) for the next (collaborative and distributed) development iteration.

The process of combining both evolved models into one model is called *merging* (see Section 2.3.2). It is one of the main topics discussed by this research (see Section 1.2.2). The result of merging both evolved models is called the *merged model*.

Since evolved models are modified independently from each other, the same model elements might have been modified in different and potentially contradicting ways. For instance, an UML class called “A” in the common ancestor model might have been renamed “A1” in one evolved model and “A2” in another, or an element could have been modified in one evolved model but deleted in another. For instance, UML class “A” might have been deleted in one evolved model, but a new operation “foobar” may be added to “A” in the other model. These situations are called *model merge conflicts*. They usually cannot be solved automatically by the merge tool since it cannot decide which element version to use in the merged model [CW98]. Hence, modellers have to manually resolve conflicts (see Section 2.3.2).

The approaches to merging electronic documents in general and UML models in particular discussed in Section 2.3.2 merge two evolved models (originating from the same common ancestor model) at a time. While it would technically be feasible to merge more than two evolved models at the same time, however, this is likely to make the merge process too complex for users and prone to errors — especially when conflicts have to be resolved. Users would then have to reason about changes made to more than two evolved models in order to resolve conflicts. That is why the envisaged approach to merging OMOS models has to take only two evolved models (and their common ancestor) into account.

Given the above assumptions, the envisaged approach for working in parallel with OMOS models should be based the principles and objectives defined in the remainder of this section.

4.2.3. Meta-model-based difference and conflict detection and model merging

The merged model resulting from merging two evolved OMOS models has to be based on the same meta-model as the evolved models, and all other OMOS models, namely the UML meta-model (see Section 2.1.2). This guarantees that a merge model is an ordinary OMOS model (i. e., a UML class model and its class diagrams) and can be worked on with ordinary OMOS modelling tools.

4.2.3.1. State-based merging

The envisaged approach to OMOS merging should be based on the changes made to the two to-be-merged evolved model versions in comparison to their common ancestor model. Each model element's changes between both versions have to be determined and merged based on the element's state. This approach is called *state-based merging* [Fei91]. The states of the evolved models must not change during the process of merging them.

The approach for determining a OMOS model element's state is defined by the UML meta-model (see Section 2.1.2) and UML's meta-meta-model, called meta-object facility (MOF, see Section 2.1.2.1). The possible state of a OMOS model element is thus defined by the UML meta-model [Wes91, KWN05]. UML *properties* are the meta-model constructs used to define the *possible* state of instances of the respective meta-model elements. For instance, a class (i. e., an instance of the UML meta-model element *Class*) has a *name* attribute property (inherited from its super class *NamedElement* which is also defined by the UML meta-model) and an *ownedOperation* reference property (representing the operations belonging to a concrete class). Hence, when two versions of a class are compared (as part of the OMOS model merge process), the values of their *name* attribute properties and their *ownedOperation* reference properties have to be compared (among other properties).

4.2.3.2. Operation-based merging

In addition to state-based merging, there exists *operation-based merging* [KTW97]. It is based on the actual operations (user actions) which are performed by a modelling tool to modify a model according to the tool commands the user executed [RW98]. The operation-based approach requires the modelling tool to create an *edit log* of the user's operations. Therefore, in order to enable model comparison and merging, modelling tools have to be extended to support the creation of edit logs. In addition to the technical challenge of such a tool extension, there exists a legal one: it is not always desirable to record the exact user actions performed to modify a model. Especially in industrial environments edit logs might not be feasible because they might disclose intellectual property that companies might want to protect from, for example, customers which work collaboratively with the company on a model.

4.2.3.3. Identifier-based determination of equivalent model elements

Equivalent to-be-compared and to-be-merged model elements have to be matched by their identifier. As argued by Ohst et al. [OWK04], identifier-based matching of equivalent model elements if preferable to similarity-based matching because the latter might not detect certain model elements as equivalent if, for example, they were relocated to different container elements. However, the identifier-based approach has the arguable disadvantage that when model element are added, they are guaranteed to have distinct identifiers. When a structurally/semantically identical model element is added to both evolved models (and thus the different element versions have different identifiers), two distinct, but semantically identical model elements will, therefore, be added to the merged model.

However, the advantages of the identifier-based approach far outweigh its disadvantages — the disadvantages can, for instance, be mitigated by similarity analysis performed for added model elements (which is outside the scope of this research).

4.2.3.4. Consistent merged models

The state of model elements, i. e., the values of their properties can, of course, only be assigned according to the rules defined by the meta-model of the merged model. As further explained in Section 6.2.1, a fundamental rule inherent to all MOF-based meta-models is that each model element (except for the model's root element) belongs to a container element. Since this meta-model is based on MOF, the rules defined by MOF have to be adhered to by the merged model. Therefore, the process of deciding whether model element changes are conflicting is primarily driven by the rules defined by MOF [KWN05, Wes10]. The merge approach envisaged for OMOS models thus has to adhere to the fundamental syntax rules defined by MOF. However, more complex syntax rules defined by the UML meta-model in addition to the MOF rules have to be checked by additional tooling (which is outside the scope of this research). Such tools are usually part of the modelling tools used to create the UML models. For instance, UML modelling tools usually provide the possibility to validate a UML model based on the syntax rules defined by the UML standard (see Section 2.1.2).

4.2.3.5. No meta-model extension or profiling

During the model merge process, change and conflict information have to be associated with model elements. One possible way to achieve this is extending the UML meta-model by means of a UML profile (see Section 2.1.2.2) to add change and conflict information [LWWC11]. Extending the UML meta-model to better support merging is, however, likely to cause problems because the UML meta-model is complex, extending it in a way that does not contradict with the rules defined by the UML standard is only possible by means of UML profiles. However, using a profile to, for example, support for two classes containing the same operation at the same time requires extending the model merge tool to be able to deal with those operations. Therefore, the editing possibilities provided by standard, non-extended tooling do not work. In addition, model consistency rules defined by the UML standard cannot be applied to the extension. For instance, applying UML consistency rules for operations will not work for the operation brought into the merged model by the profile extension because the operation is outside of the scope of the consistency rules since it is not part of an ordinary class (due to the container class conflict). Another example of why UML extensions supporting model merging can contradict with the UML standard are single-value properties, i. e. attributes or references whose multiplicity has an upper bound of one. If such a property is modified in different ways in both evolved models, it is not possible to assign both values in the merged model because the property has an upper bound of one. For instance, a class's attribute can only have one type. When the type is conflictingly modified in both evolved models, it is not possible to assign both types to this attribute in the merged model.

The envisaged approach for merging OMOS models shall, therefore, use the standard UML meta-model and shall not apply a UML profile for annotating model elements with change and conflict information.

4.2.4. Automatic creation of the initial merged model

Following the optimistic collaboration approach suggested in this chapter for the to-be-developed model merge tool, the process of merging concurrently evolved models consists of two phases [CW98, OWK04]. In the first phase, the model merge tool automatically creates the initial merged model. Because of merge conflicts, this model may not be complete. Then, modellers have to solve these conflicts in the second phase of the merge process.

Regarding the envisaged OMOS model merge tool, the principles and objectives of the first phase (whose details are further discussed in Section 6.4) are as follows:

1. The merge process has to allow the merging of two evolved models (which are the result of two modellers independently modifying a common ancestor model in parallel). For the envisaged approach to merging OMOS models, this means that, in a first step, the OMOS models should be automatically merged. As shown in Chapter 3, OMOS models are created in a visual way using diagrams. Therefore, diagrams as well as the underlying OMOS models themselves have to be taken into account by the automatic merge approach, i. e., the underlying UML models and the OMOS (UML class) diagrams depicting the model elements should be merged (this step is discussed in detail in Section 6.4).

2. The merge process should detect all possible types of changes and conflicts. Therefore, a three-way merge approach has to be used (see Section 2.3.2). The

common ancestor model hence has to be taken into account in addition to both to-be-merged evolved models. Non-conflicting and conflicting changes of both evolved models have to be detected during the merge process. (This step is discussed in detail in Section 6.3.)

3. Changes made in both to-be-merged evolved models in comparison to their common ancestor have to be taken into account during the merge process. The changes of both evolved models should be automatically applied to the merged model. The merged model should thus represent the union of the changes of both evolved models (based on their common ancestor).

4. Because of merge conflicts, i. e. conflicting changes, fully automatically merging the evolved models might not be possible. Conflicting changes cannot be automatically merged. The merged model resulting from automatically merging evolved models is therefore called the *initial* merged model.

5. Conflicting changes should be automatically detected. Non-conflicting changes of an evolved model should be automatically applied (i. e., accepted) to the merged model. One could argue that only non-conflicting model elements should become part of the merged model. This would mean that model elements which had, for instance only a name conflict, would not be part of the initial merge model. Since the existence of (non-root) model elements depends on the existence of their container, all the children of these elements would, therefore, *not* be part of the merged model, too, no matter whether they had merge conflicts or not. This is too restrictive, therefore, a conflict of a property value that is not the model element's containment reference should not prevent the model element from becoming part of the merged model. If a concurrent modification conflict exists for an *attribute* property, the ancestor value shall be used in the merged model to ease manual conflict resolution for modellers. For instance, if the attribute property was the model element's name, it is helpful to modellers if at least the model element's name from the common ancestor is presented in the merged model during the conflict resolution phase. This approach is not useful for *reference* properties because the referenced model element from the common ancestor might not exist in the merged model (for instance, it may have been deleted from both evolved models).

Model and diagram conflicts cannot be resolved automatically, modellers have to resolve them manually. This is done as part of the conflict resolution process. The goal of this process is to solve all merge conflicts (in the merged model) and, thereby, create a merged OMOS model which does not contain any merge conflicts. The objectives and principles of the conflict resolution process are explained next.

4.2.5. The need for dedicated model merge tooling

The purpose of this section is to discuss how to cope with merge scenarios that may or may not be conflicting and, if they are conflicting, may thus need user interaction. This discussion is mainly related to the need of the model merge tool to automatically accept certain types of changes. The main issue with automatically accepting changes is that it is unfeasible to define a meta-model-based approach that is able to decide (1) which changes are not conflicting, and can thus be automatically accepted (and applied to the merged model), and (2) which changes are indeed conflicting, and thus need to be solved manually by modellers (as will be explained next).

For individual changes of single model element properties, it might seem sensible to automatically deal with those changes in most cases (provided they are not direct deletion-modification or concurrent-modification conflicts). For instance, individual attribute property value changes, like changing the name of an element, or reference property value changes, such as an element referencing new elements or not anymore referencing certain elements. When reasoning about those changes in isolation, it is obvious to conclude whether some change is conflicting or not: changing an element's name to "X" in one evolved model and to "Y" in the other are of course conflicting changes. However, determining conflicting changes becomes less obvious in a model-wide context. Should, for instance, adding (in one evolved model) a reference called "myA" to an element "A" be regarded as conflicting with "A" being renamed to "X" (in another evolved model)? It can be argued that these changes are *semantically* conflicting because the expectation that an element named "A" is being referenced is no longer met. However, there might be other situations where it is acceptable to change "A"'s name (to "X" in one evolved model) and reference it as "myA" (in another evolved model), and, thus, accept both changes in the merge model. Those simple examples already demonstrate that distinguishing between conflicting and non-conflicting changes is not a black or white, one-size-fits-all decision, but is something that has to be decided on a case-by-case basis by domain experts, not automatically by the merge tool. The merge tool can only detect structural conflicts based on the meta-model, however, it cannot detect semantic conflicts.

The examples above make it clear that, in isolation, most changes appear to be easily and automatically decidable, however, when reasoning about them in a broader, model-wide context they no longer appear to be so easily decidable. There are, of course, changes that are easy to decide upon: changing the very same element in different ways (in two model versions) is certainly a conflict or deleting an element in one version and modifying it in the other is also a conflict. However, there are other cases which are not as easy to decide and should thus not be dealt with purely automatically. There simply is no silver bullet. Hence modellers (domain experts) have to be involved in the merge process. A dedicated model merge tool is, therefore, required — it should guide modellers through the merge process by visually communicating how elements changed and how they are connected to each other to allow modellers to reason about changes and their impacts and ultimately decide which changes to accept or reject.

4.2.6. Dealing with model and diagram changes and merge conflicts

After the initial merged model has been automatically created, modellers have to manually solve merge conflicts (which shall be automatically detected when the initial merged is created). During the conflict resolution phase of the merge process, in order to resolve merge conflicts, modellers first of all have to understand why conflicts exist, i. e., they have to understand which and for what reason contradicting model element changes (made in both evolved models) are causing conflicts. Therefore, information about changes and conflicts have to be communicated to the modellers.

The merged OMOS model should be presented to modellers in a similar fashion as provided by the modelling tool used to create the (original) models. For the envisaged approach, and for merging UML models in general, this means that the original diagrammatic representation of models has to be used for the merge tool too. The OMOS model merge tool should provide a similar user experience as the modelling tool used to created OMOS models in the first place. To resolve merge conflicts, the merge tool should, therefore, allow modellers to work on the same level of abstraction and visualisation as the modelling tool used to create the original models provides. With respect to merging OMOS models, it follows that modellers should be able work with the merge model in a graphical way through OMOS diagrams and in a structural way by means of a model hierarchy tree (see Section 3.6).

Regarding conflict resolution, the envisaged approach to collaborative OMOS modelling has to be efficient in order to provide any advantages compared to the existing sequential single-model approach. In order to ease the conflict resolution process, model elements and diagram symbols should be annotated with change and conflict information to allow modellers to reason about non-conflicting and conflicting changes made in both evolved models (in comparison to the common ancestor model). The merge tool should communicate changes and conflicts to modellers by visualising them as part of the respective model elements and diagram symbols. As further discussed in Chapter 6, the visualisation should be done for the evolved models and in the common ancestor model, but not in the merged model, because, due to merge conflicts, the latter might not contain all model elements and is likely to change in the process of resolving merge conflicts. It follows that the ancestor model, both evolved models and the merged model have to be accessible to modellers during the conflict resolution process. These models have to be taken into account and have to be presented to modellers during the conflict resolution process.

In order for modellers to be able to efficiently reason about conflicts and solve them, conflicts have to be presented to the modellers in a way that is familiar to them from the model creation process which took place before the models were merged. Most importantly, conflicts have to be presented in the original diagrammatic modelling context. Conflicts exist because the to-be-merged evolved models were concurrently modified in conflicting ways. In order for modellers to solve conflicts, the conflicting modifications have to be communicated to them so they can reason about possible ways to solve the conflicts. These modifications therefore have to be brought to the modellers' attention during the conflict resolution phase. Resolving merge conflicts then means that modellers have to decide which of the conflicting changes are applied to the merged model. For instance, if an operation was moved to different (container) classes in both evolved models, the modeller has to decide to which class the operation belongs in the merged model. Modellers could choose between one of the two (container) classes used in the evolved models. However, they might also decide to leave the operation at the (container) class it belonged to in the common ancestor model (in case the container class still exists in the merged model), or they might even choose for the operation to be contained by a different class (i. e., a class which was *not* the operation's container neither in any of the evolved models nor in the common ancestor model).

It is, therefore, not sufficient to provide, for instance, a textual description of changes and conflicts, and let modellers resolve them by means of an ordinary UML modelling tool — i. e., the kind of "collaborative modelling" approach implemented by most commercial UML modelling tools (see Section 2.4). Instead, dedicated tooling is required which supports annotating model and diagram elements with change and conflict information. In addition to visualising change and conflict information in an appropriate graphical manner in the actual diagrammatic modelling context modellers are familiar with, modellers have to be supported by a merge tool during the conflict resolution process.

4.2.6.1. Standard model visualisation

Adding a model element twice in a model and/or a diagram to visualise changes and conflicts as suggested by other research (for instance, [LWWC11]) discussed in 4.2.3 raises the question of how to present its sub-elements. For instance, if a package was conflictingly moved to other container packages in both evolved models, then the moved package will have to be shown at both of its container packages. Should the packages sub-elements, for instance, its classes and sub-packages, be displayed for both variants of the package, which would mean that they are displayed twice? What if a sub-element was added or removed in one of the evolved models, is it then only displayed or not displayed as a sub-element of the respective variant of the package resulting from the respective evolved model? It can even be argued that the relocated package should also be displayed as part of the package it belonged to in the common ancestor version (and from which it was moved away in both evolved models). Doing so would display the same package at three different container packages. Another question then is what if the old container package in the common ancestor was (non-conflictingly) deleted from both evolved models, should the deleted element become part of the merged model again in order to display the sub-package? The same question holds for changes of the package itself. For instance, the package could be renamed in the evolved models. Which name is used for displaying the variants of the package? All those unanswered questions demonstrate that visualising (conflicting) model modifications in the merged model does not work because, in order for it to do so, elements from four different models (the common ancestor, the two to-be-merged models, and the merged model with conflicts) would have to be presented as a single model element. Furthermore, modifications from two evolved models would have to be visualised for this one element too. Therefore, a different approach to visualise and deal with merge conflicts and the involved models has to be used. This approach should work as follows:

All four models (the common ancestor, the two to-be-merged models, and the merged model) involved in the merge processes should be part of the conflict resolution process. They should be presented to modellers in the same visual way they are familiar with from creating OMOS models, i. e, as OMOS diagrams and model trees. The common ancestor and the two to-be-merged models should be used to visualise model changes and conflicts, therefore, model elements should be visually annotated with change and conflict information. Providing access to all three original models is beneficial because it allows modellers to inspect the complete models. The common ancestor model can be investigated to reason about the "old" state of the model, both evolved models can be investigated to understand how both models changed in comparison to the common ancestor and which changes are conflicting. Because the common ancestor model is part of the conflict resolution phase, model elements which were deleted in one or both evolved models can nonetheless be visualised (as part of the common ancestor model) and annotated with information about the deletion changes enabling modellers to learn about model element deletions.

As defined in Section 4.2.3, the models should not be extended in any way, they should be ordinary UML models. In order to resolve conflicts, the modeller should only be allowed to modify the merged model, the common ancestor and the two to-be-merged models must not be modified.

The same approach should be taken for visualising changes and conflicts of diagram symbols: All three original OMOS diagrams (i. e., the common ancestor and the two evolved diagrams) and the merged diagram have to be presented to the modellers in the OMOS merge tool. The three original diagram versions (of every OMOS diagram) should be annotated with change and conflict information. However, the merged diagram should not be annotated with such information.

Since the information presented to modellers is taken from the models, this information is based on the models' state, i. e., the values of their elements' properties. Since all change and conflict information (regarding model/diagram merging) are based on the state too (see Section 4.2.3), it is possible for this information to become part of a model's or diagram's visualisation because change and conflict information are based on the properties of the model elements and diagram symbols.

4.2.6.2. Visualising changes and conflicts in OMOS diagrams

All diagrams changes have to be communicated to modellers in order to allow them to reason about what changed in a diagram in both its evolved versions, why these changes are conflicting (if they are), and how to resolve the conflicts. As explained in the previous section, change and conflict information have to be visualised in the context of the changed model elements and diagram symbols directly in the original diagrams, not in the merged diagram (and model) because the elements might not exist in the merged diagram (and model). Like the merged model's elements, the merged diagram's symbols should thus not be annotated with any change or conflict information.

The conflict resolution phase should, therefore, involve four versions of each diagram (provided the diagram was not added or deleted): the three original diagram versions (the common ancestor and the two evolved diagrams) and the merged diagram. The ancestor diagram of course depicts the diagram as it was before the two evolved diagram version were modified in parallel. Deleted symbols should be annotated in the ancestor diagram version because they do not exist in the evolved diagram(s) in which they were deleted. If the diagram symbol deletion is in conflict with changes made to the symbol in the other evolved diagram version, both symbols should then be annotated with conflict information.

4.2.6.3. Visualising UML model element changes and conflicts directly at the respective diagram symbols

Symbols of OMOS diagrams depict elements of the underlying UML model. Therefore, changes and conflicts of the underlying UML elements should be also visualised as part of the respective diagram symbols. It should be possible for modellers to visually distinguish between diagram symbol changes and conflicts and model element changes and conflicts. This allows modellers to get direct visual feedback for model element changes and conflicts directly in the diagrams depicting those model elements and allows them to distinguish between changes and conflicts affecting diagram symbols and those affecting UML model elements. For instance, in addition to presenting its diagram changes and conflicts, a class symbol should be visually annotated with information about changes and conflicts of its underlying UML class as well as the class's attributes and operations displayed as part of the class symbol.

4.2.6.4. Resolving merge conflicts and working with the merged model

Modellers have to resolve merge conflicts by either accepting or rejecting the respective model changes which caused the conflicts. Conflicts have to be resolved in the merged model. Modellers should not be required to modify the evolved models to resolve merge conflicts. Therefore, changes should by applied to the merged model, not to the evolved models in which the changes were made. The common ancestor model and both evolved models involved in the merge process have to be read-only models, they shall (only) be used to visualise the previous model versions (from which the merged model has been created) and their changes and conflicts.

The merge tool should automatically annotate resolved conflicts as “resolved” to visually indicate to modellers that the conflicts have already been resolved. Accepting or rejecting a change (from one evolved model) which conflicts with other changes from the other evolved model should automatically reject or accept the conflicting change. When modellers accept or reject conflicting changes, their resolution state should automatically become “resolved.” If modellers decide that none of the conflicting changes can be accepted, is has to be possible for modellers to manually set a conflict’s resolution state to “resolved”, both conflicting changes then have to be automatically rejected.

The conflict resolution process is mainly, but not only, based on accepting and rejecting conflicting changes. It should also be possible for modellers to accept or reject non-conflicting changes. This is necessary because modellers might want to reject a certain change and maybe accept it later again (as part of the same model merge process) in case the rejection turned out to be the wrong decision. Modellers might also want to reject changes because they are *semantically* (not syntactically) conflicting, i. e., a certain change does not reflect the desired semantics and should, therefore, be rejected.

4.2.6.5. Visualising changes and conflicts in a model tree

In addition to diagrammatic modelling capabilities, UML modelling tools usually provide some kind of model (hierarchy) tree which visualises a model’s (containment) hierarchy and provides a textual overview of the model (in addition to the information depicted in diagrams) in a condensed form. For example, the visibility of a class’s attributes is presented as a “+” or “-” sign in a diagram, but the model tree presents it as “visibility: public” or “visibility: private.” Since it displays all model elements, a model tree provides a consolidated view on the model. This is usually not the case for a diagram; it only depicts a sub-set of a model's elements. For example, the model tree presents all association and inheritance relations of a class, while a certain diagram usually only depicts some of these relations.

In addition to visualising changes and conflicts of OMOS model elements in diagrams, they should be presented in the respective (common ancestor or evolved model's) model tree. A model tree should be shown for each of the four models involved in the merge process (i. e., the common ancestor, the two evolved models, and the merged model). The model tree should visualise the elements of a model according to their containment hierarchy (starting with the root model element, going to packages, classes, attributes, operations and so on). The ancestor and evolved models' trees should display change and conflict information and allow modellers to deal with them (i. e., accept or reject them).

To allow modellers to understand the impact of accepting or rejecting changes, for each model element the model tree should visualise those model elements which reference it (by means of reference properties). Information of the referencing elements' changes and conflicts (if they exist) including their acceptance and resolution state should be visualised too.

4.2.6.6. Directly modifying the merged model and diagrams

Modellers should be able to modify the merged model and its diagrams in any way during the conflict resolution phase of the model merge process. It should be possible for modellers to create new model elements and diagram symbols and to modify or delete existing ones. This will allow modellers to resolve merge conflicts in a more flexible way than would be possible by merely accepting or rejecting conflicting changes.

4.2.7. Version control system

Realising the principles and objectives of the collaborative OMOS modelling approach defined in the previous sections does not require implementing a complete version control system (VCS). The principles and objectives rather outline the core building blocks of a VCS, i. e., differencing, merging, conflict detection and resolution. A collaboration approach can be based on a VCS, but this is not strictly required [CW98]. With a VCS, a new version of a document can be integrated into the shared repository at any time. This (modified) version then becomes the newest version in the repository. For collaborative approaches not based on a VCS, merging (supported by a merge tool) different document versions has to be initialised manually [Leb95]. The approach envisaged for working collaboratively with OMOS models should therefore be based on copying complete models (including their diagrams) between modellers in order to distribute them to allow modellers to work in parallel on (the copies) of the same models.

4.2.8. Summary

To summarise, the collaborative modelling approach envisaged for OMOS should be based on the following principles and objectives (and take the following issues into account):

1. Optimistic, merge-based collaboration approach: The approach for enabling parallel work on OMOS models has to be an optimistic approach, i. e., it has to allow more than one modeller to work one the same OMOS model in parallel and has to allow for merging the concurrently modified models (see Section 2.3). Because of the optimistic approach, copies of a model (including the diagrams representing it) can be modified independently by more than one modeller in parallel without affecting each other or limiting the possible ways of modifying a model. In order to arrive again at a single model which represents the combination of models which evolved in parallel, they have to be merged into this single model. The merge process should allow merging two evolved models (originating from the same common ancestor model) at a time. The merge process (as a result of which the merged model is created) should be performed automatically by a computer program, called model merge tool, to avoid the repetitive and error-prone task of manually creating the merged model from the two evolved models.
2. Automatic creation of the initial merged model and change and conflict detection: The merge process should be as automatic as possible. Thus, the initial merge model and its diagrams have to be created automatically. All non-conflicting changes of both to-be-merged evolved models should be automatically applied to the merged model. Only conflicting changes should have to be deal with manually by modellers. Merge conflicts can occur during this

merge process. They result from contradicting modifications made in the different, evolved versions of a model. Conflicts have to be detected automatically (when the initial merged model is created) and need to be dealt with manually by modellers since a computer program cannot always automatically resolve them in a meaningful way.

3. State-based merging: The envisaged OMOS merge model approach has to be a meta-model-based one. It should leverage the information defined by the meta-model of OMOS models, namely the UML meta-model, to detect changes and conflicts between OMOS models based on the state of the models' elements and diagram symbols.

4. During the merge process, modellers should be able to work with the merged OMOS model in a similar way they are familiar with from creating the model in the first place. This means that models should be presented to diagrammatic way using the same UML class diagram notation used when the models were created. Accepting and rejecting model and diagram changes and solving model and diagram merge conflicts should immediately update the merge model and diagrams so that modellers get direct visual feedback regarding the resolution of model merge conflicts.

5. When manually resolving merge conflicts during the conflict resolution process, modellers have to be supported by the model merge tool. Modellers should be able to resolve merge conflicts directly in the merged model. Merge conflicts should be resolved in the merged model by accepting and rejecting conflicting changes. The merge process should support accepting or rejecting conflicting changes (in order to solve merge conflicts) and should also allow assigning arbitrary values to conflicting, non-conflicting, and unchanged properties, i. e., for properties of any model element and diagram symbol. It should also be possible to create and delete arbitrary model elements and diagram symbols.

6. The model merge tool should allow modellers to exchange a partially merged model with other modellers. This will allow a team member to resolve some of the merge conflicts and leave other conflicts to be resolved by others. It should therefore be possible to save changes and their acceptance state as well as conflicts and their resolution state and, of course, the merged model itself including all the already (automatically and manually) merged changes.

4.3. Principles drawn from the pilot study to provide a software-based solution for working in parallel with OMOS models

This section defines the principles of a software-based solution (i. e., a merge tool) for working in parallel with OMOS models which are specifically drawn from the pilot study on semantic meaning conveyed through the layout of diagrams (see Section 3.6).

4.3.1. The need for visually merging OMOS diagrams

As discussed in Section 2.4.2, existing UML modelling tools (which were analysed as part of this research regarding tool support for parallel modelling) provide solution methods for merging only UML models, but do not consider (UML class) diagrams as first class citizens. While these tools support modellers with reason about changes and conflicts made to UML *model* elements, they ignore (the) diagrams (visualising these model elements). It is argued that these approaches do not support efficient merging of visual models. If one accepts the argument that graphical representations of models offer advantages to developers, in terms of understandability and so on, then it follows that tools should allow for the merging of such models based on their graphical representation.

4.3.1.1. Three-way diagram merging

When two evolved versions of a OMOS model are merged, all diagrams belonging to both evolved models have to be merged as well. Therefore both evolved versions of each diagram have to be combined. The resulting *merged diagram* should apply all changes made in both evolved diagrams (in comparison to their common ancestor diagram). Similar to the approach used for merging evolved versions of a UML model, a three-way merge approach should be used to merge two evolved versions of a OMOS diagram. As for merging UML model elements, OMOS diagrams should be merged based on their state and state changes (see Section 4.2.3).

Since the evolved models were independently modified by different modellers, both evolved diagrams were also modified independently. As discussed in Section 3.4, OMOS diagrams are manually created free-form (UML class) diagrams. Hence, the content and layout of the same diagram can be changed independently for both evolved model versions. Even if an evolved diagram was not modified manually, its layout might have changed because, for instance, the size of class symbols change because its visualised details (name, operations, attributes, stereotypes etc.) changed due to modifications of the underlying UML model (these UML model changes might have been initiated in other diagrams) or because of diagram symbol deletions as a consequence of deleting their underlying UML model elements. That is why the merged diagram has to be synthesised based on the changes of both evolved diagrams (in comparison to their common ancestor diagram).

4.3.1.2. Diagram merge conflicts

Because of the independent evolution of a diagram, *diagram merge conflicts* can occur when two evolved versions of a OMOS diagram are merged. They are similar to *model* merge conflicts which can occur when two evolved versions of a OMOS model (underlying the OMOS diagrams, see Section 4.2.2) are merged. As for model merge conflicts, these conflicts occur if the same diagram symbol has been modified in different ways in both evolved diagram versions in comparison to their common ancestor diagram. Like model merge conflicts, diagram merge conflicts cannot be dealt with automatically. Therefore, conflicting diagram symbols cannot be automatically integrated into the merged diagram when the initial merged diagram is automatically created. The conflicts have to be dealt with manually, modellers have to resolve them during the OMOS merge process.

That is why the merged diagram that is automatically generated by the model merge tool is called the *initial* merged diagram. It is called “initial” because, due to diagram merge conflicts, it might not contain all diagram symbols.

Whilst diagram merge conflicts are discussed in Section 4.3.2, there is another kind of conflict that has to be taken into account during diagram merging: overlapping diagram symbols. When two evolved diagrams are merged by simply creating an overlay of both diagram’s symbols (as suggested by other research discussed in Section 2.4.1), symbols may overlap in the merged diagram because different symbols are placed at similar positions in both evolved diagrams. For instance, Fig. 4.1 shows a cluttered merged diagram with overlapping symbols (taken from an early, obsolete version of the diagram merge tool developed as part of this research).

As discussed in Chapter 3, OMOS models are built from a large number of class

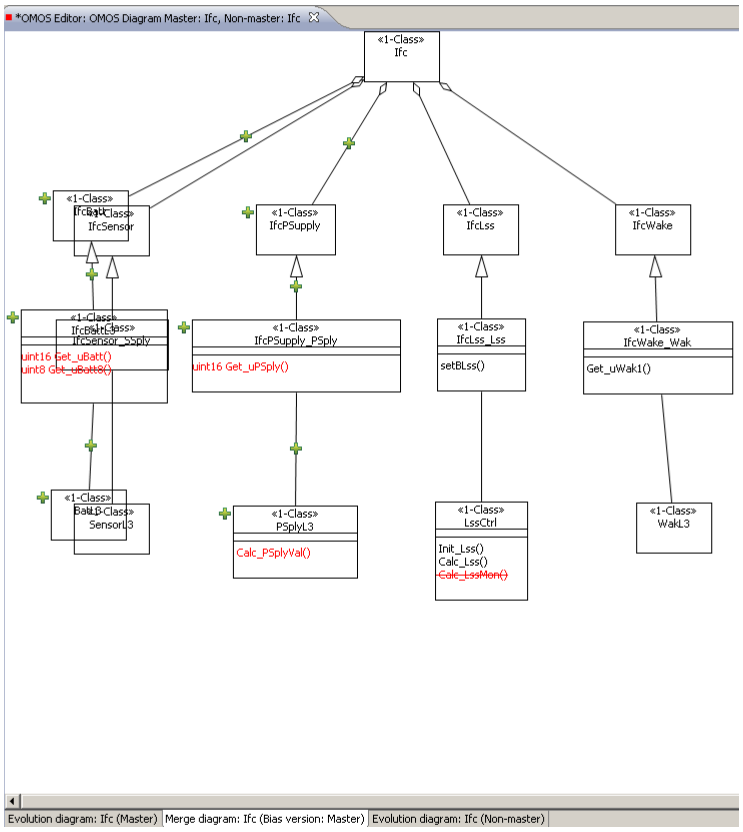


Figure 4.1.: Merged diagram with overlapping symbols.

diagrams (several dozens to about 140) and consist of hundreds of classes (and even more class symbols). Hence dealing with changes and conflicts has to be efficient in order for the collaborative approach to provide an advantage over the existing sequential single-model approach to OMOS modelling. Taking into account that models consist of more than one diagram thus becomes crucial when merging diagram versions that are produced in a distributed modelling process. Given that OMOS models consist of a large number of diagrams, burdening modellers with manually disentangling overlapping diagram symbols during the merge process is not an option. Merging OMOS diagrams by simply laying out both evolved diagrams on top of each other is therefore not a viable approach to merging OMOS diagrams. The large number of diagrams also rules out fully manual approaches to diagram merging burdening modellers with manually creating merged diagrams based on the evolved diagrams.

4.3.1.3. The necessity of automatic disentanglement of diagram symbols

To produce readable and usable merged diagrams, overlapping symbols have to be repositioned (moved) to disentangle them. For instance, if class symbols are moved in one evolved diagram version, and additional class symbols are added in close visual proximity of these class symbols (at their old positions) in the other evolved diagram version, the added symbols will not be positioned in close visual proximity in the merged diagram because the relocated class symbol will be positioned at its new position (taken from the evolved diagram version in which it was moved) and the added class symbols will be positioned at the (old) position (taken from the other evolved diagram version in which the diagram symbol was not moved).

Given that neither manual merging nor the simplistic "one diagram on top of the other" diagram merge approach is feasible, the only remaining option then is to automatically merge diagrams in a way that automatically yields disentangled diagrams. In order to disentangle overlapping diagram symbols, the layout of a merged diagram has to be *automatically* modified to allow for merging both evolved diagrams in an efficient and uncluttered manner.

Section 4.3.3 will provide a more detailed analysis of the issue of automatically rearranging the symbols of merged diagrams in a meaning-preserving manner.

4.3.2. Detecting and dealing with meaning-related diagram merge conflicts

Diagram merge conflicts occur if the same diagram symbol is modified in different ways in both evolved diagram versions. The most fundamental properties of diagram symbols are their position and size. If a symbol is moved by just one pixel in opposite directions in both evolved diagrams, a diagram merge conflicts occurs since a merge tool can no longer automatically decide which symbol position to choose for the merged diagram. Conflicts can also occur if the size of a symbol changes in different ways in both evolved diagrams.

Diagram merge conflicts would have to be resolved manually by modellers. It is, however, argued that this particular type of conflicts should not have to be solved manually. Taking the large number of OMOS diagrams constituting a OMOS model into account, it becomes clear that manually resolving such basic layout issues should not be burdened on modellers. Besides the number of diagrams there is another reason why modellers should not have to deal manually with such layout issues: a symbol's visual context does not change if the symbol is moved by a few pixels or if its size changes. As identified by the pilot study, modellers perceive the visual context of a class symbol as the meaning it conveys in a diagram. This context hence has to be taken into account for determining "real" diagram merge conflicts. Such conflicts occur if a class symbol's visual context was changed in both evolved diagrams in different ways. Based on the findings from the pilot study regarding addition of semantic meaning conveyed through the layout of diagrams, contradicting changes of a class symbol's visual context occur if modellers relocate it to a different group of class symbols or change the symbol's order within a group (see Section 3.5).

Like overlapping diagram symbols, which can appear when evolved OMOS diagrams are merged (discussed in 4.3.1), diagram merge conflicts for which a symbol's visual context does not change are not considered "real" diagram merge conflicts. Overlapping diagram symbols and layout conflicts related to contradicting symbol positions or sizes are therefore not referred to as diagram merge conflicts. Modellers should not have to deal with them manually. They should be resolved automatically.

The kind of diagram merge conflict that cannot be dealt with automatically are those for which a symbol's visual context has indeed changed in different ways in both evolved diagram versions. Modellers have to manually resolve these conflicts since they reflect changes in the semantic meaning conveyed through the layout of the respective class symbols. The envisaged approach for merging OMOS diagrams hence has to detect and differentiate between automatically resolvable layout issues (overlapping symbols and layout changes which do not alter a class symbol's visual context) and real diagram merge conflicts resulting from changing the visual context of class symbols.

Ideally, diagram symbol modifications that lead to irrelevant diagram conflicts could be avoided in the original (to-be-merged) evolved diagrams — possibly by allowing modellers to explicitly express meaning in diagrams, not jut through their layout by means of secondary notation. Such an approach would allow for automatically detecting diagram modifications which alter the visual context of class symbols (besides from adding or removing class and connection symbols) and would, therefore, ease the (diagram) merge process.

4.3.3. Meaning-preserving diagram merging

As identified by the pilot study in Chapter 3, diagrams are of crucial importance for the OMOS approach because OMOS models are constructed by creating class diagrams which convey additional (but informal) meaning through their layout. As demonstrated by the pilot study, OMOS modellers make use of grouping and ordering of class symbols to convey semantic meaning through the layout of OMOS diagrams. The visual context of a class symbol is chosen by modellers to convey this additional semantic meaning through the diagram's layout. The semantic information is important for modellers since it is part of their mental map of a OMOS diagram (see Section 3.2.4). It follows that groupings and orderings of class symbols have to be preserved when symbols are repositioned as part of an automatic OMOS diagram merge process (as asked for in the previous section). Therefore, when one class symbols is repositioned, the other class symbols that belong to the same *visual context* might have be repositioned as well to preserve this context and the visual relations of the context's class symbols.

Class symbols are the most important diagram symbols modellers focus on when creating diagrams, the process when the modeller's mental map of this diagram is established (see Section 3.2.4). They, therefore, are the most important diagram elements during diagram merge process, too, with respect to preserving both evolved diagrams' mental maps in the merged diagram. The layout (i. e. the visual contexts) of class symbols (of evolved diagrams), therefore, has be be preserved as much as possible when diagrams are merged.

When diagram symbols are moved from their original (evolved diagram's) position to a new position during the disentanglement, the mental map (see Section 3.2.4) of the diagram has to be taken into account. A diagram's mental map is important because (as stated by other research discussed in Section 3.2.4 and as confirmed by the pilot study's results provided in Section 3.6) the layout of OMOS diagrams conveys inherent semantic meaning. This semantic meaning must not be lost when diagrams are merged. However, this meaning is not formally defined by means of model or diagram data (i. e., model element or diagram symbol properties), it is only conveyed through a diagram's layout by means of secondary notation (see Section 3.2.3), i. e., by the symbols' visual context.

The layout of connection symbols (representing inheritance, composition, and associations relations) does not bear domain-specific information as the pilot study showed (see Section 3.6). The preservation of the layout of connections is, therefore, not strictly required when diagrams are merged.

4.3.4. Summary

To summarise, with respect to the findings drawn from the pilot study and from the research literature, the envisaged collaborative modelling approach for merging OMOS models should be based on the following principles and objectives:

1. The diagrams of the to-be-merged OMOS models have to be taken into account by any approach to visually merging OMOS models. It is not sufficient to only merge OMOS models without considering their graphical representations, i. e. the diagrams.
2. As identified by the pilot study (see Chapter 3), the following diagram symbols are used for OMOS diagrams: classes, inheritance, association and composition symbols (depicting the respective UML model elements)[23]. They have to be taken into account by any OMOS diagram merge approach. The layout of to-be-merged diagrams is important for modellers as it conveys domain meaning. The pilot study also identified that class symbols are those symbols which convey additional semantic meaning through their layout.
3. During the merge process, diagram merge conflicts (i. e., conflicts related to graphical properties of diagram symbols) have to be considered in addition to conflicts resulting from model modifications. However, modellers should only be required to resolve semantically meaningful diagram conflicts. Diagram merge conflicts should, therefore, only be brought to the modeller's attention if they are "real", meaningful conflicts, i. e., if the visual context of a class symbol was changed in contradicting ways in both evolved diagram versions. Modellers should not have to deal with basic layout issues (i. e., overlapping symbols and

[2]The envisaged approach to visually merging OMOS models does not have to deal with package assignment diagrams (discussed in Section 3.5). Automatic layout can be used for OMOS package assignment diagrams since their layout does not convey additional semantic information. That is why those diagrams are not taken into account for the layout approach for OMOS class diagrams discussed here.

[3]Even though comment/note symbols are used in OMOS diagrams, they shall be omitted from the diagram layout approach to be implemented as part of this research since they are very rarely used and most of the time represented "to do" notes for the modellers working on the diagram. Comment symbols would therefore usually be removed before the model is merged. Since comment symbols are rectangles, their shape is identical to class symbols. So the layout approach applied for class symbols can be applied to comment symbols as well.

layout changes which do not affect or alter the class symbols' visual contexts) which can be automatically resolved. Overlapping diagram symbols should be automatically untangled in merged diagrams, neither class nor connections symbols should overlap in merged diagrams. Such an approach will support efficient visual model merging because fundamental diagram conflicts caused by absolute position conflicts of diagram symbols (caused by free-form diagram layout) can be avoided [RW98].

4. Even if (some degree of) automatic layout is desirable (especially when diagrams are being merged as port of an optimistic collaborative modelling approach), fully automatic diagram layout as used by the approaches for working concurrently with UML model discussed in Section 2.4 has to be avoided since it is based on diagram layout aesthetics criteria which only take into account information from a UML model but do not take existing (manually defined) diagram layouts into account. Another disadvantage of those particular automatic diagram layout approaches is that when diagrams are merged and modellers resolve model merge conflicts, the UML model (underlying the merged diagrams) changes. Since the diagram layout aesthetics criteria are solely based on the UML model elements underlying the symbols represented in a diagram, changing these model elements can lead to diagram layout modifications. Because of the changed model elements, applying the same layout aesthetics criteria can lead to different diagram layouts. This leads to unstable diagrams layouts and destroys the additional semantic meaning conveyed through a diagram's layout and the modellers' mental map of the diagram. Therefore, criteria used for automatically laying out OMOS diagrams (during the merge process) have to take the layout of diagrams into account. It follows that the additional semantic information conveyed through the layout of the diagrams is then also taken into account.

5. OMOS diagrams should be merged as automatically as possible. However, the semantic information conveyed through the layout of the diagrams has to be preserved as much as possible when diagrams are merged. Since class symbols convey semantic information though their layout they play an important role for preserving a diagram's mental map. The meaning-conveying grouping and ordering of class symbols manually defined by modellers should only change when modellers manually change it, and should, however, not be changed automatically by a merge tool or when the underlying UML model is manually

changed by the modeller (for instance, because model changes are accepted or rejected during the merge process).

6. Even if diagrams do not convey additional semantic meaning through their layout, the mental map a modeller has established while creating or reading a diagram is important (for the modeller). Modellers should not be forced to relearn (and, therefore, rebuild their mental maps of) the layout of merged diagrams. A merged diagram's layout should be similar to the original evolved diagrams'.
7. In order to reason about model element changes and conflicts as well as diagram changes and conflicts, both evolved diagrams, the common ancestor diagram as well as the merged diagram shall be accessible by the user during the conflict resolution process.
8. To allow for convenient reasoning about changes and conflicts, they should be presented in a visual form directly in the respective evolved diagrams where these changes occurred in the first place. Model element changes and conflicts as well as diagram changes and conflicts should be visualised diagrammatically. This will allow modellers to reason about changes and conflicts in the same context they used to create the OMOS models and their diagrams.

4.4. Summary and outlook

This chapter focused on research objective 3 defined in 1.2.2: the principles and objectives were established on which any software should be based to provide a solution for working in parallel with OMOS models and their diagrams in general as well as the principles on which a solution has to be based on regarding the pilot study findings (see 3.6) in particular.

In the next chapter, an approach to laying out OMOS diagrams is presented that facilitates the preservation of semantic information conveyed through their layout when diagrams which evolved in parallel are merged.

5. Design and implementation of semi-automatic OMOS class diagram layout enabling efficient, meaning-preserving diagrammatic model merging

Whilst the approach for visually merging OMOS models (developed as part of this research) is presented in Chapter 6, a layout approach which allows for efficient merging of OMOS diagrams and supports the preservation of the meaning conveyed through their layout is provided in this chapter. This chapter (as well as Chapter 6), therefore, discusses the realisation of research objective 4 defined in Section 1.2.2: design and implement software to carry out a proof of concept.

5.1. Enabling meaning-preserving diagram merging through semi-automatic OMOS diagram layout by extending the visual vocabulary of OMOS diagrams

This section presents the design and implementation of a layout approach for OMOS diagrams which enables efficient diagram merging based on the principles defined in Section 4.3. A semi-automatic layout approach for OMOS diagrams is presented which allows modellers to manually define the grouping and ordering of class diagram symbols according to their understanding of the domain meaning represented by these symbols. As required by these principles, the approach allows the preservation of the semantic meaning conveyed through the layout of OMOS diagrams when the diagrams are merged.

5.1.1. Fully automatic layout destroys semantic meaning conveyed through the layout of OMOS diagram

With respect to diagram merging in general, the more automatically diagrams are laid out when they are created (as part of model creation process, not during the merge process), the better they can be merged. When diagrams are laid out fully automatically, *diagram* merge conflicts are avoided because modellers cannot influence the diagram layout at all (see Section 2.4). As required by the principles defined in Section 4.3, in order to allow for efficiently merging OMOS diagrams, modellers should not be required to manually disentangle overlapping symbols of merged diagrams. Therefore, automatic diagram layout may seem to be a viable solution for realising automatic diagram symbol disentanglement (as it takes care of laying out diagrams in a way that avoids overlapping symbols). However, as discussed Section 4.1, automatic algorithms lay out diagrams according to layout aesthetics criteria — an approach which often involves iterative layout and applying heuristics to create aesthetically pleasing layouts. Fully automatic layout (solely based on models), therefore, comes at the price of potentially destroying the original diagram layout (see Section 3.2.4) and, therefore, destroying the modellers' mental maps and the additional semantic information conveyed through the diagram's layout (see Section 3.2.3). For instance, adding or removing a connection symbol (representing a relationship between two classes) might place the (no longer) connected class symbols at different positions because the connection symbol "binds" the respective (now-connected or -disconnected) classes closer together according to the automatic layout algorithm's aesthetics-based layout rules.

Because of the domain-specific information inherent, but not formalised in OMOS diagrams, applying conventional UML class diagram layout approaches to automatically create OMOS diagrams and at the same time expecting stable diagram layout does not work. These approaches focus on creating aesthetically appealing diagram layouts in an automatic way, but, of course, cannot take into account domain-specific information which is not part of the OMOS model providing the information/elements to be depicted and laid out in a diagram. Laying out diagrams *fully* automatically (when they are created in the first place and during the merge process) is hence not feasible because conventional UML class diagram layout algorithms strive for aesthetically pleasing diagrams solely based on the model's structure and apply layout heuristics. Even worse, a OMOS diagram's secondary notation used to convey semantic meaning cannot be taken into account by fully automatic layout approaches because the semantic relations cannot be derived from the UML model because this information is not formally defined.

To summarise, with regards to laying out OMOS diagrams, fully automatic diagram layout has the following main drawbacks: Automatic layout algorithms are based on UML model elements, i. e., semantic elements like packages, classes, and inheritance, and association relations between classes. They cannot take domain-specific information into account because it is not part of the model. Hence, the automatic layout of OMOS diagrams would be solely based on UML elements and could not take into account domain-specific relations between classes. Laying out diagrams fully automatically is not a viable option for the envisaged approach to working in parallel with OMOS models because diagrams convey additional semantic meaning through their layout. OMOS diagrams contain layout constellations which reflect the domain-specific meaning of the model elements depicted in the diagrams. This meaning is not formally defined by means of data available in an OMOS model. It is rather something that exists in the OMOS developers' minds and, at best, in the textual documentation of the model elements. As identified by the pilot study, the grouping and ordering of class symbols conveys semantic meaning (for example, classes belonging to certain software layers or certain information/control flowing from one class to another). These informal layout features have to be preserved as much as possible when diagrams are merged (see the merge tool principles defined in Section 4.3). The diagram merge approach thus has to account for the fact that diagram symbols are laid out according to their domain meaning and the manually defined grouping and ordering of class symbols has to be preserved in merged diagrams.

However, applying some degree of automatic layout is useful when merging OMOS diagrams as will be explained in the next section.

5.1.2. The need for some degree of automatic OMOS diagram layout

Since the layout of OMOS diagrams conveys meaning, the layouts of both evolved diagrams (merged into the merged diagram) has to be preserved when OMOS models are merged in order to preserve the modellers' mental maps of the diagram and thus its additional semantic meaning conveyed through its layout in the automatically created merged diagram (see the merge tool principles defined in Section 4.3). The semantic meaning inherently conveyed through a diagram's layout is, however, not explicitly defined by means of "diagram data." Since diagram symbols can be freely laid out in a OMOS diagram without any layout restrictions, semantic meaning is only manifested in the position of class diagram symbols (see Section 3.4). Therefore, merging freely laid out diagrams in an automated way while still preserving the informal layout aspects which convey meaning (and are thus important for modellers) is not possible (see Section 4.1). An ideal solution would allow modellers to create diagrams the way they want with all possible layout freedom, but still allow *meaning-preserving* automatic layout (which would ease the diagram merge process). These two objectives are of course contradicting each other — layout freedom and automatic layout cannot be combined without limitations for one or the other (see also the discussion about dynamic graphs in Section 4.1.1).

However, as defined by the principles of OMOS diagram merging in Section 4.3, the layout of a merged diagram *has* to be *automatically* modified in order to combine both (to-be-merged) evolved diagrams in an efficient manner that, on one hand, preserves the class symbols' visual context and, on the other hand, allows the automatic rearrangement of symbols to merge both evolved diagrams in a meaningful way.

The diagram layout, therefore, has to be "understood" by the (diagram) merge tool. It has to extract information about meaning-conveying layout constellations (i. e., visual contexts) from the to-be-merged diagrams in order to preserve their semantic meaning in merged diagrams. As explained above, for free-form OMOS diagrams the only available diagram information hinting towards meaning-conveying class symbol constellations is the position of class symbols (see also the discussion on secondary notation in Section 3.2.3). However, defining a class symbol's visual context based only on the (adjacent) class symbols that are positioned in close visual proximity might not correctly capture its visual context and thus misinterpret its domain meaning as the following examples show.

OMOS diagram "Lss" discussed in Section 3.5.2 provides examples of possible misinterpretations regarding the visual context of class symbols: class "CL_OutPLssDuty" is visually closer to its associated class "CL_InpPIfcPSply_PSply" (association relationship) than to its whole-class "CL_OutPLssCtl_Mn" (whole-part relationship). However, the two class symbols connected by the whole-part relationship, define the classes visual context, because those classes are semantically related and this relation is semantically more important than the association relationship. On the other hand, in diagram "OutP_IFC_General" (Fig. 3.5) and "OutP_IFC_General" (Fig. 3.6), and in Fig. 4.1, visual contexts formed by the class symbols connected by association (and the generalisation) relationships are indeed important because the associated classes are semantically related.

It follows that, when related symbols (which have to be re-arranged together/as a group when one of these symbols has to be re-arranged in order to disentangle overlapping symbols in a way that preserves the symbols' visual context, i. e., their semantic relations) are automatically determined during the process of untangling overlapping symbols, relying solely on the symbols' position is very error-prone. The process of determining the visual context of class symbols, therefore, cannot be based solely on proximity and the connection symbols (relationships) between class symbols.

5.1.2.1. Semi-automatic layout based on explicitly defined semantic information to be conveyed through the layout of diagrams

As explained in the previous section, fully automatic layout is not suited for merging OMOS diagrams with respect to the preservation of both evolved diagrams' mental maps and domain-specific information conveyed through the diagrams' layout in the resulting merged diagram. Because of their drawbacks regarding the preservation of the mental map and semantic meaning conveyed in diagrams, conventional automatic diagram layout approaches do not provide appropriate solutions for merging OMOS diagrams. One of the challenges of this research therefore is to enable efficient model and *diagram* merging while still allowing the preservation of the domain-specific information inherent in OMOS diagram layouts.

Given the drawbacks of fully automatic diagram layout, but acknowledging that automatic layout is useful to some extent, and given that merging freely laid out diagrams in a meaningful way is not possible, a layout approach for OMOS diagrams is suggested (by this research) that allows modellers to explicitly define the visual context of class symbols, i. e., they can define the symbols' grouping and ordering according to their semantic relations. As demonstrated in the next section, if the layout approach is used for creating OMOS diagrams in the first place, they can be merged in an efficient manner as required by the principles regarding OMOS diagram merge process defined in Section 4.3. Despite using semi-automatic diagram layout, modellers are still able to arrange class symbols in a way that allows them to convey additional semantic information as the modellers used to do through the layout of freely laid out diagrams. This approach is discussed in the next section.

5.1.3. Meaning-preserving semi-automatic layout of OMOS diagrams

In this section, the main features of the semi-automatic layout approach are provided first, it is then explained how modellers can create OMOS diagrams and define the semantic meaning to be conveyed through the diagrams' layout.

A hybrid approach to laying out OMOS diagrams is proposed and implemented (by this research). It makes use of automatic layout features, but at the same time allows OMOS modellers to express domain-specific meaning through the layout of diagrams.

The *semi*-automatic layout approach presented in this section (and implemented as part of this research) allows modellers to explicitly define the grouping and ordering of class symbols. As discussed in Section 3.6 and Section 4.3, these two layout features were identified as the ones used by modellers to define and convey additional domain-specific meaning through the layout of OMOS diagrams.

The additional diagram information (on grouping and ordering of class symbols) is used to lay out OMOS diagrams in an automatic fashion. The information is leveraged to position class symbols according to the manually defined grouping and ordering. Thus, modellers are able to explicitly define the principle horizontal and vertical layout of class symbols. The diagram is then automatically laid out in a hierarchical, top-down manner, i. e. as trees.

For instance, Fig. 5.1 presents a OMOS diagram which has been laid out according to these rules: "C1" and "C3" are root class symbols, "C2" and "C4" are "C1"'s children class symbols, "C5" is the child class symbol of "C4", and "C6" is the child class symbol of "C3." The layout of this diagram was created in an automatic manner. However, the top-down (parent-child) and left-right order of class symbols has been manually defined.

The semi-automatic layout approach is used for both manually creating diagrams by OMOS modellers and automatically merging them as part of the OMOS models merge process. The reason why the layout approach is also used when diagrams are created in the first place is that this allows to lay out the original diagrams in the same way as the merged diagrams. Therefore, original and merged diagrams will look similar. Furthermore, when the semi-automatic approach is used, modellers can be relieved of manual diagram symbol re-arrangements which can be required in several diagrams even for a single model element modification since the element can appear in more than one OMOS diagram. With fully manual layout, modellers

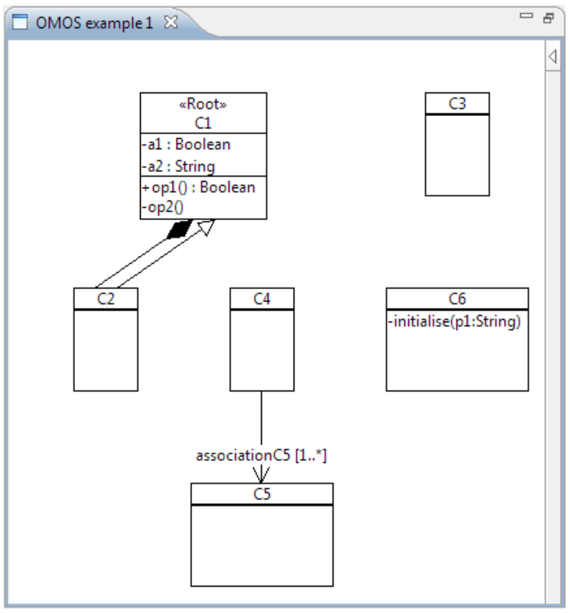


Figure 5.1.: OMOS diagram example 1.

would have to manually rework all these diagrams' layouts because, when modellers modify models and diagrams (as part of their normal modelling activities, not during the merge process), fully manual, free-form diagram layout requires them to manually rework several diagrams' layouts when model elements depicted in those diagrams are modified. This can cause considerable manually effort for modellers. With the suggested semi-automatic "tidying up" diagrams is done automatically — which results in less cluttered and more readable diagrams. Automatic layout can therefore help to keep diagrams readable and relieve modellers from having to manually disentangle diagram symbols.

However, the most important advantage of applying the semi-automatic layout approach when diagrams are created is that, since the semantic relations of class symbols are explicitly defined, readable *and* semantically meaningful diagram layouts can be automatically created. A certain degree of automatic layout is, therefore, desirable when diagrams are created as part of the modelling process (as well as for merging diagrams).

When modellers create OMOS diagrams using the suggested layout approach, they can only define the grouping and ordering of class symbols. This is the only layout information that can be defined manually. This limitation is made in order to facilitate efficient diagram merging. The more layout features modellers can manually influence, the more diagram merge conflicts can occur because (the values of) these features may have been conflictingly changed in parallel in different evolved diagram versions. These conflicts then would have to be manually resolved by modellers.

It can be argued that defining only the class symbol hierarchy in an explicit manner, in comparison to free-form layout, limits modellers with respect to diagram layout expressiveness. These limitation are discussed in Section 5.1.6; it is, however, argued that the suggested approach enables automatic diagram layout and merging without the drawbacks of fully automatic diagram layout (see Section 2.4.3) while still allowing OMOS modellers to express relevant (as identified by the pilot study in Section 3.6) semantic meaning through the layout of diagrams.

5.1.4. Semi-automatic OMOS diagram layout details

While its details are discussed in D.3, the main features of the semi-automatic layout approach for OMOS diagram are discussed in this section.

As required by the principles defined in Section 4.3, the suggested OMOS diagrams layout approach takes into account class symbols, association, composition and inheritance relationships[1]. All diagram symbols are drawn in black colour. With respect to a class's details, i. e., attributes and operations, depicted by class symbols, modellers can choose between three different options for displaying them: display all attributes and operations, display only public ones, or do not display any attributes and operations.

5.1.4.1. Manually established and automatically laid out parent-child class symbols hierarchy

Each OMOS diagram's class symbols are laid out in a top-down manner as a tree (parent-child) hierarchy according to the hierarchy manually defined by modellers. Each class symbol has one parent symbol — except for root, i. e. top-most, symbols which have no parent. When creating a OMOS diagram, modellers manually define the (top-down) parent-child hierarchy of the diagram's class symbols (see Fig. 5.1). When a class symbol is added to a diagram, the modeller defines the parent of a class symbol and its direct neighbours within the parent's (direct) child class symbols, i. e.,

[1]As defined in Section 4.3, OMOS package assignment diagrams do not convey semantic meaning. That is why packages and package assignment diagrams are not taken into account here.

the modeller defines the class symbol's visual context by defining its grouping and ordering. The actual layout of the class symbol (with respect to its position and size) is done automatically.

Class symbols are automatically laid out as a balanced top-down tree hierarchy, i. e., parent class symbols are centred above their child class symbols. The top of each class symbol belonging to the same "tree level" (or rank) is horizontally aligned. The "tree level" is referred to as the *layer* of the class symbol. Layers are not diagram elements, they are rather a concept for referring to all class symbols belonging to a certain tree level and to emphasize their horizontal alignment. Each class symbol is assigned to exactly one layer — root class symbols on the highest layer, followed by the layer to which the children class symbols of all root symbols belong. For instance, all root class symbols belong to the first layer, the direct children of *each* root symbol belong to the next lower layer, their children to the next lower layer and so on.

The horizontal position of a layer is defined by the largest class symbol of the layer above. This guarantees that class symbols on adjacent layers do not overlap (as requested in Section 4.3). Furthermore, a vertical gap guarantees that connection symbols do not overlap (see Section D.3). For instance, in Fig. 5.1, "C1" and "C3" are top-aligned on the first layer. "C1" is the highest class symbol in this layer and thus defines the layer's height. The class symbols of the second layer are again top-aligned at the top of this layer. Hence, "C3"'s child "C6" is top-aligned with "C1"'s children "C2" and "C4."

5.1.4.2. Manually defining the ordering of class symbols

As requested by the principles defined in Section 4.3, the grouping and ordering of a class symbol's direct children is manually defined by modellers to allow them to express additional semantic information through a diagram's layout. The order of class symbols does only change if modellers manually modify it. (In the diagram editor, the order can be altered by dragging a class symbol to the left or right border of another class symbol, or, if the desired parent class symbol has no children yet, by dragging the class symbol to the bottom border of the parent.) When a class symbol is relocated to a different parent class symbol or reordered within its current parent's children, all its (direct and indirect) children will be relocated as well; they "follow" their relocated parent class symbol and are, therefore, still be children of the relocated class symbol, furthermore, their order will not be altered when they are automatically relocated. They are positioned below the updated position of the actively relocated class symbol.

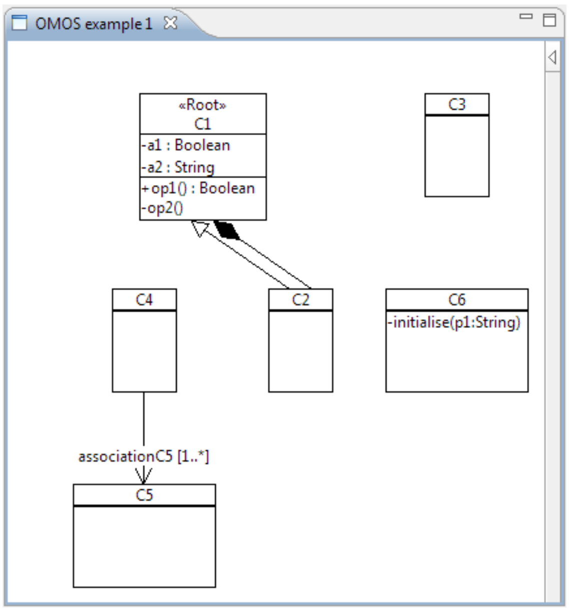


Figure 5.2.: OMOS diagram example 2.

For instance, in Fig. 5.2, “C1”’s children “C4” and “C2” have been reordered in comparison to Fig. 5.1. Hence, “C4”’s child class symbol “C5” was moved too. This approach ensures that the visual context of a class symbol is preserved as required by the principles defined in Section 4.3.

5.1.4.3. Connection symbols

This section discusses the layout of connection symbols. As defined by the principles in Section 4.3, connections should be laid out in a stable way: their layout should only change if the order of class symbols changes.

With the suggested layout approach, modellers only define that a certain association, composition or generalisation (model element) should be depicted in a diagram (as a connection between the two class symbols depicting the connected classes). The connection is then automatically routed. Modellers can neither influence where it connects to the class symbols nor any other layout aspect, for instance, bend points.

The connection symbols between class symbols are depicted as poly-lines. They are automatically laid out. If possible, connections are be depicted as short, straight lines. They may however have bends if required. Connections are not necessarily drawn in an orthogonal style.

The process of laying out connections consists of two steps: After the class symbols were laid out (see above), the connections are ordered according to their category (see below) in a first step. Then, the connections are laid out (as poly-lines). Both steps are discussed in detail in Section D.3. Connections are positioned in a way that arranges them around the centre of the respective class symbol's border.

The layout approach taken for a connection depends on which layer its source and target class symbols reside. The following connections categories are differentiated:

1. Inter-layer connections, i. e., connections between class symbols belonging to different layers
   a) between class symbols belonging to adjacent layers; or
   b) between class symbols belonging to non-adjacent layers.
2. Intra-layer connections, i. e., connections between class symbols belonging to the same layer
   a) between adjacent class symbols; or
   b) between non-adjacent class symbols.

5.1.4.3.1. Inter-layer connections are routed in a vertical manner since they connect class symbols whose vertical positions differ, i. e., one of the class symbols is aligned to a higher layer (closer to the top of the diagram) than the other. The former class symbol is called the source class symbol and the latter the target one. An inter-layer connection, therefore, connects the bottom of the source class symbol and the top of the target symbol.

5.1.4.3.2. Inter-layer connections between class symbols belonging to adjacent layers are drawn as straight lines of any angle. They connect to their source and target class symbol at the bottom and top, respectively. For example, in Fig. 5.1, all three connections are inter-layer ones connecting class symbols belonging to adjacent layers.

5.1.4.3.3. Inter-layer connections between class symbols belonging to non-adjacent layers are routed along the parent class symbols of the connection's target class symbol. They are poly-lines consisting of (1) straight vertical lines (called *parent-passing segments*) passing by at the side of the parent class symbols and (2) straight lines (*layer-connecting segments*) of any angle between the layers. For instance, in Fig. 5.3, the connection going from "C3" to "C5" is an inter-layer one connecting class symbols belonging to non-adjacent layers.

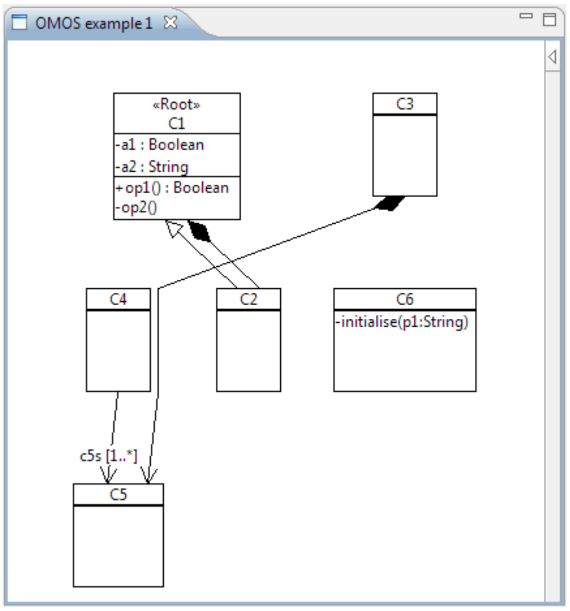


Figure 5.3.: OMOS diagram example 3.

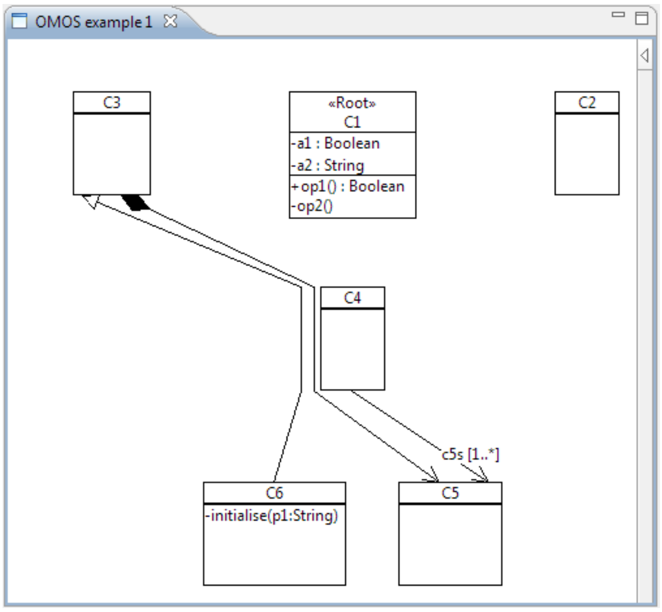


Figure 5.4.: OMOS diagram example 4.

The first segment of any non-adjacent inter-layer connection goes from its source class symbol to its target class symbol's (direct or indirect) parent — a layer-connecting segment. Then, a vertical segment follows which passes by this parent class symbol ("C4" in Fig. 5.3 and Fig. 5.4), this is a parent-passing segment. This type of segment does not directly connect to the class symbol, but ends on the left- or right-hand side, just next to the class symbol.

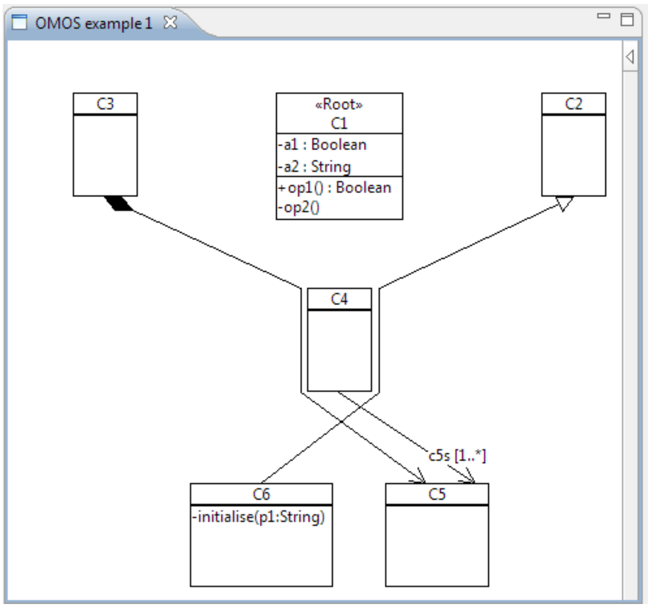


Figure 5.5.: OMOS diagram example 5.

On which side a parent-passing segment is drawn next to the parent depends on their source class symbol. For instance, in Fig. 5.5, the composition connection going from “C3” to “C5” passes “C4” on the left-hand side because “C3” (the connection’s source class) is left of “C4.” But the inheritance connection between “C2” and “C5” passes by “C4” on the right-hand side because “C2” (the connection’s source class) is right of “C4.”

As demonstrated by the association connection named “c5s” connecting “C4” and “C5” in Fig. 5.6, parent-passing segments passing by parent class symbols (“C6”) which are directly below the source class symbol (“C4”) are passed on the left-hand side.

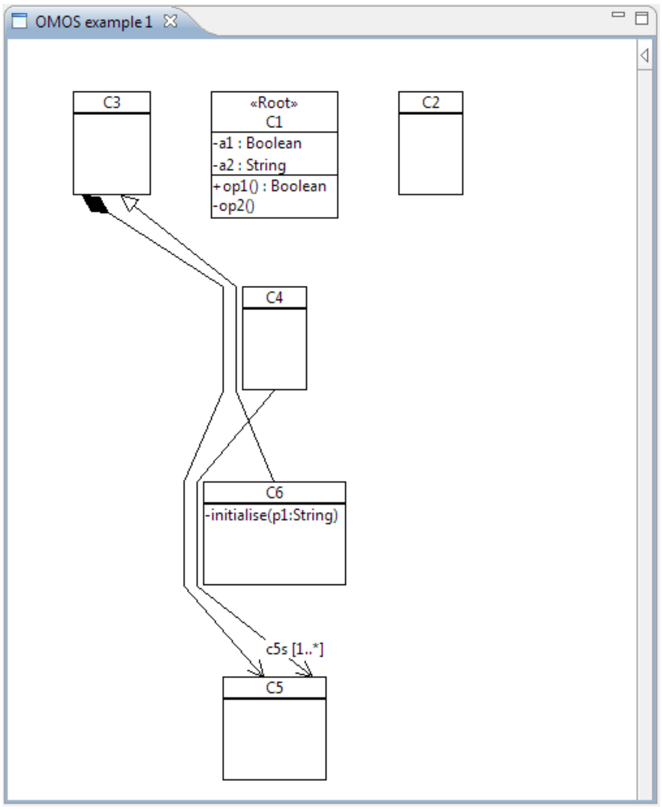


Figure 5.6.: OMOS diagram example 6.

If the connection's target class symbol belongs to the next lower layer, the approach for connecting inter-layer connections between class symbols on adjacent layers is used to draw a line segment from the parent to the target class symbol. If the connection's target class symbol does not belong on the next lower layer, a layer-connecting segment which goes to the next (direct or indirect) parent of the connection's target class symbol is drawn, and so on until the target class symbol (belonging to the next lower layer) is reached. For example, in Fig. 5.6, the composition connection connecting "C3" and "C5" (source and target class symbol) passes "C5"'s parent class symbols ("C4" and "C6").

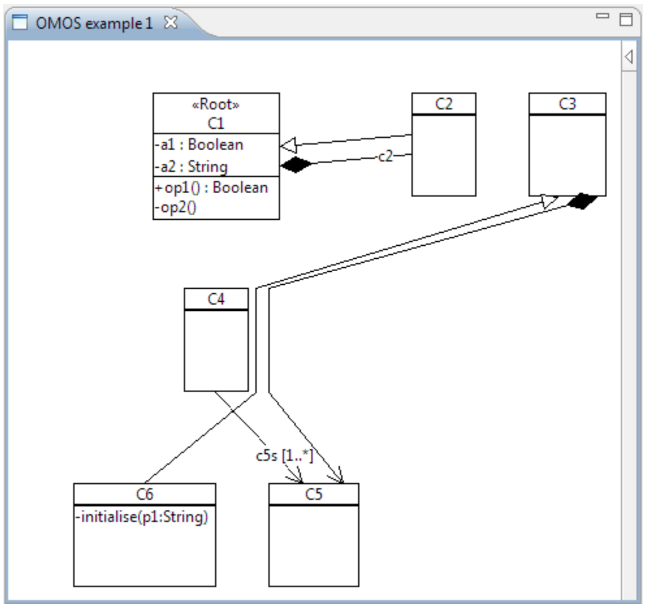


Figure 5.7.: OMOS diagram example 7.

As Fig. 5.7 demonstrates, a class symbol's parent-passing segments coming from the same source class symbol are ordered such that the inheritance connection connects closer to the centre of its source class symbol.

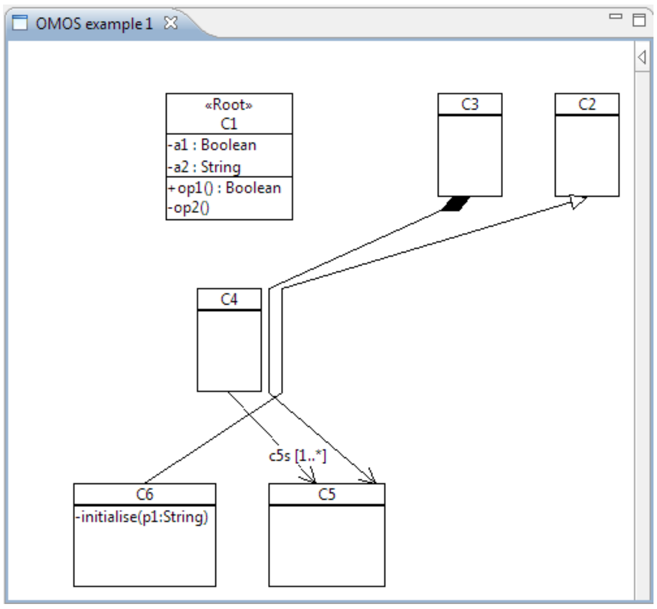


Figure 5.8.: OMOS diagram example 8.

As demonstrated in Fig. 5.8, parent-passing segments passing at the same side of a class symbol are ordered according to their source class symbol. Parent-passing segments coming from source class symbols further left will be positioned before those coming from further right.

5.1.4.3.4. Intra-layer connections between adjacent class symbols are drawn as a straight (but not necessarily orthogonal) lines connecting to the respective left-hand side class symbol (source) at its right side and attaching to its target class symbol (right) on its left side. For instance, the composition association and the generalisation connections between "C1" and "C2" in Fig. 5.9 are intra-layer connections between adjacent class symbols.

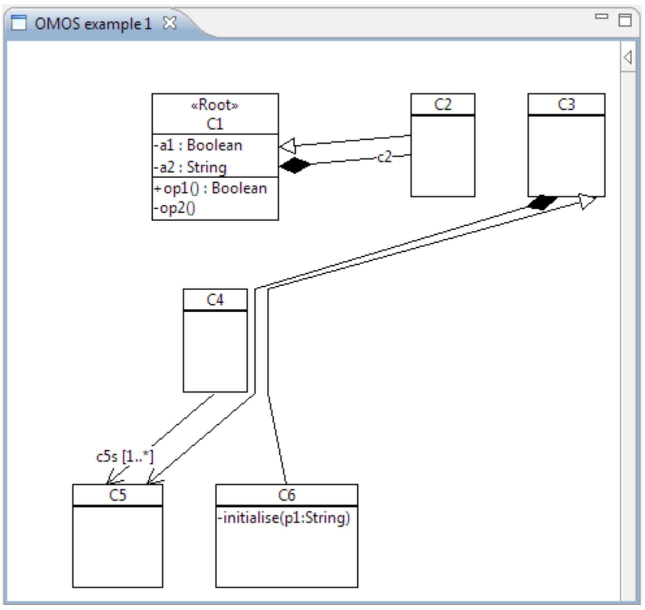


Figure 5.9.: OMOS diagram example 9.

5.1.4.3.5. Intra-layer connections between non-adjacent class symbols are drawn as three orthogonal lines segments as shown in Fig. 5.10. A connection's start and end segment are both vertical lines connecting to the top of the source and target class symbol, respectively. The middle segment is a horizontal line connecting the two vertical ones.

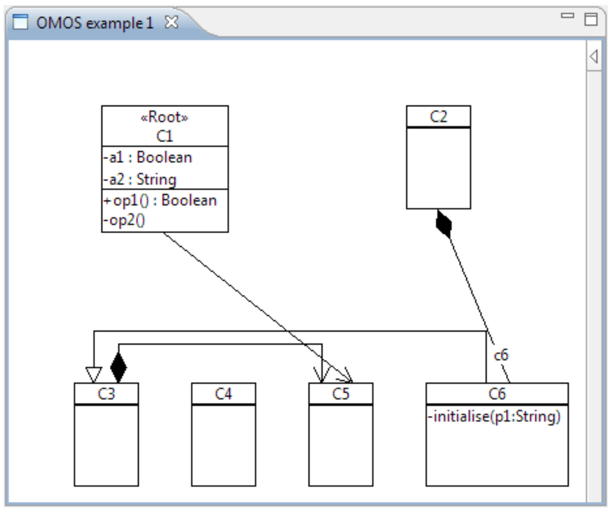


Figure 5.10.: OMOS diagram example 10.

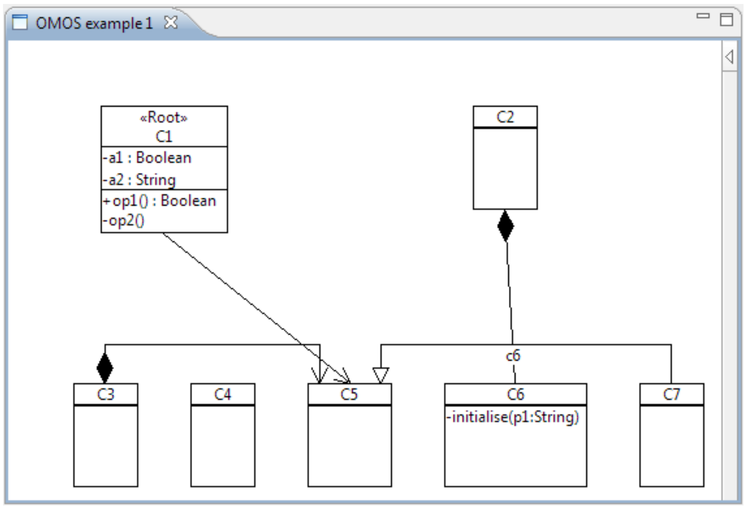


Figure 5.11.: OMOS diagram example 11.

The layout of intra-layer connections guarantees that such connections do not cross or overlap. This is achieved because a class symbol's connections coming from class symbols positioned further to the left connect to this class symbol before (i. e., further left of) interlayer connections and intra-layer connections going to non-adjacent class symbols further to the right. For instance, in Fig. 5.11, the composition association (intra-layer connection) from "C3" is positioned before (i. e, further left) of the inter-layer connection from "C1" and the intra-layer connection to "C7" (depicting an inheritance relationship). Overlapping of intra-layer connections (belonging to the same layer) are prevented by drawing each horizontal line on a separate horizontal position (see the two intra-layer connections in Fig. 5.10).

However, as Fig. 5.10 demonstrates, a limitation of this approach is that, if the same class symbol has intra- and inter-layer connections, this can lead to crossing connections. This issue is further discussed in Section 5.1.6. A possible solution is to draw inter-layer connections as orthogonal segments (as is done for intro-layer connections), but because of the additional space required to avoid overlapping orthogonal lines (see the two horizontal lines in Fig. 5.10), the height of the diagram would then increase.

5.1.5. Advantages of semi-automatic layout

Compared to conventional automatic UML class diagram layout approaches, the most import advantage of the semi-automatic layout approach is that modellers explicitly define the grouping and ordering of class symbols. This enables software tools to “understand” the diagram layout. Information about class symbol groups and their order as manually defined by modellers becomes a formally defined part of the diagram’s layout data. The explicitly defined class symbol hierarchy allows for the automatic rearrangement of diagram symbols (in order to untangle overlapping diagram symbols) in a meaning-preserving way. If this information was not available, semantic information conveyed through the layout of diagrams would be lost when diagrams are merged and a lot of manual work regarding the layout of diagrams would have to be burdened on modellers.

In order to automatically merge OMOS diagrams, the structure and relations of the diagram symbols have to be understood by the merge tool in order to be able to merge diagram symbols from both evolved diagrams and position them in the same visual context they belong to in their evolved diagram (if this context is not conflictingly defined for both to-be-merged diagram versions) in order to preserve the symbols’ semantic meaning (inherently conveyed through the diagram’s layout) and thus preserve the evolved diagrams’ mental maps in the merged diagram. The explicitly defined class symbol hierarchy can be taken into account when two evolved versions of a diagram are merged. The additional layout information allows the automatically creation of meaningful merged diagrams according to the class symbol grouping and ordering which was manually defined by modellers. The explicitly defined class symbol hierarchy allows automatically laying out diagram in a stable and meaning-preserving way. As required by the principles for merging the OMOS diagrams defined in Section 4.3, the layout of class symbols is manually defined by modellers and should change only if modellers change it. The manually defined diagram hierarchy allows the preservation of the diagram's layout during the diagram merge process if it was not changed in conflicting ways (diagram merge changes and conflicts will be discussed in Chapter 6).

This approach enables diagram symbols to become automatically re-arrangeable during diagram merging in a way that preserves the original grouping and ordering of diagram symbols. Being able to automatically re-arrange diagram symbols during the diagram merge process relieves modellers from having to deal with unimportant layout merge conflicts (for instance, symbol overlapping conflicts) and allows the automatic creation of uncluttered diagrams during the merge process. Modellers then only have to deal with real layout merge conflicts, i. e., conflicts of the diagram's mental-map-defining features which are manually defined by the modellers and cause merge conflicts if modified in conflicting ways.

The layout approach nevertheless takes into account and allows the creation of diagram layouts according to the UML layout suggestions for UML class diagrams. It, therefore, allows the creation of inheritance and whole-part hierarchies as well as association relationships between class symbols following the hierarchical layout suggested by UML.

5.1.6. Limitations and critique

The semi-automatic layout approach presented in this chapter focuses is on laying out the manually defined class symbol hierarchies in a balanced way. Doing so puts an emphasis on the parent-child hierarchy of class symbols and results in aesthetically pleasing and stable layouts of class and connection symbols. The layout of connection symbols is not done in such a balanced way; only the connection symbol's position at the source and target class symbol are balanced. Also, passing-by connections are not balanced. Depending on their source connection symbol, they are positioned at one side of passed-by class symbols. This can lead to unbalanced connection layout if several connections are routed along the same side of a class symbol.

The layout approach tries to avoid connection symbol crossings for certain constellations by ordering connections such that they cannot cross. For instance, intra-layer connection are positioned at the left or right side of inter-layer connections such that both do not interfere and, therefore, do not cross. However, because all layout decisions are taken *a priori*, connections are in, contrast to class symbols, not laid out in a balanced manner. A balanced approach could potentially yield more aesthetically pleasing layouts. For instance, “nicer” layouts might be achieved by allowing one connection symbol to pass on the left-hand side of a passed-by class symbol and the other on the right-hand side, even if both originate from the same class symbol. However, such decisions could then also depend on whether connections cross. As soon as such dynamism becomes part of the then iterative and possibly heuristics-based layout process, layouts are not as stable and predictable any more — although they might be more aesthetically pleasing.

With the suggested approach to connection symbol layout, the presence of orthogonally laid out connections (intra-layer connection symbols between non-adjacent class symbols) and other connections laid out as non-orthogonal straight lines (inter-layer connection symbols) at the same class symbol may lead to overlapping symbols. This can be avoided by drawing inter-layer connections in an orthogonal way; the first line segment leaving the source or target class symbol is then drawn as a vertical line, then a non-orthogonal straight line follows before a vertical segment finishes the connection by connecting to the source or target class symbol. A completely orthogonal approach for laying out connection symbols could also be used. However, doing so will potentially increase the height of diagrams because the horizontal gaps between layers have to get wider in order to avoid crossings of connection symbols.

The layout approach presented here is of course not suitable to expressing any kinds of layouts. However, it works well for the OMOS diagrams analysed in the pilot study in Chapter 3 as experts in the field (i. e., OMOS modellers; see Section 7.1) who evaluated the approach confirm. The details of the evaluation of the suggested approach are discussed in Chapter 7.

5.2. OMOS model editor

A OMOS model editor prototype was implemented as part of this research. An overview of the editor’s capabilities is provided in this section. Its design and implementation details are discussed further in Section D.2.

The editor allows the creation of OMOS models and diagrams. It implements the semi-automatic approach for laying out OMOS diagrams discussed previously in this chapter. The model editor allows the creation and manipulation of OMOS models in a visual manner. The editing capabilities provided by the editor are similar to those provided by other UML modelling tools. A unique feature of the editor, however, is that it allows the creation of OMOS diagrams using the semi-automatic layout approach discussed in this chapter.

The editor allows modellers to create OMOS models by means of OMOS diagrams. It also provides a model tree. This tree provides an overview of all model elements constituting a OMOS model and allows modellers to navigate to diagrams and diagram symbols which visualise the respective model element.

The editor allows modellers to create OMOS diagrams by explicitly defining the grouping and ordering of class symbols. Using the editor, modellers can, therefore, manually define the additional semantic meaning conveyed through the layout of class diagrams. This information can then be used to merge diagrams in a way that preserves the diagrams' additional semantic meaning.

5.2.1. Diagram and model updates and automatic re-layout of all affected diagrams

With respect to diagram editor tooling, diagrams have to be updated when the states of their diagram symbols are updated. For the OMOS model editor, symbol updates can occur because of the following reasons:

- Purely graphical updates affecting only a certain diagram but not the underlying UML model. Those updates are interactively initiated by the modeller by modifying the graphical properties of symbols of a certain diagram, for instance, relocating a class symbol. The updates only affect the symbols of the respective diagram but no other diagrams.
- Graphical updates which affect the underlying UML model and, thus, affect all diagrams depicting these model elements.
  - An end of a connection symbol is relocated (this update does affect the underlying UML model; so, moving a connection end to another class symbol relates the underlying model element(s) to a different class);
  - Deleting a class or connection symbol from a diagram and the underlying class or relationship from the model will affect all diagrams depicting the respective model element.

- When modellers use the model tree to directly modify the UML model, all diagrams depicting the modified model elements are automatically updated. For instance, when the parameter list of an operation is modified, all class symbols depicting this operation have to be updated because their widths may have changed. The width update, in turn, will require the respective diagrams to be newly laid out because changing the width of a class symbol requires rebalancing of the class symbol tree. This, in turn, might result in updating the position of class and connection symbols.

The ability to automatically update a diagram's layout when its symbols were updated (directly in the diagram editor and/or by altering the underlying UML model) and still preserve the layout features of this diagram (as requested by the OMOS diagram layout principles defined in Section 4.3) is one of the advantages of the approach to laying out OMOS diagrams developed by this research.

5.3. Chapter summary and outlook

This chapter presented a semi-automatic layout approach for OMOS diagrams. The appraoch was implemented as a proof of concept and represents one of the main contribution of this research. With this approach, modellers are able to define the horizontal and vertical order and grouping of class symbols as an ordered tree (parent-child hierarchy). The diagram is then automatically laid out based on the manually defined hierarchy. Class and connection symbols are automatically aligned according to this hierarchy. This chapter (together with Chapter 6) discusses the realisation of research objective 4 defined in Section 1.2.2.

The semi-automatic diagram layout approach presented in this chapter strives for creating stable and balanced class symbol hierarchies. Since the order of class symbols can only be changed manually by the modellers, a diagram's layout structure will only change if the order of class symbols is modified manually. Connection symbols are laid out fully automatically, modellers only manually define which relationships should be depicted in a diagram (as connection symbols). Connection symbols do not influence the layout of class symbols. Adding or deleting connection symbols, therefore, preserves the grouping and order of a diagram's class symbols.

The actual diagram layout does not rely on incremental layout and heuristics to create more aesthetically pleasing diagrams. All layout decisions are defined *a priori*,

i. e., there is a constant set of layout rules (see Section D.3) which is applied to a given diagram's graph (see Section D.7.1.1) in order to lay out its diagram symbols.

These two diagram layout features (manually defined parent-child class symbol hierarchy and the static layout rules) facilitate the preservation of a diagram's overall structure when it evolves and allow the preservation of the modellers' mental map (see Section 3.2.4) of a diagram. These features are, therefore, also important when diagrams which evolved in parallel are merged.

As discussed in Chapter 6, the parent-child class symbol hierarchy allows the automatic merging of diagram symbols and the production of meaningful and uncluttered merged diagrams. Futhermore, change and conflict detection during diagram merging and diagram merge conflict detection and resolution can be based on the explicitly defined visual contexts of class symbols (see Section 6.5.5).

The semi-automatic layout approach presented in this chapter facilitates the realisation of the requirements of meaning-preserving visual OMOS model merge approach defined in Section 4.3. The features and advantages of the semi-automatic layout approach presented in this chapter can be summarise as follows:

1. By extending the visual vocabulary of OMOS diagrams, the semi-automatic OMOS diagram layout enables efficient diagram merging by explicitly defining the grouping and ordering of class symbols to allow for meaning-preserving automatic layout.
2. Modellers have control over the grouping and ordering of class symbols, i. e., about the main properties that define the mental map of a OMOS diagram and convey semantic meaning. Class symbols are manually ordered on a per-diagram basis allowing the user to express domain-specific grouping of class symbols.
3. The semi-automatic layout approach is used both when diagrams are created in the first place and when diagrams are automatically merged. The manually defined information regrading the visual contexts of class symbols (explicitly defined by OMOS diagrams) are used during the automatic diagram merge process to preserve the semantic information in merged diagrams.
4. The layout approach strives for stable and predictable layouts. The overall structure of a diagram's symbols does not change when class or connection symbols are added or removed from a diagram. The order of connections between two class symbols is kept constant to support the preservation of the

diagram's mental map, i. e., adding or removing connections does not cause re-ordering of any other connections — not even between the two respective class symbols. Relocating a class symbol might, however, influence the order of connections because the order of connections is determined by the order of the class symbols they connect. The stability of a diagram's layout is increased by avoiding iterative layout and layout heuristics. The semi-automatic layout approach, therefore, facilitates the preservation of the diagram's mental map and, thus, the preservation of the semantic meaning conveyed through the layout of diagrams.

5. Connections are laid out fully automatically. Connections are ordered such that crossings and overlappings are avoided. However, no heuristics are used to minimise crossings because this could lead to unstable layout (for example, "jumping edges"). The number of bends of a connection is *not* minimised because this would require the usage of iterative layout and, possibly, heuristics which, again, could lead to unstable layout.

6. The model editor has the following advantages:

   a) Modellers can partition a model into any number of diagrams and, hence, create a hierarchy of diagrams depicting the various parts of a OMOS model as discussed in Section 3.6.1.

   b) Modeller can define the principle layout of each diagram. They can define which UML classes should be depicted in a particular diagram and in which visual context (i. e., define the respective class symbol's semantic relation). Modellers can choose for each class symbol whether all, only public, or no attributes and operations of the respective UML class are displayed.

   c) Furthermore, modellers can also define the UML relations to be depicted in a particular diagram. Adding or removing connection symbols from a diagram does not change the principle layout of the class symbols it connects; their visual context does, therefore, not change.

   d) Since the model editor uses the semi-automatic approach, the manually created OMOS diagrams are automatically laid out in the same way merged diagrams are laid out during the merge process.

The next chapter will discuss the design and implementation of a visual merge approach for OMOS models based on the semi-automatic diagram layout presented in this chapter. An evaluation of this diagram layout approach is then provided in Chapter 7.

6. Design and implementation of visual OMOS model merging

This chapter presents the approach for merging OMOS models, i. e., their diagrams and the underlying UML models. This chapter (together with Chapter 5) discusses the realisation of research objective 4 defined in Section 1.2.2, namely, to design and implement software to carry out a proof of concept.

6.1. Introduction

As defined in Section 4.2, the principal approach to collaborative modelling envisaged for OMOS is based on the following premisses: Software engineers can work independently and in parallel on copies of a OMOS model. This model is the common ancestor model which serves as the starting point for any modifications made in both copies. After the engineers have finished working on their versions the model, called evolved model versions, these models have to be brought back into one model again. This process is called merging. As a first step during the model merge process, the initial merged model is automatically created; it contains all non-conflicting modifications made to both evolved models' elements. Because both evolved models were modified independently from each other in parallel, contradicting modifications can occur; they cannot be automatically integrated into the merged model. These model merge conflicts have to be detected automatically and resolved manually during the merge process.

Based on the common ancestor OMOS model, two to-be-merged evolved versions (which were modified independently from each other at the same time) of this model are automatically merged to create the initial merged model. This first merge step is divided into two sub-steps. First, the differences between both evolved versions compared to their common ancestor are calculated. In a second step, the calculated differences are then applied to (a copy of) the common ancestor model. However, because of conflicting modifications in both evolved models, the merged model might not contain all modifications. Then, human interaction is required to solve these merge conflicts.

This chapter provides an overview of the design and implementation of the approach to visual OMOS model merging taken by this research. A detailed discussion on the implementing the OMOS model merger is provided in Appendix D.

6.2. Meta-model- and MOF-based model merging

Following the principles of merging OMOS models defined in Section 4.2, the merge and conflict resolution process developed as part of this research takes into account the UML class diagrams constituting a OMOS model as well as the underlying UML class model. Furthermore, the principles ask for merging OMOS models based on the state of the elements constituting the to-be-merged models. UML (class) models are formally defined by the UML meta-model discussed in Section 2.1.2. A meta-model defines the possible types of model elements that can exist in a model adhering to this meta-model and also defines the possible state of these model elements. As will be explained in detail below, state-based model merging approaches can make use of the information provided by a meta-model. This approach is taken by the OMOS model merger implemented as part of this research.

As explained in Section 2.1.3, OMOS models are ordinary UML models. OMOS is completely based on UML concepts, for instance, classes, operations, associations, generalisations etc. A OMOS model element is an instance of such a UML concept. As further described in Section D.5.1, the instance is defined by its state. The possible state of an instance depends on its type (i. e., the UML concept it represents). The UML meta-model, which is discussed in Section 2.1.1, formally defines all the concepts (types) and their allowed state. For instance, a class (i. e., an instance of UML concept *Class* defined by the UML meta-model) has a *name* attribute (inherited from its super class *NamedElement* which is also defined by the UML meta-model) and an *ownedOperation* reference (representing the operations belonging to this class). The *name* attribute holds the class's name state, for instance, "CarEngine", and the *ownedOperation* reference points to the class's operations, i. e., instances of UML concept *Operation* (also defined by the UML meta-model), for instance, operations "accelerate" and "break." A more detailed discussion of the part of the UML meta-model relevant for OMOS is provided in Section D.1.

As discussed in Section 2.1.2.1, the UML meta-model is formally defined by the meta-object facility (MOF). It follows that UML model elements are instances of MOF classes defined by the MOF-based UML meta-model. For instance, (the) UML (meta-model) defines a class called *Operation* (an instance of MOF class *Class*) and its properties (instances of MOF class *Property*), for example, *name* (attribute property) and *ownedParameters* (reference property). A meta-model hence defines the types of the model elements (instances of the meta-model) available in models which adhere to this meta-model. A type's properties and their *actual* features (for instance, name, ordering, uniqueness or upper/lower bound) are defined by this meta-model too. The *possible* features of properties are defined by MOF, the meta-model's meta-model. The properties then "come to life" as part of model elements, for which the property values define a model element's state.

The findings above show that, in order to create a state-based model merging approach, the to-be-merged model's meta-model and the meta-meta-model provide important information for determining the actual state of the to-be-merged models' elements and for gaining information on how this state can change. The latter information is important for determining the modifications between to-be-merged models in order to apply those modification to create the merged model as well as to detect merge conflicts. Regarding model merging, meta-models provide information about the possible model elements which can exist in to-be-merged models as well as their state, and the meta-meta-model (i. e., MOF) provides information about the structure and state-defining features of a meta-model's elements and, thus, allows gaining insights aobut the possible state changes of actual model elements to be taken into account during the model merge process. All this information can be leveraged to automatically generate the initial merged model and to detect modifications and merge conflicts. Further information on the meta-meta-model aspects regarding model merging are provided in Section D.4, and the relevant meta-model aspects are discussed in Section D.6.

The UML meta-model formally defines the UML class models underlying the UML class diagrams visually representing OMOS models. However, the UML meta-model does not take into account their visual representation, i. e., diagrams; the UML standard does not define a meta-model for UML diagrams because such meta-models are considered to be tool-specific. In order to apply the meta-model-based merge approach used for merging UML models for merging OMOS diagrams, a meta-model for OMOS diagrams is defined as part of this research. As the UML meta-model, this meta-model is based on the meta-object facility. The details of OMOS diagram meta-model are provided in Section D.7, and details of the UML meta-model are provided in Section D.1.

Since the OMOS diagram meta-model is based on the meta-object facility, the same model merge approach (based on the state of model element properties defined by MOF-based meta-models) can be applied for merging OMOS diagrams as well as the underlying UML models. This approach is, therefore, taken by this research for merging the UML models and UML class diagrams constituting OMOS models.

6.2.1. Introduction to MOF-based modelling

Based on the introduction to UML provided in Section 2.1.2 and the meta-object facility (MOF) in Section 2.1.2.1, this section outlines the features of UML and MOF which are relevant for differencing and merging MOF-based models as used in the context of this research. The details of MOF-based modelling and model comparison are further discussed in Section D.4.

6.2.1.1. Models as graphs

As demonstrated in the previous section, models in general are structured documents since they are based on meta-models [Wes91, KWN05]. With respect to containment reference properties, UML models and all other models adhering to MOF-based meta-models represent *trees*. However, because of the non-containment reference properties, UML models and any other MOF-based models are not pure trees in the mathematical sense, they are networks of model elements, i. e., a *graph* [KWN05].

MOF-based models (i. e., instances of MOF-based meta-models) can be represented as graphs in the following way:

- Nodes (the model graph's nodes represent either model elements or attributes)
  - Nodes are typed (their possible types are defined by the model's meta-model). They are either
    - model elements whose types are defined by classes (instances of MOF element *Class*) in the meta-model; or
    - attributes whose types are instances of MOF class data type/attribute *Property* also defined by the meta-model.

    - Nodes can define relations to other nodes (see “edges” below).

- Edges (the model graph’s edges represent relations between nodes)
    - Two categories of edges exist: relations between (1) two model elements and (2) a model element and an attribute.
    - The following features are relevant for both categories: Ordered/unordered values, single-/multi-valued, and unique/non-unique values.
    - Edge category 1 (defining a relation between two model elements):
        - The relation type is defined by a meta-model reference (i. e., an instance of MOF class type/reference *Property*).
        - In addition to the relation features described above, two additional features can be defined for this type of relations: composition and uni-/bi-directionality.
    - Edge category 2 (defining a relation between a model element and an attribute):
        - Their types are defined by meta-model attributes (i. e, instance of MOF data type/attribute *Property*).
        - The composition and bi-directionality features (mentioned above for edge category 1) are irrelevant for this kind of relations because attribute values always belong to a certain model element (composition) and, because their values are unstructured, cannot reference the owning model element.

Further details of the meta-object facility (MOF) and MOF-based model merging are provided in Section D.4.

6.3. Overview of the model comparison process OMOS models

In order to merge OMOS models, the differences between the two evolved (to-be-merged) models and their diagrams in comparison to their common ancestor have to be determined. Taking the common ancestor model into account enables three-way merging which allows to detect and handle more types of changes and conflicts than a two-way merge approach (see Section 2.3.2). An introduction to model comparison in general is provided in this section, the design and implementation of comparing OMOS models and diagrams is discussed in detail in Section D.4.

As further explained in Section D.5.1, the merge process implemented by this research is based on the state of model elements. The determination of how two models changed thus requires that the differences between both models are calculated. Difference calculation (i. e., model comparison) is based on the state of the models' elements. In contrast to the *edit log* approach (to determining differences between two model versions) which lists all operations applied to a model (by a modeller by means of a modelling tool), the state-based model comparison is solely based on the current state of the compared model elements. Of course, the state of the to-be-merged models elements has been defined by modellers by means of a modelling tool. However, the modelling actions performed by the modeller in order to arrive at a model's current state are not part of the model and are, therefore, not known during the model merge process.

The calculation of differences between both evolved models in comparison to their common ancestor model works as follows:

1. Given the common ancestor model A and the evolved models B and C (which are the result of independently modifying copies of the common ancestor model), calculate

    a) all changes (change set ΔB) between (the state of the model elements of) common ancestor A and evolved model B, and

    b) all changes (change set ΔC) between (the state of the model elements of) common ancestor A and evolved model C.

2. Create the merged model by applying the modifications described by the change sets ΔB and ΔC to (a copy of the) common ancestor model.

Since the evolved models share a common ancestor model, the following principle types of changes can exist in an evolved model in comparison to the common ancestor [Wes91]: *modification*, *deletion*, and *addition changes*. The former two types of changes, modification and deletion, apply to elements that already exist in the common ancestor model. Addition changes apply to elements that are added to an evolved model. Modification changes refer to differences in the state of two versions of the same model element belonging to the two compared models.

The following principle types of chances can be detected when comparing two models (for instance, UML models or OMOS diagrams):

1. Existence change: A model element with a certain identifier exists in one model, but not in the other. When comparing an evolved model version with its ancestor, existence changes can be further specialised as addition or deletion changes because it is known that the ancestor model has been created before the evolved one.
2. State change: The state of two versions of a model element with a certain identifier (i. e., equivalent elements from the two different models) differs.

Additional information about model changes and a meta-model for describing them (in the form of models) developed as part of this research are provided in Section D.5.4.2.

6.3.1. State-based model comparison

The process of comparing two models (i. e., an ancestor and an evolved model) consists of the following steps:

1. First, the model elements which are equivalent in both compared model versions are detected in order to determine which element pairs have to be compared. The equivalence of model elements is based on their identifiers as explained in Section 6.3.2.
2. After equivalent model elements were determined, added and deleted model elements (i. e., existence changes) are determined. Those are the model elements for which no equivalent elements exist in the other model version.
3. Finally, the values of the properties (defined by the meta-model for the respective element type) of equivalent model elements are compared to detect state changes.

Since each OMOS diagram represents an individual (diagram) model (based on the meta-model OMOS diagram meta-model defined in Section D.7.1.1), a set of changes is detected for each diagram. Therefore, the result of comparing two versions of a OMOS model is a set of UML model changes and a set of OMOS diagram change sets (one for each diagram model).

6.3.2. Detecting equivalent model elements based on their identifiers

According to the principles defined for the envisaged approach to working in parallel with OMOS models (see Chapter 4), the determination of equivalent model elements is be based on element identifiers. Further information on model element identification and matching equivalent model elements is provided in Section D.5.3.

6.3.3. Summary

The result of comparing an ancestor model with an evolved model is a set of changes. Before the actual merge process is initiated, the two change sets of the two to-be-merged evolved models are calculated by comparing both evolved models with their common ancestor model.

The model comparison approach explained in this section works for UML models as well as for diagram models or any other model whose meta-model is based on MOF. Since model elements are defined by their state, it is based on comparing the states of the elements of to-be-merged models.

A detailed discussion of comparing UML models and OMOS diagrams and the implementation developed as part of this research is provided in Section D.5 and Section D.7.

6.4. Overview of model merge process for OMOS models

This section provides an overview of how the initial merged OMOS model is created. Further details for the design and implementation of the model merging approach developed as part of this research are provided in Section D.6.

The model merge approach allows merging models which are instances of meta object facility (MOF)-based meta-models. This means that, like the model comparison approach, the merge approach is independent of the actual to-be-merged models. In the context of this research, it is not only used for merging UML class models but also for merging the OMOS diagrams visualising the elements of these UML models.

As required by the principles of the OMOS merge approach defined in Chapter 4, this approach has to ensure that the merged model only contains model elements which can be contained in a model according to the rules of the model's (MOF-based) meta-model. The reasons why model element cannot exist in a merged model are discussed below. Creating merged models which only contain “allowed” model elements according to their MOF-based meta-models has the advantage that all tooling infrastructure used to create such models in the first place can be leveraged in the merge process too. The model merge tool, therefore, is based on the same meta-models as used by the to-be-merged models. It is, therefore, not necessary to define and deal with meta-models different from those used for the to-be-merged models.

When the initial merged OMOS model is created (see below), in order for it to contain the changes of both evolved model versions, the changes made to the evolved models (in comparison to the common ancestor model version) have to be applied automatically to the initial merged model. However, conflicting changes can occur, they cannot be applied automatically. Therefore, the changes resulting from comparing both evolved model versions with their common ancestor have to be compared with each other in order to decide which changes are not conflicting each other and can thus be applied to the merged model and which changes cause model merge conflicts. This process is performed for the model changes of the to-be-merged UML models and the model changes of every OMOS diagram belonging to these UML models.

6.4.1. Accepting and rejecting changes, and detecting model merge conflicts

The objective of the conflict detection process is to detect changes made in parallel in both evolved models which contradict each other and thus cannot be applied to the merged model without further consideration by modellers. If changes are found (by the merge tool) to be non-conflicting, they are automatically *accepted* and the respective model element in the initial merged OMOS model will be changed accordingly (if it is part of it). As further discussed in Section D.6.3, depending on the type of conflict, conflicting changes are rejected or accepted. When a model merge conflict is created (by the merge tool as part of the process of creating the initially merged model), its resolution state is *unresolved.* Modellers have to deal with the conflicts and resolve them manually.

In order to detect model merge conflicts, the changes made in one evolved OMOS model are compared to the changes made in the other evolved OMOS model. The two change sets produced during model comparison are analysed for conflicting changes, i. e., for each change from one change set it is analysed whether it conflicts with changes made in the other change set.

First, model element addition and deletion changes are analysed (by the merge tool) in order to decided whether the addition or deletion can be accepted. Then, model element relocation changes are handled, and, thereafter, property value changes are checked for conflicts.

Based on the change types mentioned in Section 6.3 and defined in Section D.5.4.2, the following types of conflicts can be distinguished [Wes91]:

- *Concurrent property value change conflicts* indicate contradicting changes (made in both evolved models) of a model element's property value(s).
- *Concurrent model element relocation change conflicts* indicate that a model element was relocated in conflicting ways in both evolved models, i. e., it was moved to a new container element which is a different one for both evolved models.
- *Deletion-modification conflicts* occur when a certain model element was deleted in one evolved model and its properties' values were modified in the other one.
- *Existence conflicts*: A model element cannot be part of the merged model because, according to the rules of meta-object facility (MOF), it cannot be contained in its container element.

6.4.1.1. Summary

As a result of analysing the UML model and OMOS diagram changes for merge conflicts, every change is either accepted or rejected (called the acceptance state of a change). If it was rejected, the change conflicts with changes from the other evolved model. A merge conflict is thus created which references the conflicting model changes. The details of analysing changes and detecting UML model merge conflicts and OMOS diagram merge conflicts are discussed in Section D.6.3 and Section D.7.1.3.

6.4.2. Creating the initial merged OMOS model

Based on the UML model and OMOS diagram changes and their acceptance state, the initial merged UML model and its initial merged OMOS diagrams are created by applying the accepted changes to the ancestor version of the OMOS model. The creation of the initial merged UML model works in the same way as creating the model's initial merged OMOS diagrams. This is possible because the meta-model of the UML model and the diagrams are based on the meta-object facility, applying model changes thus works identically.

The input for the UML model merge process are the three OMOS model versions (the common ancestor and, if they exist, both evolved OMOS model versions), the model changes, and model merge conflicts.

The process of creating the initial merged UML model and the initial merged diagrams starts with the respective model's root element (which is the *Model* element for UML model and the *Diagram* element for each diagram) and first merges the values of the containment reference properties before the values of all other properties are merged. Merging property values is based on the acceptance state of property value changes, values with accepted changes or without changes are automatically merged, rejected changes are not automatically applied.

The resulting (initial) merged diagrams are instances of the graph model of a diagram (see Section D.7.1.1) describing the OMOS diagram's class symbol hierarchy. The merged diagram models are, however, not instances of the fully laid out version of a diagram model describing the OMOS diagram's actual layout by means of absolute symbol positions and sizes as used by the model editor to visualise OMOS diagrams. The fully laid out diagram model is derived from the hierarchical diagram model which, as explained in Section D.7.1.1, contains all information required for laying out the OMOS diagram.

A detailed discussion of the creation of the initial merged OMOS model as implemented by the research is provided in D.6.4.

6.4.3. Summary and outlook

The details of the model merge implementation realised as part of the research presented in this thesis are provided in Section D.6.

After the initial merged UML model and all initial merged OMOS diagrams have been automatically created, the second phase of the merge process starts in which modellers manually resolve model merge conflicts. This process's details are discussed in the next section.

6.5. Visualising and dealing with OMOS model and diagram changes and conflicts

After the initial merged model has been automatically created (as described in the previous section), it may not contain all modifications made to both to-be-merged evolved models. This may happen because of conflicting modifications (i. e., because of merge conflicts). Some types of merge conflicts, called existence conflicts (see Section 6.4), can even prevent model elements from becoming part of the initial merged model. There are other types of conflicts which do not prevent model elements from becoming part of the initial merged model (for instance, concurrent value change conflicts of attribute properties). All merge conflicts ultimately have to be solved manually by modellers. Resolving conflicts is the main objective of the second phase of the merge process. This section presents the approach (taken by this research) to supporting modellers with resolving UML model and diagram merge conflicts.

As required by the principles of the OMOS merge process defined in Chapter 4, the approach is based on the four OMOS models participating in the merge process, i. e., the common ancestor model, both evolved models, and the merged model. In addition, the both change sets and the set of conflicts derived from them are used as well (see Section 6.3). Each change set includes change information regarding the UML model and all diagrams belonging to the merged OMOS models (see Section D.7.1.2), and the conflict set contains information about model merge conflicts of the UML model and its OMOS diagrams (see Section 6.4).

6.5.1. A dedicated merge tool

The change and conflict information (i. e., the change sets and the conflict set) is not part of the models or diagrams themselves, this information is kept outside the actual models and diagrams. It is only accessed by the OMOS model merge tool which visualises models and diagrams together with change and conflict information and supports modellers to reason about model changes and resolve model merge conflicts. An extended version of the OMOS model editor which is used to create OMOS models and their diagrams in the first place (see Section 5.2) is, therefore, used to visualise change and conflict information of the diagram symbols and the underlying UML model elements, and to dealing with them (see Section 6.5.5). In addition to visualising models in a diagrammatic manner, the model editor provides a model tree which is annotated with change and conflict information.

The next sections discusses the approach to visualising and dealing with merge changes and conflicts as defined in Chapter 4 and implemented by this research.

6.5.2. Standard visualisation for all four model versions involved in the merge process

As requested in Chapter 4, the model merge conflict resolution approach implemented by this research takes into account all four OMOS models (the common ancestor, the two to-be-merged models, and the merged model). All four OMOS models are presented to modellers during the conflict resolution process. The models are visualised in the same diagrammatic way that is used to create OMOS models. All UML model and OMOS diagram changes and conflicts are visualised in the common ancestor and the two to-be-merged model versions. The merged model and its diagrams are not annotated with change or conflict information.

6.5.3. Visualising and dealing with model changes and model merge conflicts in OMOS diagrams

As discussed in the previous section, in order for modellers to efficiently reason about model merge conflicts and to solve them, the model merge tool presents changes and conflicts to modellers in the same way as used for creating the to-be-merged models. Chances and conflicts are presented in the original visual modelling context. For instance, if an operation was relocated to different (container) classes (in both evolved models), the operation is visualised as an operation of either class to allow modellers to see which container class the operation was relocated from and to. This is, of course, not possible in the merged model because, according to the UML meta-model, an operation can only belong to one class. There are two ways to solve this issue: Extend the UML meta-model to support merging or visualise conflicts in a different way which does not require conflicting model elements to be part of the merged model. As defined in Chapter 4, the merge approach implemented as part of this thesis takes the latter approach.

All diagram changes are communicated to modellers in order to enable them to learn about what changed in a diagram in both its evolved versions, why these changes are conflicting (if they are), and how to resolve the conflict.

Because the approach to merging OMOS diagrams can take the explicitly defined information regarding the diagrams' layout hierarchies into account, modellers have to deal only with real diagram changes and conflicts which relate to modifications of the visual contexts of class symbols (manually made by modellers in evolved diagram versions). As further explained in Section D.7.1, modellers do not have to deal with “pixel” changes and conflicts related to the modification of the absolute position or size of diagram symbols.

Changed symbols are highlighted in diagrams by means of colours, annotating a diagram symbol with diagram change and conflict information is achieved by painting the symbol’s border in a colour other than black (which is the colour used for diagram symbol borders). A yellow border indicates that the symbol was added to the diagram, grey indicated that it was deleted, blue means it was modified, and red indicates unresolved conflicts. If colours cannot be used, change and conflict information can be visualised as icons attached to the diagram symbols instead of colouring them. Displaying change and conflict information by means of icons will also help to overcome challenges colour-impaired modellers may experience with coloured diagram symbols.

When diagram symbols have changes which are not accepted in the merged diagram (i. e., their acceptance state is “rejected”), these symbols have a dotted-line border (instead of solid-line border). Diagram symbol changes which could not be accepted when the initial version of the diagram was created, are marked as conflicting. That is why their border is drawn as a red coloured dotted line. After the conflict has been resolved, the symbol’s border will not be coloured red any more (but in the diagram symbols’ default black colour).

As defined in Section 4.3, modellers are allowed to delete symbols from a merged diagram which do not have deletion changes. Doing so is an ordinary diagram editing action like it is used when a diagram is created (not during diagram merging). The deleted symbol’s border is then also painted in grey in any diagram of the original ancestor and evolved model versions depicting the deleted symbol’s UML class.

Annotating changes and conflicts by means of colour (or icons) provides direct visual feedback and allows modellers viewing a diagram to immediately understand which symbols have (conflictingly) changed — without the need to select the symbols in the diagram and to open a list (or similar) of changes and conflicts — and indicates to modellers which diagram symbols and model elements need further investigation. In case unresolved conflicting changes exist, modellers can immediately see that there are unresolved conflicts.

6.5.3.1. Visualising UML model changes and conflicts directly at the respective diagram symbols

As requested by the principles and objective defined for OMOS model merge tools in Section 4.2, changes and conflicts of the UML model element shall be directly visualised at the respective diagram symbols. Therefore, changes and conflicts of the underlying UML elements are also visualised as part of the diagram symbols depicting the UML elements. In order to distinguish between diagram symbol changes and conflicts and model element changes and conflicts, the latter are visualised by filling the background of the respective diagram symbol using the same colours defined above. This way, modellers get direct visual feedback for model element changes and conflicts directly in diagrams as well and can distinguish between changes and conflicts affecting diagram symbols and those affecting UML model elements. For instance, a class symbol visually annotates changes and conflicts of its underlying UML class as well as the class's attributes and operations if they are visible for this class symbol (depending on its details visibility level).

With respect to UML model element changes and conflicts, their respective diagram symbols do not provide all information available about its (conflicting) changes. Detailed change and conflict information are provided in the model tree which presents the respective model elements of the respective model version. This visualisation is discussed in the next section.

6.5.4. Visualising and dealing with changes and conflicts in OMOS model trees

The merge process's conflict resolution phase involves four versions the UML model (underlying the class definition diagrams of a OMOS model): The three original models (the common ancestor and the two evolved models) and the merged model.

In addition to visualising them in diagrams, changes and conflicts of UML model elements are presented in the respective model tree too. The model tree is also part of the OMOS model editor (implemented as part of this thesis) used to create OMOS models in the first place (see Section 5.2). A model tree visualises the elements of a UML model according to their containment hierarchy (starting with the root model element, going to packages, classes, attributes, operations and so on). It consists of tree items. Each item consists of a text labels and an optional icon. For the conflict resolution phase of the model merge process, the model trees are extended to display change and conflict information and to allow modellers to deal with them (see Section 6.5.5).

As for merging diagrams, there exist four model trees, one is depicting the common ancestor model's elements, another two are depicting each of the two evolved models' elements, and the fourth one visualises the merged model's elements. As for merged diagrams, the merged model's elements are not annotated with change and conflict information in the merged model's model tree. The ancestor model's model tree visualises the ancestor model's elements. It provides a view of the model as it was before the two evolved models were modified in parallel. Like the symbols of an ancestor diagram, the ancestor model's tree highlights model elements which are deleted in one or both evolved models. This model tree hence annotates model elements which have deletion changes and deletion conflicts in one or both evolved models. Deleted model elements are annotated in the ancestor model tree because they are not part of the evolved model in which they were deleted.

In addition to annotating model elements which have deletion changes, each model element's property values which have property value removed changes (see Section D.5.4.2) are annotated as well because the value is only available for the respective model element and the respective property in the ancestor model. If the respective property is a *reference* property (i. e., it references model elements), it also shows information about whether elements removed from the property were deleted from the model (or still exist and are just no longer referenced). If the property is a containment reference property, removing values means that the respective model element was either relocated to a different container element or deleted from the model. Information about both types of changes and, if they exist, conflicts are displayed for these removed values in the ancestor model tree too. If the property is an ordered one, reordered values are also annotated with property value reordered changes and, if a conflict exists, with conflict information in the ancestor model tree too. The reason for showing this information in the ancestor model tree is again because the ancestor may be the only model for which the property value exists at this very position because it may have been (conflictingly) reordered or deleted in both evolved models.

In addition to visualising the actual model elements, each evolved model's model tree visualises change and conflict information. Information about changes regarding the "whole" element and not particular properties, i. e., model element addition or relocation changes (see Section D.5.4.2), are displayed in front of each element's tree item; “+” and “*” are used to indicate addition and relocation changes respectively. (As discussed above, deleted model elements and the respective model element deletion changes are visualised in the ancestor model’s tree.)

Property value changes are visualised at the respective property’s values. Property values with property value added or reordered changes are marked with “+” and “*” signs. (As discussed above, removed property values and the respective property value removed changes are visualised in the ancestor model’s tree.)

If a change is not accepted (in the merged model), the sign indicating this change for the respective model elements is put in parentheses, i. e, “(“ and “)”, to indicate that the change was rejected.

If changes are conflicting and the conflict is unresolved, they are marked as such by adding a “!” (exclamation mark) in front of the conflicting changes. When a conflict has been solved (see Section 6.5.5 below), the exclamation is removed from the respective changes’ visualisation.

As will be discussed in Section 6.5.5, modifying the merged model during conflict resolution by accepting or rejecting changes or by changing the merged model directly, will update the acceptance state of changes and the resolution state of conflicts. The respective tree items in the model trees will then be updated accordingly.

By selecting a model element or a property value in a model tree, the respective element’s tree items get automatically selected (i. e., highlighted) in the other model trees if this element exists there; this way, modellers can easily see if and how the element changed in the evolved models and what it was like in the common ancestor model.

If a model element has an unresolved (direct or indirect) existence conflict (i. e., it does not exist in the initial merged model), it will be annotated with a “!” to indicate the conflict.

6.5.4.1. Visualising diagram changes and conflicts as part of UML model elements

Because OMOS models are constructed by means of diagrams, the UML model elements visualised in model trees are depicted in diagrams. In addition to depicting UML model elements, the model tree provides information about the diagrams in which the UML model elements are depicted. As for UML model elements, diagram symbol change and conflict information are provided if the symbol is changed in a respective diagram. The change and conflict information regarding the diagram symbols of a certain UML model element can be leveraged in model trees because the diagram symbols reference the UML model elements they depict[1]. This allows modellers to learn in which diagrams symbols representing a certain model element changed, and allows them to access the symbol in these diagrams in a convenient way by navigating from the model tree directly to the respective diagrams.

6.5.4.2. Model change impacts

In order to allow modellers to understand the impact of accepting or rejecting a change (which will be discussed in Section 6.5.5), the model tree shows for each model element those model elements which reference it (by means of reference properties). So even though the model element does not reference those referencing elements, information about those referencing model elements are visualised at the referenced model element. In addition to showing the type and name of each referencing model element, change and conflict information of the referencing property's property value are visualised (including the acceptance state of the change and the resolution state of the conflict). As discussed above, the visualisation of both states is updated when they change because the modeller accepts or rejects the respective changes during the process of resolving merge conflicts. This process is discussed in the next section.

6.5.5. Resolving merge conflicts and working with the merged model

After the initial merged OMOS model was automatically created (as discussed in Section 6.4), merge conflicts may exist. They are caused by contradicting changes made in parallel to both evolved models and/or diagrams. Modellers have to manually resolve conflicts in order to finally arrive at a merged OMOS model that does not have model merge conflicts (in its OMOS diagrams and the underlying UML model).

Conflict resolution is mainly based on accepting and rejecting conflicting changes. Doing so will automatically set the respective conflict's resolution state to "resolved."

[1]UML attributes and operations are not directly referenced from the class symbols which depict them in diagrams, but based on the class symbols reference to the UML class it depicts and the class symbol's details visualisation level, the depicted attributes and operations can be automatically determined.

The actual consequences of accepting or rejecting changes are discussed below. In case none of the conflicting changes should be accepted, modellers can also manually set a conflict's resolution state to "resolved."

As for accepting or rejecting conflicting changes, the same can be done for non-conflicting ones. Modellers might also want to reject changes because they are *semantically* (not syntactically) conflicting, i. e., a certain change does not reflect the envisaged semantics and is thus rejected by modellers.

The merge process, therefore, not only supports accepting or rejecting conflicting changes (in order to solve merge conflicts), it also allows (1) assigning arbitrary values for conflicting, non-conflicting, and unchanged property values, i. e., for any property of any model element, and (2) creating and deleting arbitrary model elements. The tooling provided for the conflict resolution process is, therefore, similar to the modelling tools used to create the original, to-be-merged OMOS models.

Accepting or rejecting changes affects the state of the merge model's element. The actual effect of accepting or rejecting them depends on the type of changes as is explained in the next sections.

6.5.5.1. Accepting an addition change

When an addition change is accepted, the respective UML model element or diagram symbol is added to the merged UML model or diagram. When an element is added to the merged model, its non-conflicting contained elements (referenced via containment reference properties), i. e., all elements which do not have *direct* existence conflicts, are automatically added to the merged model or diagram. Adding contained elements is done in a recursive manner. Elements contained by contained elements and so on are, therefore, also added to the merged model. Contained elements which have *indirect* existence conflicts can be accepted/added since these conflicts only exists because their container elements were not part of the merged model or diagram. Contained elements which were already added to the merged model or diagram are ignored.

Adding an element is only possible if the container element exists in the merged model or diagram. The container element is defined by the evolved model or diagram. If this element does not exist in the merge model or diagram, the modeller can choose to add the element to a different container element.

All elements added to the merge model by manually accepting an addition change will also reference those elements which they reference by means of non-containment reference properties if those elements exist in the merge model, i. e., if they do not have existence conflicts and were not deleted manually from the merged model or diagram (see Section 6.5.5.8).

The property values of the added elements are calculated in the same way as was used for calculating the property values of elements of the *initial* merged model elements or diagram (see Section D.6.4).

Elements already existing in the merged model or diagram might define references to added elements (these reference were, of course, defined in the evolved models or diagrams, not in the merged one). When the to-be-referenced elements are added to the merged model/diagram, the dangling references are updated and turned into real references. The respective dangling reference conflicts are removed. The property values of the referencing elements cannot be calculated in the same way as was used for calculating the property values of elements of the *initial* merged model elements or diagram (see Section D.6.4) because their values might have already been manually modified by modellers (see Section 6.5.5.8), therefore the element is appended to reference properties' values of the referencing elements; for single-valued references, this means the value is simply set.

If an addition change is conflicting, the conflict will be resolved when the addition change is accepted.

6.5.5.2. Rejecting an addition change

If the model element or diagram symbol is part of the merged model or diagram, rejecting an addition change will delete the element or symbol. This works in the same way as accepting a deletion change which will be explained next. If an addition change is conflicting, the conflict will be resolved when the addition change is rejected.

6.5.5.3. Accepting a deletion change

When a deletion change is accepted, the respective UML model element or diagram symbol will be deleted from the merged UML model or diagram. All its directly and indirectly contained elements will be deleted as well. All deleted elements are also removed from (the reference property values of) any referencing elements.

If a deletion change is conflicting, the conflict will be resolved when the deletion change is accepted.

6.5.5.4. Rejecting a deletion change

If the model element or diagram symbol is not part of the merged model or diagram, rejecting a deletion change will add it. This works in the same way as accepting an addition change explained previously. If an deletion change is conflicting, the conflict will be resolved when the deletion change is rejected.

6.5.5.5. Dangling reference conflicts

Such conflicts exist when a UML model element or diagram symbol references another one which is not part of the merged model or diagram. Modellers can resolve those conflicts by either adding the referenced element to the merged model or diagram, or by directly setting the conflict's resolution status to "resolved." The reference to the element which is missing in the merged model or diagram is then no longer marked as conflicting. The modeller thereby indicates that not referencing the element in the merged UML model or diagram is accepted and, therefore, resolves the dangling reference conflict.

6.5.5.6. Resolving conflicting concurrent model element relocation changes

When a model element was conflictingly relocated to different container elements, none of the two relocation changes is automatically accepted when the initial merged model is created and thus the model element does not automatically become not part of the merged model. In order to resolve model element relocation change conflicts, modellers thus either have to accept that the model element is not part of the merged model or have to define a container element (which already exists in the merged model) for it. The container element can be one of the container elements to which the element had been (conflictingly) relocated to in the evolved models, or it could also be a different model element (see Section 6.5.5.8). In the former case, the respective conflicting model element relocation change is accepted. In the latter case, both model element relocation changes are still rejected. In any case, the model element relocation change conflict is resolved and the model element does not have an existence conflict any more because it is now part of the merged model. Identical to accepting an addition change (as previously described), resolving a concurrent model element relocation conflict by bringing the respective model element into the merged model will also include its direct and indirect contained elements which do not have direct existence conflicts.

6.5.5.7. Resolving conflicting concurrent value property changes

Resolving concurrent value property conflicts is based on the individual values of the property (since each individual property value change is marked as conflicting). For single-valued properties, modellers can chose between one of the two evolved values or the ancestor; but they can also chose a completely different one (see Section 6.5.5.8). Doing so will update the property's value in the merged model, accept or reject the respective property value changes, and resolve the respective conflict.

For multi-valued properties, changes of individual values can be accepted or rejected. Doing so has the same effect as dealing with value changes of single-valued properties described above.

If the changed property is a *reference* one, accepting value property added changes or rejecting value property removed changes can only be done if the respective referenced model element actually exists in the merged model.

If the property is a *containment* reference one, accepting value property added changes or rejecting value property removed changes will affect the existence of the contained model element and all its direct and indirect sub-elements. Accepting or rejecting those changes has the same effect as accepting or rejecting an addition change as described previously.

6.5.5.8. Directly modifying the merged UML model and diagrams

As already mentioned in the previous section, modellers can modify the merged model and diagrams in any way during the conflict resolution phase of the model merge process. New model elements and diagram symbols can be created or existing ones can be deleted or modified. This allows modellers to resolve merge conflicts in a more flexible way than simply accepting or rejecting conflicting changes.

Modifying the merged model elements or diagrams might affect changes and conflicts of these elements. The respective changes and conflicts can be detected because a connection between elements and their changes and conflicts exists. Therefore, when elements and symbols are directly modified, the acceptance state of changes and the resolution state of conflicts is updated automatically. Their acceptance and resolution state are, therefore, kept up to date with the merged model's elements even though modellers did not directly deal with the respective changes and conflicts, but directly modified the respective model elements.

6.5.6. Exchanging partially merged models

According to the principles defined for the envisaged approach to working in parallel with OMOS models (see Chapter 4), modellers should be able to exchange partial merged models (during the merge process) with other modellers. These modellers can then resolve merge conflicts their fellow modellers could not solve. The OMOS model merge prototype discussed in this chapter, therefore, allows the storing of the partially merged model (and its diagrams), the changes and their acceptance state as well as conflicts and their resolution state. The merge approach allows the exchange of partially merged models between different modellers. This is made possible in order to allow one modeller to verify the elements of the merged model he worked on, and allows him to resolve merge conflicts involving these model elements. Another modeller could then do the same for the model elements he is responsible for. So, some merge conflicts can be resolved by one modeller and other conflicts by another modeller. Being able to exchange partially merged models therefore allows them to merge models (i. e., resolve conflicts) in a cooperative manner. This possibility, therefore, helps to increases the acceptance of merge-based collaboration approaches because all modellers who worked (in parallel) on the to-be-merged models can participate in the merge process. Dedicated meta-models are thus defined for describing and managing changes and conflicts. The part of the UML meta-model used for OMOS models is discussed in Section D.1, the meta-model for OMOS diagrams is provided in Section D.3 and a meta-model for changes and conflicts (including their acceptance and resolution state) is discussed in Section D.5.4.2.

6.6. Chapter summary and outlook

In this chapter, a visual merge approach for OMOS models (i. e., OMOS diagrams and the underlying UML model) was discussed. It was implemented as a proof of concept and represents one of the main contribution of this research. This chapter (as well as Chapter 5), therefore, discussed the realisation of research objective 4 defined in Section 1.2.2: design and implement software to carry out a proof of concept.

The diagram merge approach is based on the semi-automatic OMOS diagram layout presented in Chapter 5. Because the hierarchy of class symbols is explicitly defined by modellers, the diagrams' structure is formally defined and can be represented as a model (see D.7.1.1). This allows applying the same model differencing and merging approach to the diagrams as is applied to their underlying UML models.

With respect to diagram merging, the extension of the visual grammar of UML class diagrams to support user-defined hierarchical layout of class symbols allows for efficient model merging because fundamental diagram conflicts caused by absolute position conflicts of diagram symbols occurring for free-form diagram layout [RW98] can be avoided. Since a diagram can be modified by different modellers independently, the merge approach based on the semi-automatic layout therefore does not require modellers to rearrange diagram symbols in order to untangle merged diagrams. Two versions of a diagram can thus be combined without user interaction if there are no conflicts.

The next chapter will discuss the evaluation of the diagram layout approach and the visual merging approach for OMOS models provided in this chapter and Chapter 5.

7. Evaluation, contributions and conclusions

An evaluation of the usefulness of the proposed solution for working in parallel with OMOS models (whose principles and objectives are defined in Chapter 4 and whose design and implementation is discussed in Chapter 5 and Chapter 6) from the expert opinion of engineers in the field is provided in this chapter. This chapter provides answers to research objectives 5 and 6 defined for this research in Section 1.2.2, namely, test the solution using examples from the pilot study environment and evaluate the usefulness of the proposed solution from the expert opinion of engineers in the field.

7.1. Evaluation of the proposed solution by experts in the field with a real-world parallel modelling scenario

The suggested solution to working in parallel with OMOS models (i. e., semi-automatic layout for creating OMOS diagrams provided in Chapter 5 and the visual OMOS model merge approach including conflict resolution during the merge process as discussed in Chapter 6) has been evaluated by testing it. Two test cases are used to evaluate the suggested solution by experts in the field: parallel OMOS modelling using the semi-automatic layout approach and visual merging of the concurrently evolved OMOS models (which were created using the former approach).

7.1.1. Evaluation approach and test data

In order to validate the proposed solution for working in parallel with OMOS models, a part of the "Power Supply Input/Output and Gearbox Operation Modes" model (which was used for the pilot study presented in Chapter 3) was used. Three engineers from Bosch's "Automatic Transmission Control Units" group took part in the evaluation. The engineers tested the OMOS model editor and the model merger prototypes developed as part of this research (see Chapter 5 and Chapter 6). Both tools were installed at Bosch's (Schwieberdingen) site.

In order to validate the tools (and, thereby, the suggested approaches), a part of the "Power Supply Input/Output and Gearbox Operation Modes" model, which also has been used to the pilot study (see Chapter 3) was re-created by the engineers (using the tools). This part of the model was chosen because it reflects modelling use cases which are representative for the Automatic Transmission Control Units group's use cases. The re-created model represents only a part of the "Power Supply Input/Output and Gearbox Operation Modes" model. This part, however, consists of actual elements taken from the original OMOS model. Furthermore, the layout of the OMOS diagrams visualising the selected part of the model contains layout phenomena that convey semantic meaning (as discussed in Section 3.6), namely the grouping of class symbols to visually express their assignment to software layers and the ordering of class symbols in order to visually express their close semantic relationship. The engineers (evaluating the proposed solution) also worked on the original OMOS model. In terms of evaluating the solution they are, therefore, experts in the field (of modelling automotive software using the OMOS approach).

In order to validate the proposed approach for concurrent OMOS modelling based on scenarios which can occur in real-life situations for real OMOS models, two modelling tasks were defined (by the OMOS engineers in close collaboration with the author). Both tasks were previously realised in the original "Power Supply Input/Output and Gearbox Operation Modes" model — but back then, of course, in a sequential manner as there was no support for concurrently modifying OMOS models. As will be explained in the next section, the tasks were realised by different OMOS modellers in a parallel manner. The criteria for selecting the tasks for testing concurrent modelling are as follows:

- Both tasks should realise real-world use cases as they were already realised for the original "Power Supply Input/Output and Gearbox Operation Modes" model.
- The software components implemented/affected by both modelling tasks have to be thematically close enough to allow for concurrent modifications of the same model elements and diagrams in order to evaluate the suggested approach to dealing with conflicting changes. If the tasks were thematically very different, concurrent changes of the same parts of the model would be unlikely.

However, both tasks should be sufficiently different such that they can be realised in a concurrent manner by different modellers in *different* versions of the same model.

- The tasks should involve the creation and modification of diagrams. The same diagram symbols should be modified by both tasks to evaluate the visual model merge approach and its support for conflict reasoning, handling, and resolution. Furthermore, to verify the handling of UML model element changes and conflicts, it should also be possible to change the model elements underlying the diagram symbols.
- In order to evaluate the semi-automatic diagram layout approach and the meaning-preserving merge approach based on it, the original OMOS diagrams (re-created as part of the evaluation) should convey additional meaning through their layout, i. e., class symbols should be visually grouped and ordered as identified by the pilot study (see Chapter 3). Furthermore, the visual contexts of the same diagram symbols should be modified concurrently (by different modellers in different versions of the same model) in order to evaluate the visualisation and handling of diagram conflicts.

Based on the above criteria, two tasks for evaluating the proposed solution are defined as will be explained in the next section.

7.1.2. Evaluation of the OMOS model editor

In a first step, the engineers evaluated the OMOS model editor (implemented as part of this research, see Chapter 5). Using the editor, they created three OMOS models (an ancestor model and two evolved ones). Using the model editor allowed the evaluation of the semi-automatic layout approach (discussed in Chapter 5) for creating and laying out real-life OMOS diagrams (and for creating the underlying model). The created model elements and diagrams, including their names, are all taken from the original OMOS model. The model is partitioned into several diagrams in the same way the original model is partitioned. The diagram layouts are, where possible, kept similar to the original layouts (this is not always possible because of the hierarchic layout, but the class symbols are arranged and related to each other in a similar fashion as in the original diagrams).

Before the model merge tool can be evaluated in a second evaluation step, the two evolved models have to be created to evaluate the concurrent evolution of a model

(i. e., to evaluate parallel work on the same model). These two models represent the result of two modelling tasks (see below) that are realised in parallel and independently based on (two copies of) the ancestor model (of course, including its diagrams). The description of both modelling tasks is based on real requirements defined for the real-world “Power Supply Input/Output and Gearbox Operation Modes” model.

In order to better distinguish the evolved models and the modelling tasks realised in the models, the metaphor of two teams is used in the following discussion. The two teams could, for example, represent OMOS developers working on the gearbox controller software in Hungary and Germany (see Chapter 3). For the purpose of the evaluation, the OMOS engineers formed two teams and worked on the model in parallel.

Before the details of the three models are presented in the following section, an overview of the functionalities defined by those models is provided next.

Each team was assigned one of the following modelling tasks. Team A’s objectives were (1) to realise support for communicating with a car’s power supply electronic control unit (ECU), i. e., provide interfaces to access power supply information, and (2) to extend the low-side sensor control to support monitoring low-side stages. Team B’s objective was to realise support for dealing with a car’s battery ECU, i. e., provide interfaces to access battery information. The battery component is different from the power supply component, dealt with by team A, because (for gearbox controller software) a battery is (just) a storage device whilst the power supply takes into account additional components, for example, the dynamo.

Team A’s first task, i. e., adding power supply information to the model, is mainly realised by the classes in diagram “PSply” presented in Fig. 7.4. Classes “PSplyL3” and “PSplyL2” provide the implementation and classes “IfcPSupply” and “IfcPSupply_PSply” (see Fig. 7.7) provide the interface classes for accessing power supply information. All these classes were added to team A’s model.

Team A’s second task, i. e., replacing the basic monitoring functionality (already realised by class “LssCtrl” in the ancestor model, see diagram “Lss” in Fig. 7.1) with dedicated fault/malfunction monitoring functionality, is mainly realised by class “LssCtrl_Mn” and “LssMon” (see Fig. 7.5). The latter class replaces the fault monitoring formerly provided by class “LssCtrl.” That is why operation “Calc_LssMon” has been moved from “LssCtrl” to “LssMon.”

Team B’s task, i. e., providing access to a car’s battery information, is realised in a second model version (again a copy of the ancestor model). OMOS diagram "Battery" (see Fig. 7.8) defines the main classes realising the access to battery information (class "BatL3" and "BatL2"). Furthermore, the low-side sensor control class now accesses battery information via class "IfcBattL3." This model information was added in team B's version of diagram "Lss" (see Fig. 7.9)

7.1.2.1. The common ancestor model and its diagrams

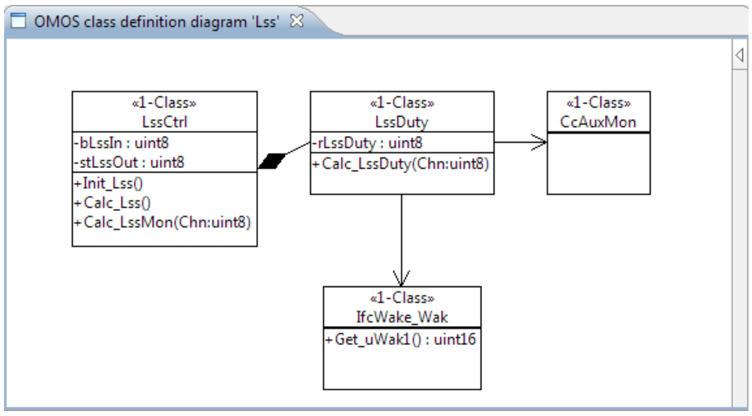


Figure 7.1.: OMOS diagram "Lss" (ancestor version).

In order to evaluate the concurrent modification of the model, a first model (the ancestor) has been created which provides the basis for parallel modifications of the model by the two teams. The OMOS model which serves as the starting point for the two parallel modelling tasks defines classes for controlling the "low-side sensors" (see Fig. 7.1). The layout of this diagram follows the original diagram's layout discussed in Section 3.5.2.

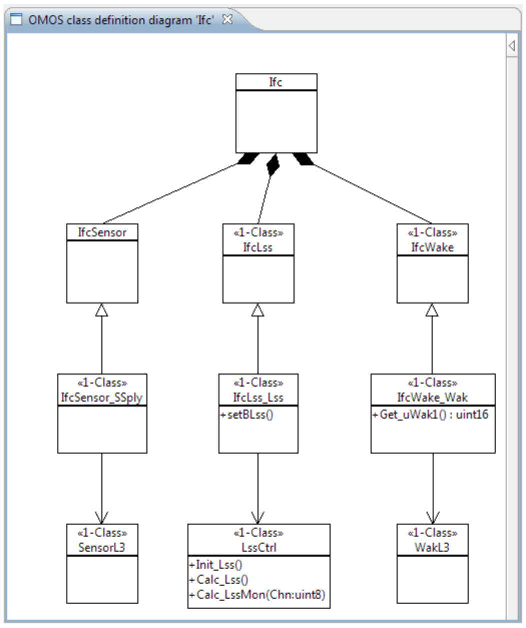


Figure 7.2.: OMOS diagram “Ifc” (ancestor version).

The ancestor OMOS model also provides functionalities for calculating low-side sensors input signals and for basic fault monitoring of the low-side controller. It defines interface classes dealing with the “wake-up signal” handling and “sensor supply voltage” access (see Fig. 7.2) used by other classes of this OMOS model. The layout of this diagram follows the original diagram’s layout discussed in Section 3.5.1.

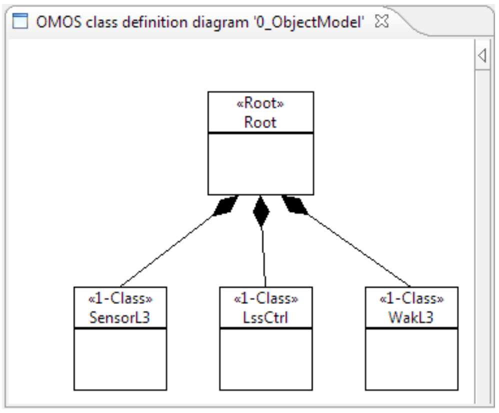


Figure 7.3.: OMOS diagram “0_ObjectModel” (ancestor version).

Finally, diagram “0_ObjectModel”, shown in Fig. 7.3, defines the hierarchy of the ancestor model's root classes. Again, the layout of this diagram follows the original diagram's layout discussed in Section C.4.

7.1.2.2. Team A's evolved model and its diagrams

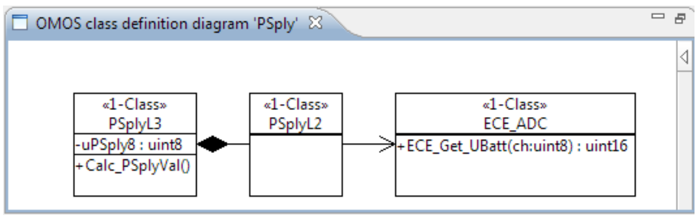


Figure 7.4.: OMOS diagram "PSply" (Team A version).

Fig. 7.4 presents diagram "PSply" added by team A to model the power supply classes ("PSplyL3", "PSplyL2", and "ECE_ADC") and their operations and attributes. The layout of this diagram follows the original diagram's layout discussed in Section C.3 and reflects the assignment of the depicted classes to software layers as discussed in Section 3.5 (using the OMOS model editor, modellers explicitly defined the order of the class symbols).

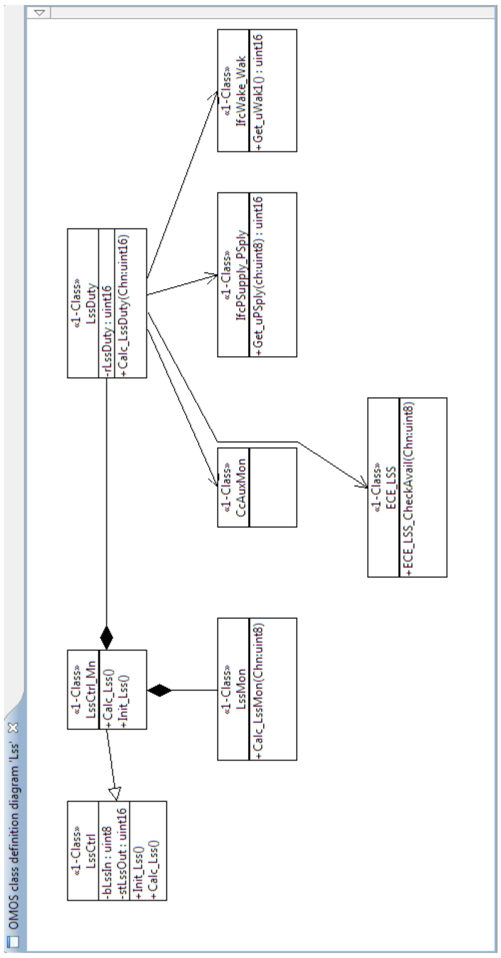


Figure 7.5.: OMOS diagram "Lss" (Team A version).

Fig. 7.5 presents team A's evolved version of diagram "Lss." According to team A's tasks, fault monitoring support for the low side controller (i. e., dedicated fault monitoring replacing the basic monitoring previously realised by class "LssCtrl") was added in team A's version of the model. The resulting evolved version of this diagram is similar to the original "Lss" diagrams discussed in Section 3.5.2. Its class symbols are arranged in the same way that reflects their assignment to software layers as the original diagrams were. However, with the OMOS model editor and its semi-automatic layout approach, the modellers were able to explicitly define the arrangement of the class symbols.

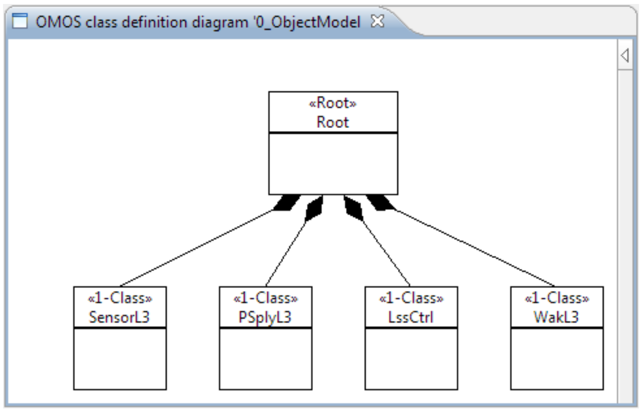


Figure 7.6.: OMOS diagram "0_ObjectModel" (Team A version).

The new class "PSplyL3" defined in diagram "PSply" (Fig. 7.5) was added to diagram "0_ObjectModel" (Fig. 7.6) which represents the hierarchy of the model's root classes.

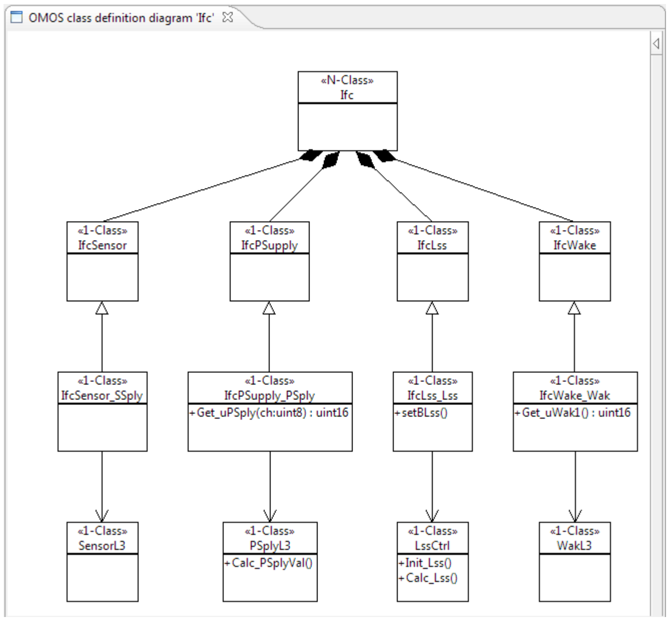


Figure 7.7.: OMOS diagram "Ifc" (Team A version).

According to team A's tasks, the new power supply interface classes are modelled in diagram "Ifc" (Fig. 7.7). This diagram defines all interface classes other classes in the OMOS model can make use of to, for instance, access information provided by other electronic control units. The new class symbols added in diagram “0_ObjectModel” and “Ifc” were arranged in the same as done in the original diagrams.

7.1.2.3. Team B’s evolved model and its diagrams

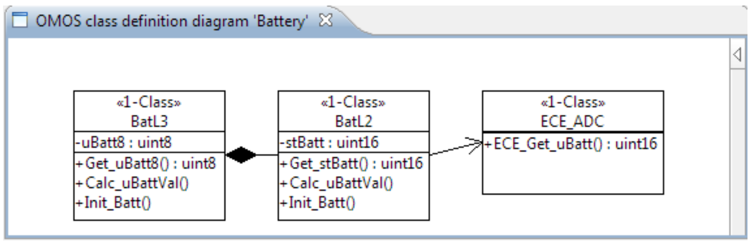


Figure 7.8.: OMOS diagram “Battery” (Team B version).

Team B’s new classes (“BatL3”, “BatL2”, and “ECE_ADC”) are modelled in a new diagram called “Battery” (Fig. 7.8). As requested by team B’s modelling tasks, these classes realise the functionality for accessing a car’s battery information.

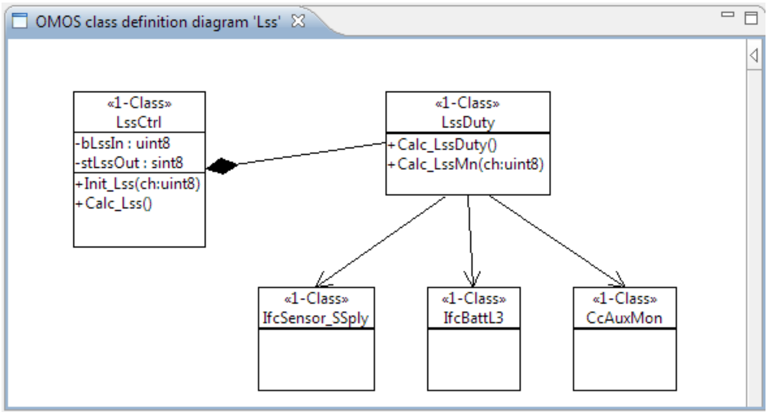


Figure 7.9.: OMOS diagram “Lss” (Team B version).

Access to battery information was added to the low side control via interface class “IfcBattL3” in diagram “Lss” (Fig. 7.9).

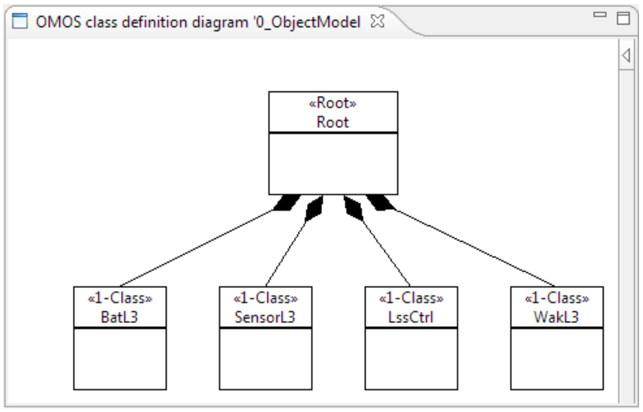


Figure 7.10.: OMOS diagram “ObjectModel” (Team B version).

Team B added class “BatL3”, defined in diagram “Battery” (Fig. 7.8), to diagram “0_ObjectModel” (Fig. 7.10) which defines the hierarchy of the model’s root classes.

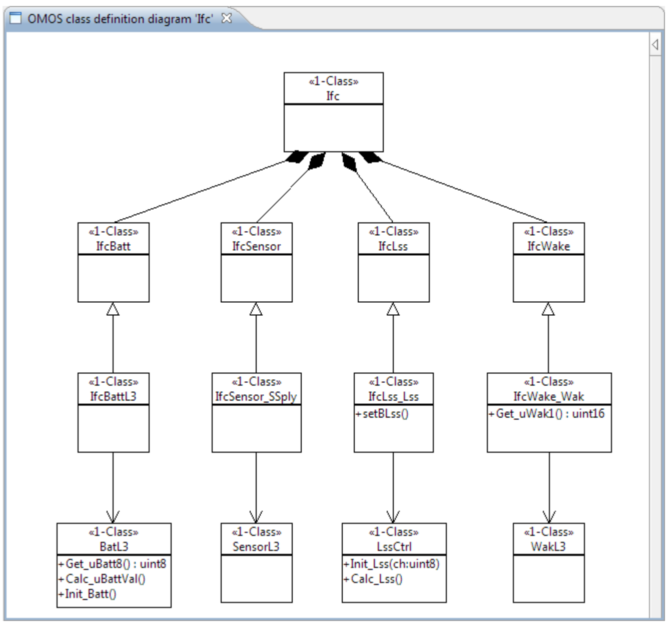


Figure 7.11.: OMOS diagram “Ifc” (Team B version).

The new battery interface classes (“IfcBatt”, “IfcBattL3”, and “BatL3”) are modelled in diagram “Ifc” (Fig. 7.11) which defines all interface classes other classes in the OMOS model can make use of. The class symbols added to diagram “0_ObjectModel” and “Ifc” were arranged according to the original diagram versions.

7.1.2.4. Summary

As explained above, to evaluate the semi-automatic layout approach (and the model editor realising it), the diagrams created by the modellers for all three models are laid out in a similar way as the original diagrams from the "Power Supply Input/Output and Gearbox Operation Modes" model. The additional semantic meaning conveyed through the original layouts (see Chapter 3) is now explicitly defined by the modellers by means of grouping and ordering class symbols. At the same time, class symbols can be arranged according to UML's layout guidelines (see Section 3.2.2). For instance, in diagram "Ifc", the newly added class symbols (i. e., "IfcSupply", "IfcSupply_PSply", and "PSplyL3" for team A's evolved diagram version, and "IfcBatt", "IfcBattL3", and "BatL3" for team B's) are positioned in the same visual context as in the original diagram discussed in Section 3.5.1. Because of the OMOS model editor's semi-automatic layout, the modellers did not have to rearrange the diagram layout when symbols were added (or removed), they simply dropped the classes symbols next to the desired neighbour or parent class symbols and the respective diagrams (7.7 and Fig. 7.11) were automatically laid out according to the class symbol hierarchy defined by the modellers.

With the suggested layout approach, modellers can only define the grouping and ordering of class symbols to define the symbol's hierarchy, however, they cannot influence the actual positions of the symbols. Nevertheless, the semi-automatic layout approach allowed modellers to re-create diagrams with similar layouts as the original diagrams. This confirms that the suggested layout approach allows the explicit definition of the secondary notation features required by OMOS modellers to define semantic relations conveyed through the layout of OMOS diagrams. As discussed in Section 7.2, this has also been acknowledged by the Bosch engineers who evaluated the approach.

7.1.3. Evaluation of the OMOS model merger

This section discusses the evaluation of the model merge tool (developed as part of this research, see Chapter 6). The tool is used to merge the OMOS test models and diagrams whose creation process is described in the previous section. The Bosch engineers used the merge tool to create the initial merged model from the evolved models created by team A and B (based on the common ancestor model). This model served as the starting point for evaluating the layout of the merged diagrams, the visualisation of (model and diagram) changes and merge conflicts, and the tool support related to change handling and conflict resolution.

7.1.3.1. Accessing all four models and visualising changes and conflicts

Following the principles of visual OMOS model merging defined in Chapter 4, the OMOS merge tool visualises changes and conflicts in the original versions of the models and diagrams (i. e., the ancestor and the two evolved versions). All four versions of each diagram and the underlying model are accessible to modellers during the merge process.

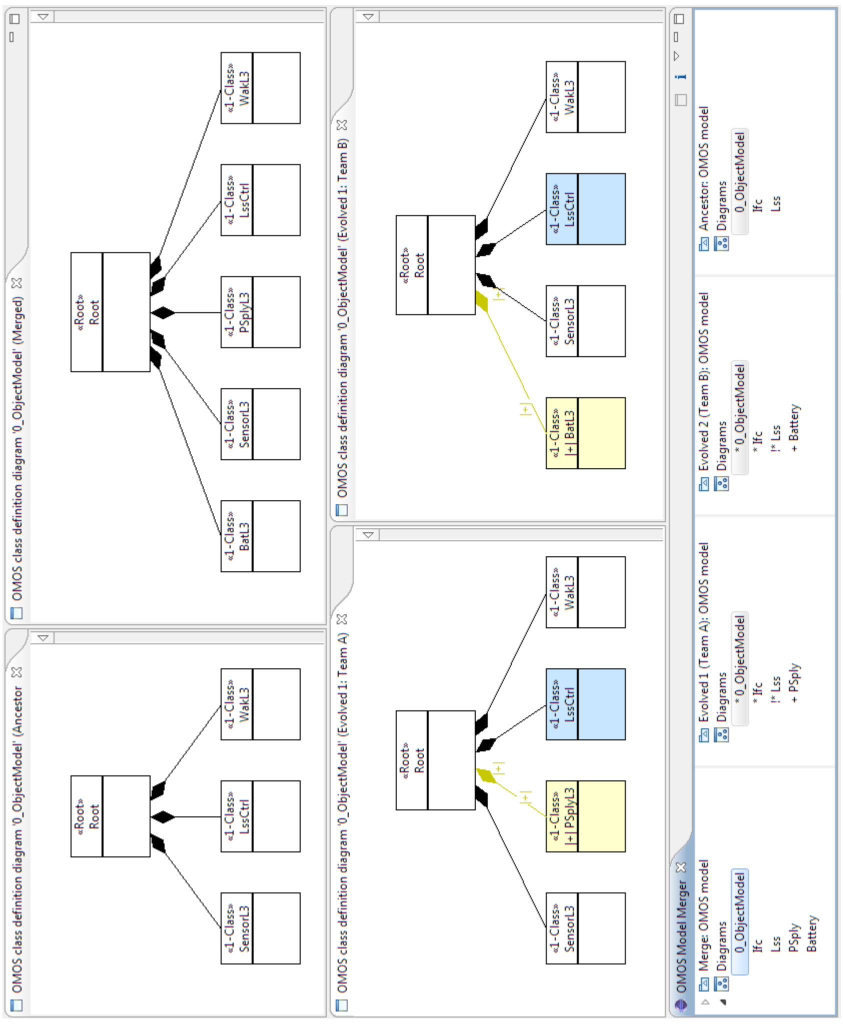

OMOS class definition diagram '0_ObjectModel' (Merged)
OMOS class definition diagram '0_ObjectModel' (Ancestor
OMOS class definition diagram '0_ObjectModel' (Evolved 1: Team B)
OMOS class definition diagram '0_ObjectModel' (Evolved 1: Team A)
«Root» Root
«1-Class» WakL3
«1-Class» LssCtrl
«1-Class» PSplyL3
«1-Class» SensorL3
«1-Class» BatL3
OMOS Model Merger
Merge: OMOS model
Diagrams
0_ObjectModel
Ifc
Lss
PSply
Battery
Evolved 1 (Team A): OMOS model
* 0_ObjectModel
* Ifc
!* Lss
+ PSply
Evolved 2 (Team B): OMOS model
+ Battery
Ancestor: OMOS model

Figure 7.12.: Merged OMOS “0_ObjectModel” diagram and its original versions.

For example, for diagram “0_ObjectModel” shown in Fig. 7.12, the initial merged diagram version is shown on the top-right, the ancestor diagram version is shown on the top-left, and the both evolved diagram versions are shown at the bottom. The latter three versions are visually annotated with information regarding changes and conflicts of the respective diagram and the respective underlying UML model (diagram “0_ObjectModel” has only model and diagram changes, but no conflicts). Furthermore, the diagram list including change information regarding the diagrams is shown at the bottom. This kind of visualisation of model and diagram changes and conflicts is also requested by the principles of visual OMOS model merging defined in Chapter 4 to allow convenient access to change and conflict information.

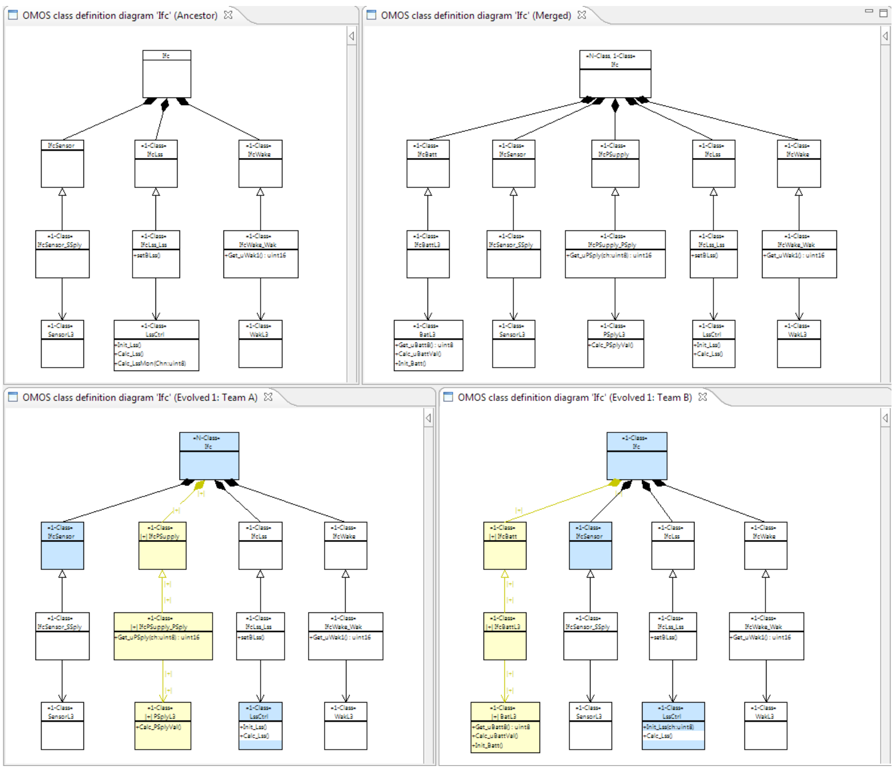

OMOS class definition diagram 'Ifc' (Ancestor)
OMOS class definition diagram 'Ifc' (Merged)
OMOS class definition diagram 'Ifc' (Evolved 1: Team A)
OMOS class definition diagram 'Ifc' (Evolved 1: Team B)

Figure 7.13.: Merged “Ifc” OMOS diagram and its original versions.

In Fig. 7.13, all four versions of diagram “Ifc” are presented. Again, the initial merged diagram version is shown on the top-right, the ancestor diagram versions is shown on the top-left, and the both evolved diagram versions below them. The latter three versions are annotated with change and conflict information regarding the diagram and the underlying UML model (the symbols of those diagrams and the underlying model element had no conflicts, but non-conflicting changes).

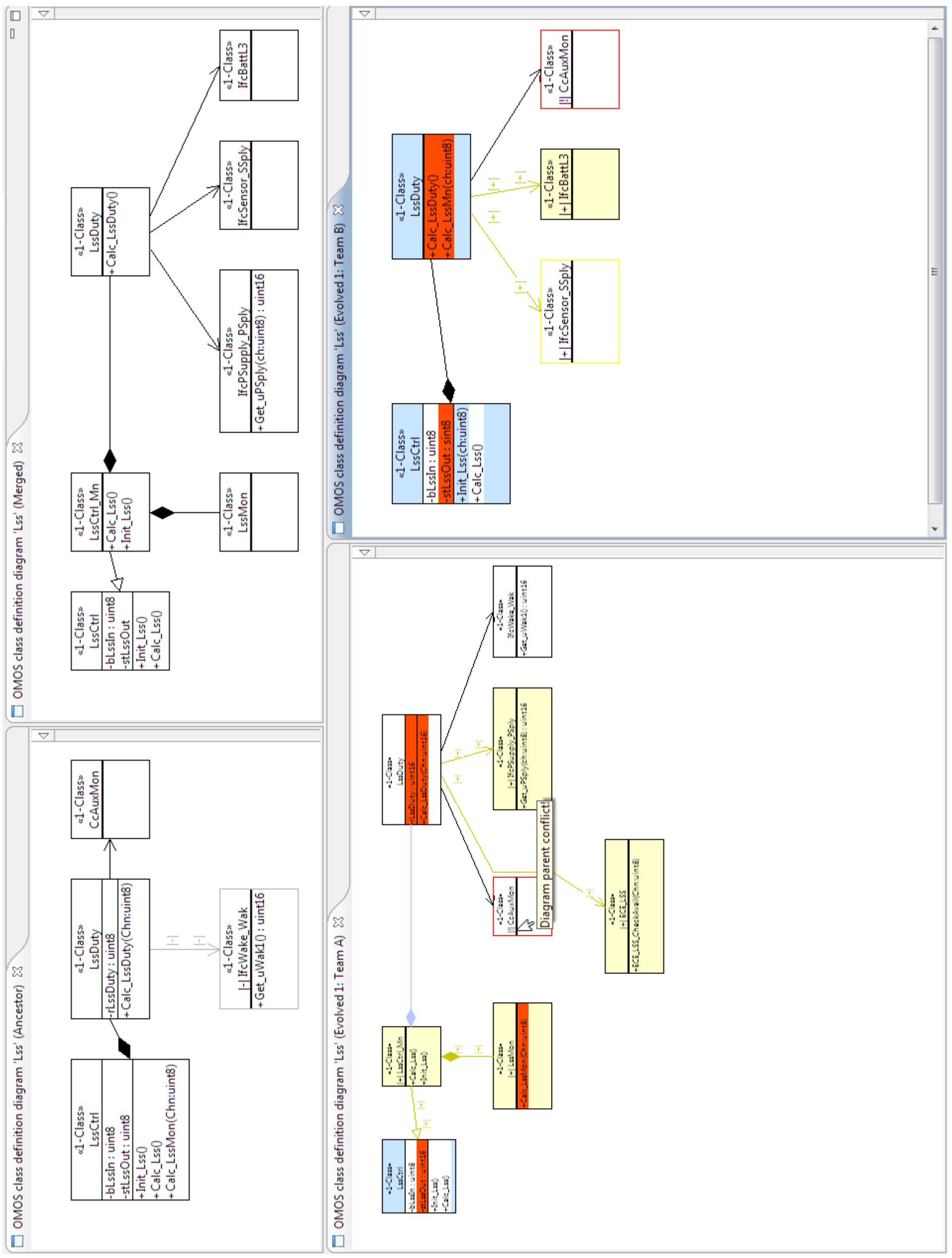

OMOS class definition diagram 'Lss' (Merged)
OMOS class definition diagram 'Lss' (Ancestor)
OMOS class definition diagram 'Lss' (Evolved 1: Team B)
OMOS class definition diagram 'Lss' (Evolved 1: Team A)
«1-Class» LssDuty
«1-Class» LssCtrl
«1-Class» LssCtrl_Mn
«1-Class» LssMon
«1-Class» IfcBattL3
«1-Class» IfcSensor_SSply
«1-Class» IfcPSupply_PSply
«1-Class» CcAuxMon
«1-Class» IfcWake_Wak
Diagram parent conflict!

Figure 7.14.: Merged “Lss” OMOS diagram and its original versions.

To evaluate the OMOS model merger’s approach to reasoning about and resolving model and diagram merge conflicts, the Bosch engineers made contradicting model and diagram changes to both evolved OMOS models. In Fig. 7.14, all four versions of diagram “Lss” are presented. Some diagram symbols and some UML model elements they depict were changed conflictingly. They are highlighted in red colour. To evaluate the visualisation and handling of diagram conflicts, class symbol “CcAuxMon” had been conflictingly relocated to different parent class symbols in both evolved diagram versions. That is why it is annotated with a diagram conflict (concurrent relocation conflict) and is not part of the initial merged diagram (shown on the top-right of Fig. 7.14).

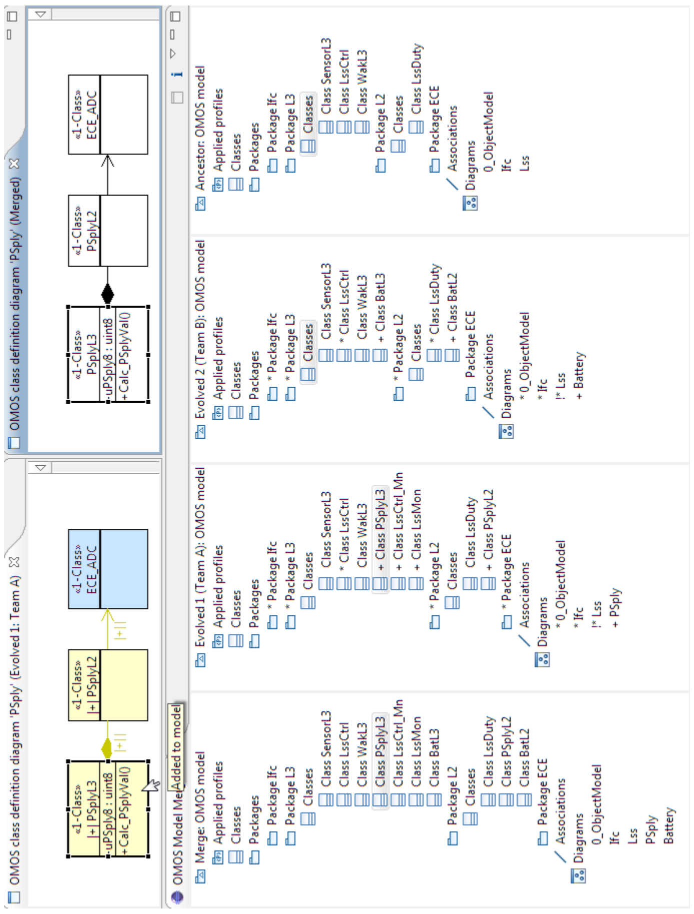


Figure 7.15.: Merged “PSply” OMOS diagram and the underlying UML model and its original version.

Diagram “PSply”, shown in Fig. 7.15, was added in team A’s evolved model version. That is why there exist only two versions of this diagram, the evolved diagram version (shown on the left) and the merged one (shown on the right). The evolved diagram version’s class and connection symbols are visually annotated with information about changes of the symbols themselves and their underlying model elements (if they have changed).

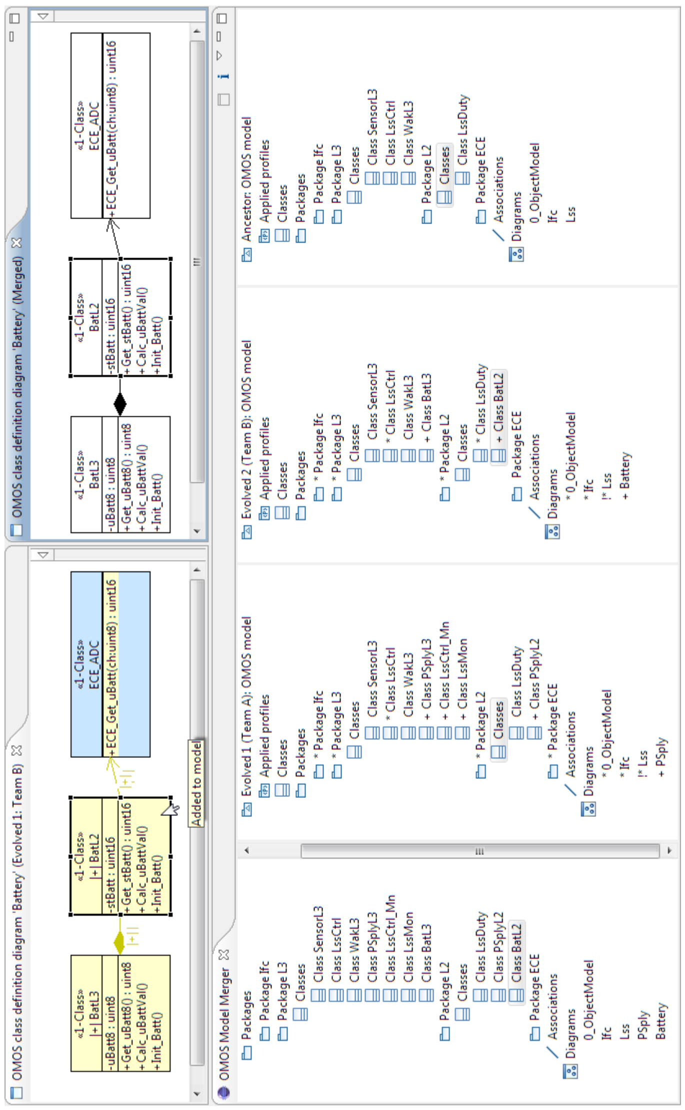


Figure 7.16.: Merged “Battery” OMOS diagram and the underlying UML model and its original version.

Diagram “Battery” shown in Fig. 7.16 was added in the evolved model version of team B. Similar to diagram “PSply” (which was added by team A), that is why there exist only two versions of this diagram, the evolved diagram version (shown on the left) and merged one (shown on the right). The evolved diagram’s symbols are visually annotated with change information regarding these symbols and, if they have changed, their underlying model elements.

7.1.3.2. Summary

As the examples in this section show, the suggested OMOS model merge process allows for automatically creating the initial merged model and its initial merged diagrams. Because the to-be-merged (ancestor and evolved) diagram versions (created in Section 7.1.2) contain information about the explicitly defined class symbol hierarchy, they can be automatically merged in a meaning-preserving way. For example, the initial merged version of diagram “0_ObjectModel” (on the top-right in Fig. 7.12) preserves the layout of both evolved diagram versions (shown on the bottom of Fig. 7.12). The same holds for the initial merged version of diagram “Ifc” in Fig. 7.13.

Diagram “Lss” (Fig. 7.14) could not be entirely merged automatically since it has a diagram layout conflict: class symbol “CcAuxMon” had been relocated in different ways in team A’s and B’s evolved diagram versions. However, despite the diagram merge conflict, the initial merged “Lss” diagram (shown on the top-right in Fig. 7.14) is still automatically laid out in a meaningful way and preserves the layout of both evolved diagram versions (shown on the bottom of Fig. 7.14). By looking at the change and conflict annotations visualised in the evolved diagram versions, modellers can reason about the diagram merge conflict and learn how to solve it.

In regards to visualising model and diagram changes and conflicts in a diagrammatic manner directly in (the ancestor and both evolved) diagrams, diagram “Lss” (Fig. 7.14) demonstrates that diagram changes and conflicts as well as those regarding model elements are directly visualised for the respective diagram symbols. For example, in addition to the diagram merge conflict of class symbol “CcAuxMon”, the addition of associations and classes is visualised in the respective evolved models (by means of “+” labels and beige colour) and conflicting model elements are highlighted in red colour (some of the attributes and operations shown in diagram “Lss” were modified in conflicting ways). To allow users to distinguish between symbol and model element changes and conflicts, model element changes and conflicts are visualised by filling the respective diagram symbol with the respective colour, whilst only the border to the symbols if coloured accordingly for diagram changes and conflicts.

Fig. 7.16 and Fig. 7.15 provide examples for visualising model and diagram changes and conflicts in the model tree view. Changes and conflicts of model elements are visualised directly at the affected model elements in the respective model tree views of the ancestor and both evolved model versions.

7.2. Success and impact of the proposed solution

The evaluation based on real OMOS models presented in the previous section focused on evaluating the three major approaches implemented as part of this research: the applicability of the suggested semi-automatic layout for creating meaning-conveying OMOS diagrams (discussed in Section 7.1.2), automatic meaning-preserving OMOS diagram merging and efficient handling of changes and conflicts during the merge process (as discussed in Section 7.1.3). Two test cases were, therefore, used to evaluate the solutions suggested by this research, i. e., parallel OMOS modelling based on a semi-automatic layout approach and visual merging of the concurrently evolved OMOS models.

As explained in Section 7.1.1, three Bosch engineers from the "Automatic Transmission Control Units" group who worked with OMOS on a daily basis evaluated the prototype tools developed as part of this research. The OMOS modelling tool as well as the merge tool were installed at Bosch's site and the experts in the field tested the proposed solution with real OMOS models.

In addition to providing feedback via email and phone calls, the tools developed as part of this research were discussed with the Bosch engineers in a meeting (which took place in November 2008). In this meeting, the engineers gave positive feedback regarding the results of their evaluation of the usefulness of the tools. Regarding the evaluation results, the most important feedback provided by Bosch is that they confirm that the solution allows them to work in parallel on OMOS models and, therefore, they will be able to use the OMOS approach to modelling automotive software systems in a more efficient way. The Bosch modellers also acknowledged that the semi-automatic layout approach suggested (which limits diagram layout freedom in favour of efficiently merging models, see Section 5.1.6) by this research allows them to convey the desired domain-specific information through the layout of diagrams and, at the same time, gain the ability to work in parallel on the same OMOS models because of the meaning-preserving diagrammatic merge approach. The engineers regard the semi-automatic diagram layout approach as an advantage for diagram merging since it allows automatically merging different versions of a diagram in a meaning- and, therefore, mental-map-preserving way. The modellers are positive about the visualisation of changes and conflicts directly in the common ancestor model and both evolved models because this approach provides access to all the models involved in the merge process. The modellers are also positive about the approach of visualising UML model element changes and conflicts directly at the affected symbols in diagrams. This way, they can reason about and deal with model and diagram changes and conflicts directly in the diagrammatic manner they are familiar with (from creating the diagrams in the first place), and can still distinguish between model and diagram changes and conflicts. Furthermore, the model trees (which are annotated with change and conflict information) allow them to reason about and deal with changes and conflicts in a model-wide context.

Regrading the semi-automatic diagram layout approach the Bosch engineers provided positive feedback on this approach since it allows them to create diagrams whose layout reflect their understanding of the domain and still lays out diagrams in an automatic, meaning-preserving way saving a lot of manual layout effort with respect to undesired diagram layout rearrangements. Here is an excerpt from an email to the author (from 13.06.2008) by M. Magiera, one of the engineers who took part in the evaluation, regarding the results of creating the three evaluation models (see Section 7.1.1) using the OMOS model editor:

> Hello Mr. Grimm,
>
> thank you for the delivery [of the OMOS model editor]. The tool is starting to look really good! The diagrams look quite good now! [...] We like the graphical layout. The dynamic adjustment of the [model tree view's] column width looks really great :) Using the tool is really fun. Keep up the good work![1]

As explained above, apart from minor issues the Bosch engineers also gave positive feedback on the visual OMOS model merge approach too. Here is a excerpt from an email to the author (from 21.10.2008) by M. Magiera regarding the results of merging the independently evolved OMOS models using the merge tool developed as part of this thesis:

[1]The original email (in German) read: *"Hallo Herr Grimm, danke für die Auslieferung. Langsam sieht das Tool wirklich gut aus! Die Diagramme sehen nun wirklich recht gut aus! [...] Die grafischen Darstellungen gefallen uns. Das Feature mit der dynamischen Änderung der Spaltenbreite sieht wirklich sehr gut aus :) So macht es so richtig Spaß das Tool zu bedienen. Weiter so! [...]"*

> Hello Mr. Grimm,
>
> [...] your [OMOS merge tool] delivery looks quite good. Generally, we like the [diagram] layout.
>
> We identified the following issues: A tad of colour would help to better recognise [model and diagram] changes. Could you please add graphical icons to modified relationships? The textual [icons] <+>, <!> or <*> are sometimes hard to recognize. The green check box looks a bit blurred. However, this is a minor issue.[2]

The official minutes of meeting of the final (prototype evaluation) meeting with Bosch conclude with

> We [Bosch] would like to thank Mr. Grimm for his excellent and dedicated work on this project.[3]

7.2.1. Bad timing for commercialising the solution

The prototype tools for working in parallel with OMOS models developed as part of this research were evaluated in 2008. Unfortunately, an economic downturn began in 2008 and Bosch was not prepared to provide additional funding to turn the prototypes into industrial-grade products. However, the author later had the opportunity to work on a project funded by the European Space Agency (ESA). This project [EMdK09] dealt with model-based space systems engineering (based on the ECSS-E-TM-10-23 meta-model[4]) and its requirements asked for the possibility for space systems engineers to work in parallel on the same model. The author, therefore, implemented a similar approach to model merging as has been done for working in parallel on OMOS models.

Encouraged by the success of the ESA project, the author's current employer now has enough interest to, together with Bosch, pursue further the business opportunity of implementing industrial-strength model merge tooling for Bosch's OMOS

[2]The original email (in German) read: *"Hallo Herr Grimm, [...] Ihre Auslieferung sieht ganz gut aus. Grundsätzlich finden wir die Darstellung gut gelungen. Uns sind die folgenden Punkte aufgefallen: Bei einer Änderung wäre ein bisschen Farbe schön damit die Änderungen gleich sichtbar sind. Könnten Sie bei Änderungen an Beziehungen hier auch grafische Icons hinzufügen? Die textuellen <+>, <!> oder <*> sind teilweise nur schwer zu erkennen. Der Grüne Hacken ist noch ein wenig verzerrt. Ist aber nicht zu wild. [...]"*

[3]The original minutes of meeting read: *"Wir danken Herr Grimm für die stets sehr gute und engagierte Arbeit für dieses Projekt."*

[4]http://atlas.estec.esa.int/uci_wiki/tiki-index.php?page=ECSS-E-TM-10-23 (access date: 4/12/2012)

approach to automotive software development based on the solutions suggested by and developed as part of this research.

8. Conclusions

The research presented in this thesis looks at an industrial, model-driven approach to developing software for electronic control units called OMOS (see Section 2.1.3). OMOS models are UML models used to model the static structure of electronic control unit software. OMOS models are created in a diagrammatic manners using UML class diagrams (called OMOS diagrams). The goal of this research is to allow more than one software engineer (potentially at different development sites) to work in parallel on the same OMOS models.

Following research objective 1 defined in Section 1.2.2, the need for parallel working when developing models in general and diagrammatic representations of models in particular was identified in the research literature discussed in Chapter 2. Furthermore, solution methods that others have reported that could potentially solve the problem were reviewed and available software was evaluated in this chapter to verify that there is no existing software that adequately solves the problem.

As requested by research objective 1b (see Section 1.2.2), it is confirmed by previous research (discussed in Chapter 2) that diagrams of models do indeed convey additional semantic information through their layout. Based on the findings from the research literature, a pilot study (see research objective 2 in Section 1.2.2) was then carried out in Chapter 3 to verify that the problem exists in the pilot study environment and to identify the extent of the problem. A real-life, industrial software project consisting of two OMOS models was analysed for the pilot study. The most important findings of this pilot study are:

1. OMOS models are constructed in a visual way using UML class diagrams. Each model is composed of a (potentially large) number of diagrams. Each diagram is used to model a part of a OMOS model which represents a certain area of the electronic control unit software defined in the particular OMOS model.
2. OMOS diagrams convey additional domain-specific knowledge through their layout. The pilot study determined that the class symbols of OMOS diagrams

are positioned in ways which reflect the modellers' domain understanding of these classes. This phenomenon is known as secondary notation (see Section 3.2.3), it refers to the fact that diagram layouts convey additional semantic information which are not part of the actual diagram notation but which bear semantic meaning important for modellers. The pilot study identified grouping and ordering of class symbols as the layout features used by modellers to convey domain-specific meaning through diagrams. It also identified that relationship symbols (i. e., connections between pairs of class symbols representing UML relationships like inheritance, association, and composition) are not the key factor for defining the position of the class symbols. Class symbols may thus be laid out in ways that do not follow the diagram layout guidelines suggested by the UML standard. The modellers' intention to express certain domain facts through diagram layout, therefore, "overruled" the UML guidelines.

Given that more than one OMOS modeller has be able to work independently and in parallel on a certain OMOS model, the envisaged (and implemented) approach for working collaboratively with OMOS models has to be an optimistic, i. e., merge-based one. Based on the literature review (see Section 2.3) on parallel work in general and parallel modelling in particular, a merge-based approach was chosen because it allows any modeller to potentially modify any model element independently from any other modeller.

The specifics of OMOS diagrams explained above had to be taken into account for the considerations made in order to provide an approach for collaborative modelling for OMOS models. On one hand, there is a potentially large amount of manually created UML class diagrams (one of the analysed OMOS models consisted of about 140 diagrams), and, as stated by the research literature (reviewed in Section 3.2), automatic diagram layout could help to merge these diagrams. However, merging manually laid out diagrams is a difficult (possibly futile) endeavour and will most likely result in cluttered diagrams which have to be untangled manually in order to remove overlapping symbols and layout issues (see Section 2.4). The resulting diagrams' layout might be destroyed and the semantic information conveyed through their layout is lost. On the other hand, merging all those diagrams manually or even requiring to manually untangle them after they have been automatically merged in a rudimentary way would be too time-consuming. Automatic layout approaches for laying out the diagrams comprising a OMOS model might appear to be a possible solution to this dilemma. Then, only the UML models comprising a OMOS model would have to be merged. Diagrams would not need merging because they are laid out automatically. Existing UML class diagram layout approaches are purely based on the information available in the model (see Section 2.4.3). UML class diagrams are, therefore, laid out according to a model's (inheritance and containment) relationships and the resulting hierarchies. However, as discussed in Section 4.1, this approach is not suited for laying out OMOS diagrams because:

1. Modellers have no influence on the way these model elements are laid out in a diagram. When a model element is removed, the layout of the diagrams depicting this model element might change considerably because the diagrams are newly laid out when the underlying UML model changes.
2. Automatic layout algorithms focus on laying out a model in a single diagram, a OMOS model, however, consists of a large number of diagrams. This drawback can be circumvented by allowing modellers to define which model elements should be depicted in a certain diagram and then use automatic diagram layout algorithm to arrange these model elements — with all the drawbacks of automatic layout.
3. The additional semantic information conveyed through the layout of diagrams is neither part of a diagram itself nor part of the underlying model. That is why conventional automatic diagram layout algorithms cannot take this information into account. Therefore, using conventional UML class diagram layout algorithms will remove the possibility of laying out class symbols of OMOS diagrams in a domain-specific way allowing to convey additional meaning through the layout of diagram symbols. The semantic information conveyed through the layout of diagram will, therefore, be lost.

Given the above drawbacks of conventional automatic layout, a semi-automatic approach to laying out OMOS diagrams was implemented as part of this research. With this approach, modellers are able to manually define the grouping and ordering of class symbols on a per-diagram basis. This manually provided domain-specific layout information is then used to automatically lay out OMOS diagrams. Modellers, therefore, define the layout information the pilot study revealed to be important for them with respect to embedding domain knowledge into the layout of OMOS diagrams. The layout itself is done in a completely automated fashion taking the class symbol grouping and ordering information defined by modellers into account. This approach allows modellers to build diagrams in accordance with their mental map (see Section 3.2.4) which reflects the modellers' understanding of the domain-specific relations of the depicted classes while still enabling efficient diagrammatic merging of OMOS models.

Following research objective 3 (defined in Section 1.2.2), the principles on which any software should be based to provide a solution for visually merging OMOS models in a meaning-preserving way are defined in Chapter 4.

Efficient diagram merging is enabled by semi-automatic OMOS diagram layout extending the visual vocabulary of OMOS class diagrams. Two prototype tools, a diagrammatic modelling tool and a diagrammatic model merging tool, were designed and implemented as part of this research (realising research objective 4 as defined in Section 1.2.2: design and implement software to carry out a proof of concept). The two main components of this tool are a diagram editor for visually creating OMOS models and a merge tool for merging OMOS models and interactively resolving merge conflicts.

Compared to completely manually or completely automatically laid out diagrams, the semi-automatic layout approach has two important advantages:

1. Because the grouping and ordering of class symbols can be manually defined (i. e., the grouping and ordering of class symbols is made explicit), additional domain-specific knowledge (conveyed by the layout of class symbols) is formally defined in OMOS diagrams.
2. Based on the manually provided layout information, OMOS diagrams can be automatically laid out and merged. Because of the explicit hierarchy of class symbols, a diagram's class symbols can be merged in a meaningful way which allows the preservation of the mental map of a diagram (in case no diagram merge conflicts exists). The automatic layout, therefore, yields untangled merged diagrams while preserving the manually defined grouping and ordering of class symbols. The modellers' mental maps and the additional semantic information conveyed through the diagram's layout are preserved. Modellers, therefore, do not have to rearrange diagram symbols to "unclutter" merged diagrams. Concurrent modifications of diagrams may lead to conflicting diagram layout changes (with respect to the grouping and ordering of class symbols) which might corrupt a modeller's mental map of a certain diagram. However, since the class symbol ordering and grouping (which, as the pilot study and other research confirm, are the most important features constituting the modellers' mental maps of OMOS diagrams) is explicitly defined, it is taken into account when diagrams are merged and, thus, changes and conflicts

can be detected and communicated to the modellers for them to verify the grouping and ordering of a diagram's class symbols and adjust it to resolve contradicting diagram layout changes.

The semi-automatic layout approach implemented for OMOS diagrams is a trade-off between diagram "mergablity" and manually creating OMOS diagrams with all the freedom with respect to positioning/laying out diagram symbols. The freedom of manual layout was reduced in favour of efficiently merging OMOS diagrams. However, the most important layout features (with respect to conveying additional domain-specific information through the layout of OMOS diagrams) can be manually and explicitly defined by modellers.

In contrast to other automatic UML class diagram layout approaches, no layout heuristics or iterative layout are applied by the implemented layout approach. These approaches are used to create more aesthetically pleasing and potentially more readable diagram layouts, but they have the drawback that the resulting layout might be different every time a diagram is laid out or when the model is updated (and, thus, the information used to calculate the layout changes). The layout approach implemented as part of this research aims for stable, predictable, and mergeable layout. This means that the grouping and ordering of class symbols is not altered as long as modellers do not change it. Connection symbols (depicting relationships between classes) are laid out completely automatically. A connection symbol's layout does not change as long as the order of the connected class symbols does not change.

Because the diagrams of a OMOS model evolve in a parallel manner (as their underlying UML models do), it is necessary to merge the diagrams comprising the OMOS models which evolved in parallel. Because modellers can manually modify diagrams, diagram merge conflicts can occur. Therefore, an approach allowing modellers to reason about and solve diagram merge conflicts was implemented. As described in Chapter 6 and Chapter 7, in addition to the semi-automatic layout approach for OMOS diagrams, an approach for differencing and merging OMOS models and models representing OMOS diagrams was implemented. The merge process consists of two steps. In the first step, the initial merged model is automatically created by determining the differences between two to-be-merged models and applying them to their common ancestor model. In a second step, modellers have to manually resolve merge conflicts. Tooling for manually dealing with merge conflicts were developed as part of this research.

All change and conflict information are directly visualised as part of the actual models and diagrams at the (conflictingly) changed model elements. This allows modellers to learn about changes and conflicts within the same visual/diagrammatic context they are familiar with from creating the to-be-merged models in the first place.

Besides visualising change and context information directly as part of the respective model elements, dedicated tooling supporting the resolution of merge conflicts has been defined and implemented. The merge tooling provides modellers with the possibility to resolve merge conflicts by accepting and rejecting model and diagram changes. Modellers cannot only modify the merged model/diagram by means of accepting or rejecting changes, they can modify it in any way. Even model elements/diagram symbols which were not changed at all (not even non-conflictingly) can be modified. The implement OMOS model merge tool provides specific editing capabilities for dealing with changes and conflicts, but it also provides the common editing functionalities of ordinary modelling tools used when models and diagrams are created in the first place. The dedicated merge tooling takes care of updating the acceptance status of changes when the merged model or diagram is updated — so that modellers can learn whether a change made in one model/diagram is (still) part of the merged model/diagram.

As requested by research objectives 5 and 7 (defined in Section 1.2.2), the usefulness of the proposed solution was evaluated by engineers in the field (see Chapter 7). Bosch engineers who worked with the OMOS models used for the pilot study (see Chapter 3) evaluated the semi-automatic diagram layout approach and the visual merge approach by testing the prototype tools developed as part of this research. Using the model editor, they re-created, in a collaborative manner using the parallel modelling approach suggested by this research, parts of the real-world OMOS models (developed by Bosch and) used for the pilot study. The engineers then merged the concurrently evolved OMOS models using the model merge prototype developed as part of this research. Bosch provided positive feedback about both prototype tools and the collaborative OMOS modelling approach. The semi-automatic layout approach allows them to convey semantic information through the layout of diagrams and enables efficient model merging preserving the additional meaning conveyed through the diagrams' layout. It also enables engineers (possibly at different sites) to work in parallel on the same OMOS models (which is one of the main objectives of the research presented here).

8.1. Future work

As discussed in Section 5.1.6, further research is required on the semi-automatic diagram layout approach presented in this thesis which gives modellers control over the parent-child hierarchy of class symbols and fully automatically lays out the respective diagram as a balanced tree (including connection symbols). This hierarchical approach enables meaningful merging of two versions of a diagram which evolved in parallel. In order to preserve the mental map of OMOS diagrams (see Section 3.2.4), the layout is kept stable by avoiding iterative and heuristics-based layout approaches as used by other automatic layout approaches to create more aesthetically pleasing layouts. To allow for more aesthetically pleasing layouts, trade-offs between stable, mental-map-preserving layout and layout aesthetics have to be made. Therefore, further research is required on the impact of rearranging diagram symbols on the mental map and layout aesthetics.

Furthermore, limiting the overall structure of diagrams to tree layouts could be avoided by defining several layout types. For example, certain types of diagram layouts might be better supported by not aligning all class symbols in horizontal layers but by aligning only a certain subset of closely related class symbols. This would then extend the visual expressiveness [Moo10] of (OMOS) diagrams by allowing modellers to define additional types of relations between diagram symbols.

Taking the extension of the visual expressiveness further, it might be feasible to define a visual language/grammar [Moo10] for OMOS diagrams which allows the expression of visual sentences and lets modellers define their intentions to express domain meaning in diagrams in an even more explicit way. The diagram merge approach could then take into account the visual tokens of this language to detect conflicts based on the visual language's grammar. Then, more educated merge decisions could be automatically taken and change and conflict visualisation and handling could be based on the visual grammar which would then allow the expression, visualisation, and reasoning of/about contradicting changes directly based on the visual language's grammar and its tokens.

Furthermore, on a more general note on future work, drawing from the success of another project (implemented by the author) which re-used parts of this research to implement concurrent modelling in the space systems domain, the author's current employer now has enough interest to pursue further the business opportunity of implementing industrial-strength model merge tooling based on this research's solutions.

8.2. Contributions

In conclusion, the contributions of this thesis are viewed to be as follows:

- An analysis of an industrial model-driven approach to developing software for electronic control units used in cars was conducted as part of this research. The findings of this analysis confirm that UML class diagrams are important for modellers because they convey additional meaning through their layout.
- The case of the general importance of diagram layout for model-based software engineering approaches has been made for a specific development approach (i. e., OMOS, see Section 2.1.3). This research confirms that modellers use diagram layout to convey additional meaning. This finding then has implications on collaborative modelling since diagrams have to be taken into account in addition to the models (i. e., structures) underlying them.
- An approach for laying out class diagrams in a semi-automatic fashion was implemented that allows modellers to manually define the grouping and ordering of class symbols and, at the same time, aims for diagrams to be mergeable. This approach provides a trade-off between layout freedom regarding the position of diagram symbols and the ability to create meaningful merged diagrams whose layout are untangled and preserve the manually defined class symbol hierarchy.
- An approach to visualising and dealing with differences and merge conflicts of merged OMOS models and their diagrams is presented in this thesis and implemented as part of this research. It enables modellers to work with merged models in the same way they are familiar with from creating OMOS models in the first place. It also allows the exchange of partially merged models between modellers in order to further support collaborative modelling during the conflict resolution phase.

A. Details on the OMOS modelling approach

A.1. Variant modelling and OMOS as a software product family approach

Bosch produces electronic control units (ECU) software systems for several customers. For example, gearbox controller software and hardware is produced for more than seven car manufacturers. Since each manufacturer usually has several different products (cars types) which all require, for instance, gearbox controller software, a large number of projects dealing with gearbox controller software can exist. Since all these projects have to cope with similar problems and belong to the same problem domain, i. e., gearbox controller software, the premise of OMOS is to allow to handle more than one project within a single OMOS model. OMOS is an approach that is capable of defining different, but related projects belonging to the ECU domain in the same models. Thus, OMOS is an industrial, model-driven *product family approach* for ECU software engineering [WL99, TH02, SDNB04].

Each customer (i. e., car manufacturer) has different requirements on the software controlling a gearbox. The requirements depend on the price category, type of the car (roadster, saloon car, etc.) and the driving experience (elegant, sporty, etc.) that customers expect from a certain kind of car. To fulfil these different requirements, different software implementations are necessary. However, even if requirements on ECU software systems may differ from customer to customer, all systems share many commonalities. Since the purpose of all systems is to control gearbox ECUs, the overall architecture of all systems is similar and different implementations usually share many functionalities. For instance, different products may use the same gear calculation functionality or may use the same protocol for communication with other ECUs (for instance, the controller area network, CAN). Since all implementations belong to the same problem domain and solve similar problems, they belong to the same product family, for instance, the family of gearbox controller software. The members of this family are called *products* [TH02].

OMOS focuses on efficient handling of a whole software product family within one OMOS model describing the design and implementation of several products. For example, a single OMOS model describes several gearbox controller software implementations. Thus a model contains classes that are used by all or a number of products and classes that are used by only one specific product.

Each base class in a OMOS model represents a certain functionality required to implement the ECU software. Since there can be variations between products on the same functionality, class inheritance is used to describe variations directly within the model. Hence, it is possible to include several variants of a certain functionality in the same model.

Base classes introduce a certain functionality, i. e., the realisation of a certain (set of) requirement(s). Concerning this specific functionality, this base class may not be suitable to realise the requirements of all products that are contained in the OMOS model. Therefore, subclasses are used to create a solution that realises the requirements on a specific functionality of a certain product or a number of products.

In OMOS, subclasses are called *variants* since they represent product-specific solutions of functionalities introduced by their base classes. Base classes are called *base variants*.

To realise the requirement that are specific for a certain product or set of products, a variant can override public and protected operations inherited from its base class, and introduce additional methods and attributes to realize its specialized functionality.

Variants may aggregate additional classes. This allows defining class hierarchies whose instances are included in some, but not in all products described by a OMOS model. This technique, allowing variants (sub-classes) to aggregate classes independently from its base class, is known as the *Bridge design pattern [GHJV95]*. This design pattern allows variants to add functionality in a more fine grained way than class inheritance does. Using this pattern, a variant can aggregate other classes which support it to realise the functionality specific to this variant. Therefore, subclasses do not necessarily need to rely on extending or overriding behaviour inherited from their base class. This allows to enable additional functionality to be used by certain variants only.

Fig. A.1 shows a OMOS model for cars with wheels controlled by the *anti-lock braking system* (ABS).

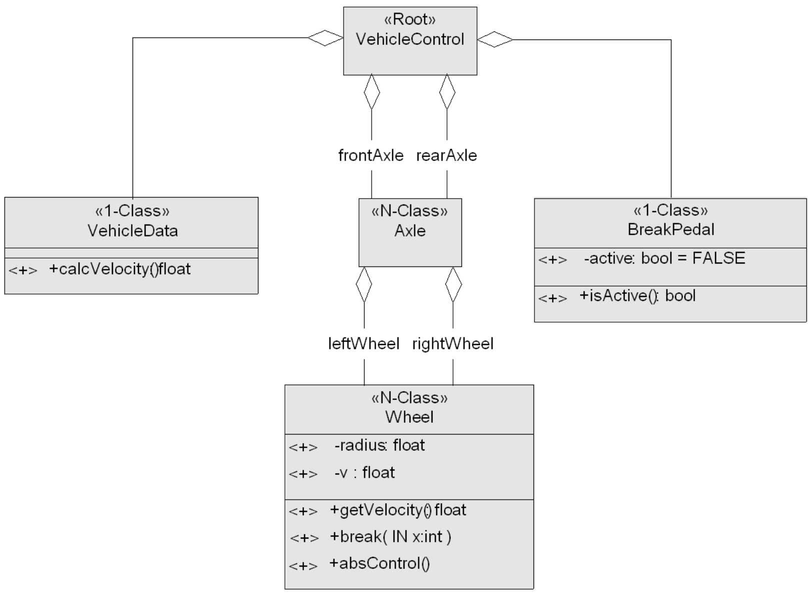


Figure A.1.: OMOS model showing the base variant of an antilock braking system.

Fig. A.2 shows a refined version of the ABS model shown in Fig. A.1. The refined model extends the ABS model by adding *anti-slipping regulation* (ASR). The model thus contains two OMOS products (implementations), the basic ABS controller software and the extended ARS version.

As mentioned in Section 2.1, each base class (except the root class) has to be aggregated by another class. Subclasses of aggregated classes inherit the class ownership defined by the aggregation. Therefore subclasses do not need to be aggregated by other classes. Aggregation is hence used to define the class structure (containment structure), while inheritance is used to describe the different variants.

A.2. The UML profile for OMOS

The details of the OMOS UML profile are be discussed in this section. Following UML's profiling mechanism discussed in Section 2.1.2.2, a UML profile for OMOS has been defined by this research. The original OMOS approach already supported the stereotypes defined below, but they were not formally defined by means of a

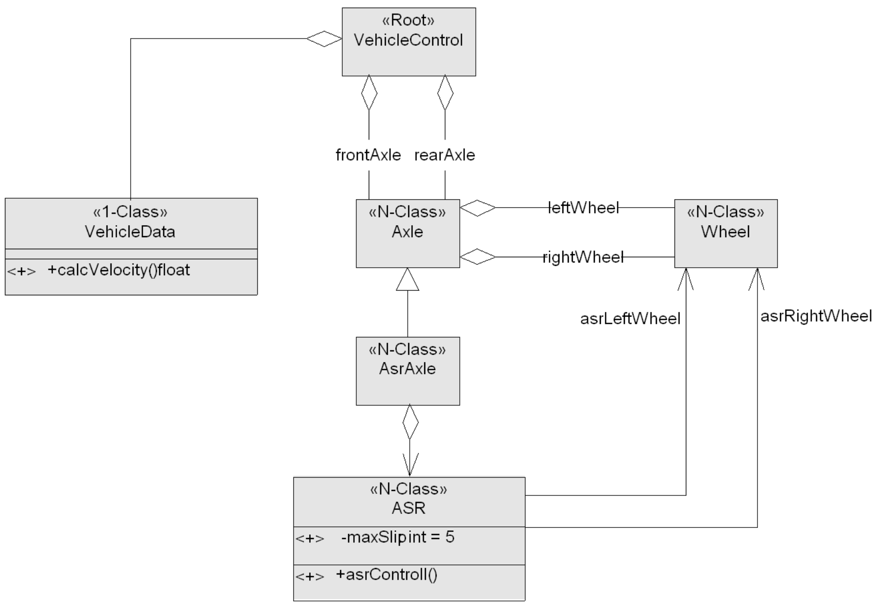


Figure A.2.: OMOS model showing a specialised anti-slipping regulation variant together with the basic anti-lock braking system variant.

UML profile, they were instead used in an *ad hoc* manner in OMOS models (which was possible because the UML modelling tool used to created OMOS models allowed to do so). The OMOS profile defines stereotypes for classes and properties:

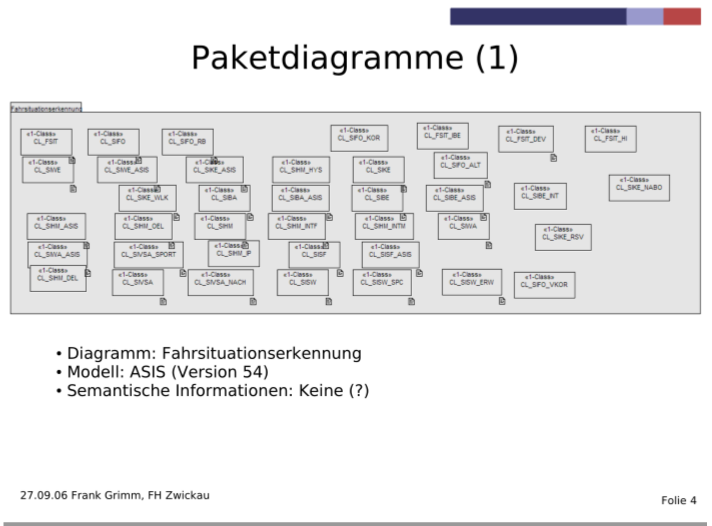
- Profile *OMOS*
  - Stereotype *1-Class (*extension of UML meta-class *Class).*
  - Stereotype *N-Class* (extension of UML meta-class *Class).*
  - Stereotype *Root* (specialization of Stereotype *1-Class).*
  - Stereotype *RAM_Groesse*[1] (extension of UML meta-class *Property).*
  - Stereotype *Kennwert*[2] (extension of UML meta-class *Property).*
  - Stereotype *Kennwerteblock*[3] (extension of UML meta-class *Property).*
  - Stereotype *Festkennfeld*[4] (extension of UML meta-class *Property).*
  - Stereotype *Systemkonstante*[5] (extension of UML meta-class *Property).*

[1]German for RAM value.
[2]German for characteristic value.
[3]German for block of characteristic values.
[4]German for constant characteristic value.
[5]German for system constant.

No tagged values (see Section 2.1.2.2) were used in OMOS; that is why the OMOS stereotypes do not define any properties.

B. Slides from the diagram layout analysis meeting discussing the pilot study findings

This chapter provides a selection of the slides used to discuss diagram layouts with to the OMOS modellers during the meeting (discussed in Section 3.3.3.1) which took part to discuss with Bosch the results of the pilot study conducted as part of this research (see Chapter 3). The slides were used to present the findings of the pilot study to the modellers and to validate these findings by discussing the diagrams with the modellers who created them.

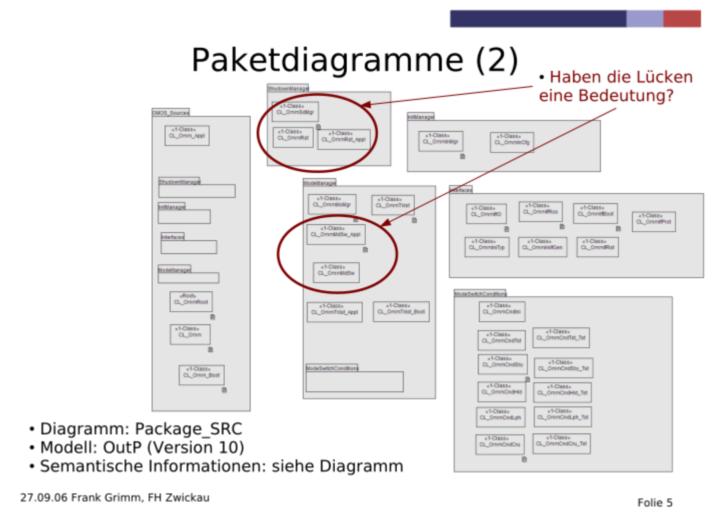


Paketdiagramme (1)

- Diagramm: Fahrsituationserkennung
- Modell: ASIS (Version 54)
- Semantische Informationen: Keine (?)

27.09.06 Frank Grimm, FH Zwickau

Folie 4



Paketdiagramme (2)

- Haben die Lücken eine Bedeutung?

- Diagramm: Package_SRC
- Modell: OutP (Version 10)
- Semantische Informationen: siehe Diagramm

27.09.06 Frank Grimm, FH Zwickau

Folie 5

Figure B.1.: Layout analysis meeting slides 4 and 5.

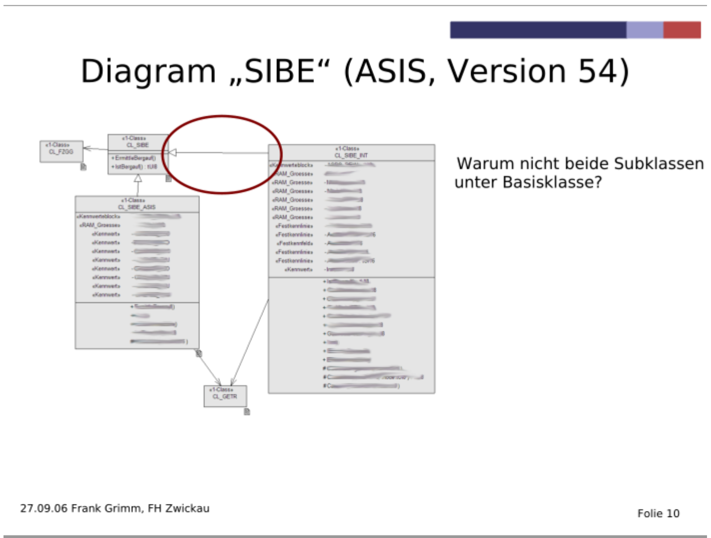


Diagram „SIBE“ (ASIS, Version 54)

Warum nicht beide Subklassen unter Basisklasse?

27.09.06 Frank Grimm, FH Zwickau

Folie 10

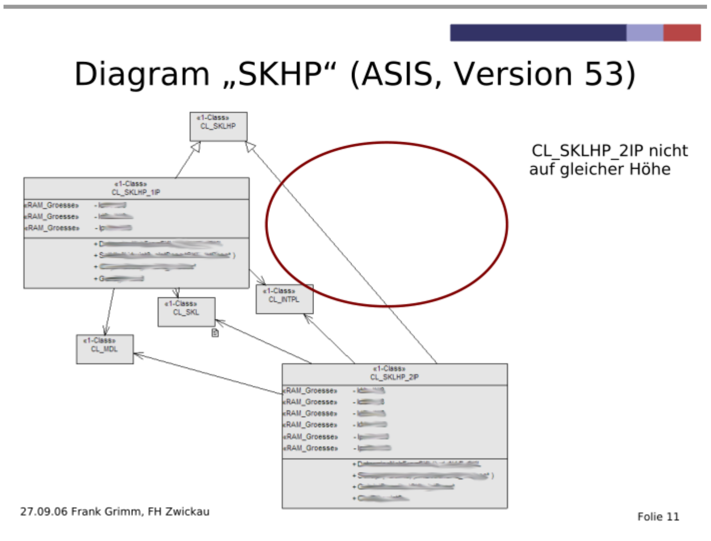


Diagram „SKHP“ (ASIS, Version 53)

CL_SKLHP_2IP nicht auf gleicher Höhe

27.09.06 Frank Grimm, FH Zwickau

Folie 11

Figure B.2.: Layout analysis meeting slides 10 and 11.

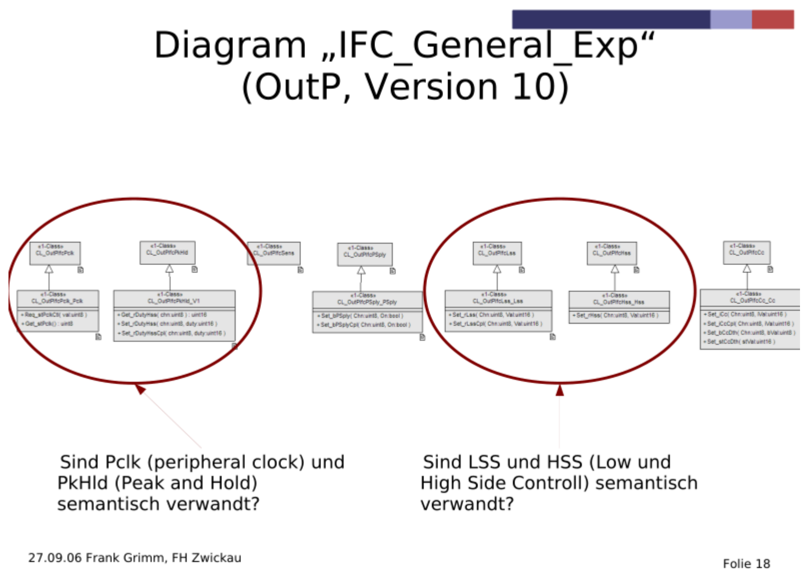


Diagram „IFC_General_Exp“ (OutP, Version 10)

Sind Pclk (peripheral clock) und PkHld (Peak and Hold) semantisch verwandt?

Sind LSS und HSS (Low und High Side Controll) semantisch verwandt?

27.09.06 Frank Grimm, FH Zwickau

Folie 18

Figure B.3.: Layout analysis meeting slide 18.

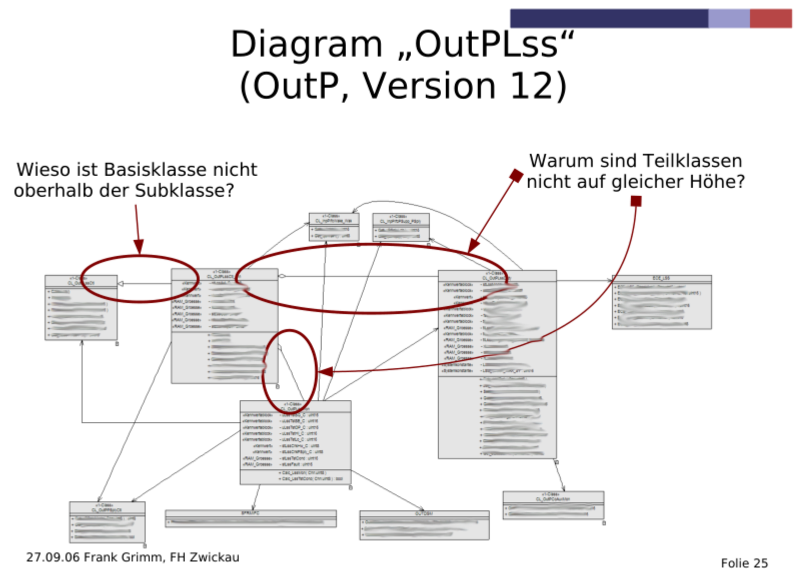


Diagram „OutPLss“ (OutP, Version 12)

Wieso ist Basisklasse nicht oberhalb der Subklasse?

Warum sind Teilklassen nicht auf gleicher Höhe?

27.09.06 Frank Grimm, FH Zwickau

Folie 25

Figure B.4.: Layout analysis meeting slide 25.

C. Additional analysed OMOS diagrams

This chapter provides additional results of the OMOS model and diagram analysis conducted as part of the pilot study discussed in Chapter 3.

C.1. Package assignment diagrams

In UML, packages are used to group classes into modules [OMG10b, p. 109]. Instead of displaying the package a class belongs to in each class diagram which contained symbols of this class. In OMOS, dedicated class diagrams are used to (1) assign classes to their package and (2) define the packages' hierarchy. These diagrams are called *package assignment diagrams.* The layout of package assignment diagrams does not convey domain-specific meaning. Packages are only visualised in these diagrams, they were *not* shown in diagrams used to define classes. However, the UML modelling tool (used to create the diagrams) allowed to display the package scope (i. e., the fully-qualified name) of classes on demand.

For the ASIS model the package hierarchy is defined in diagram "ASIS_Pool" (see Fig. C.1). It defines only packages and their hierarchy, additional package assignment diagrams exist for each of its sub-packages. For instance, for package "Fahrsituationserkennung" (driving situation detection in English) there was a diagram called "Klassen_Fahr situationserkennung" ("driving situation detection classes" in English) in which classes realising concepts belonging to the driving situation detection sub-domain are assigned to package "Fahrsituationserkennung" (see Fig. C.2). Since the only purpose of the diagrams is to add the classes to the package, only the names of the classes, but no details of any class are shown.

Package assignment diagrams are, of course, updated by modellers (in addition to the main diagrams used to define these classes) when classes are added to or deleted from a model. For instance, Fig. C.3 shows version 1 of diagram "Omm_Packages_SRC", in version three of this diagram, shown in Fig. C.4, several classes had been added to several packages.

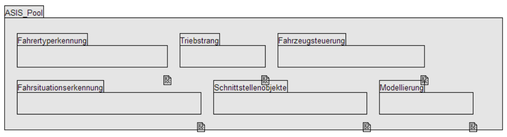


Figure C.1.: OMOS diagram “ASIS_Pool”.

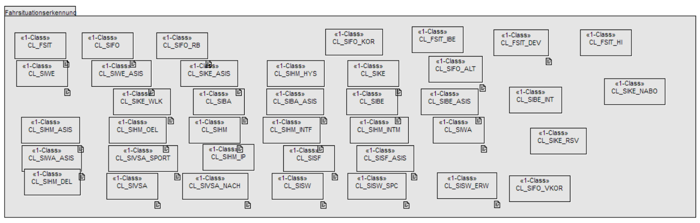


Figure C.2.: OMOS diagram “Klassen_Fahrsituationserkennung”.

C.2. OMOS diagram “InpP_Chip”

OMOS diagram “InpP_Chip” is depicted in Fig. C.5.

- Purpose of this diagram: Get input values from a chip.
- Layout findings: Flow from left to right (layer 3 to layer 2 to ECE); see Section 3.6 for a discussion on assigning class symbols to layers.

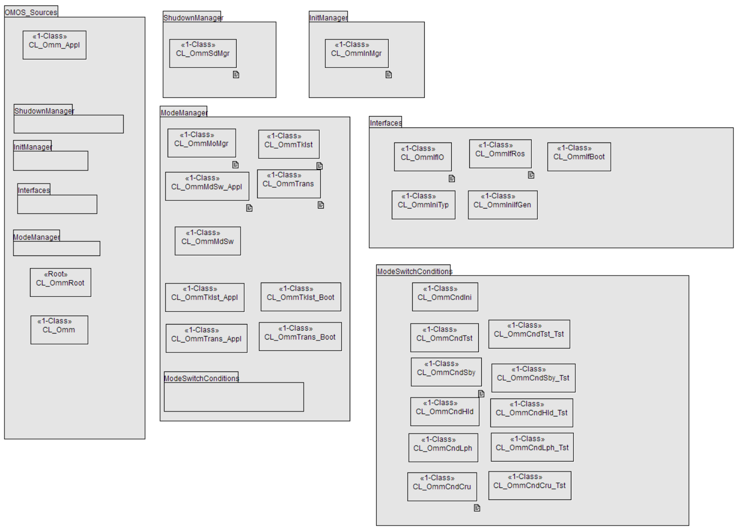


Figure C.3.: OMOS diagram “Omm_Packages_SRC” (version 1).

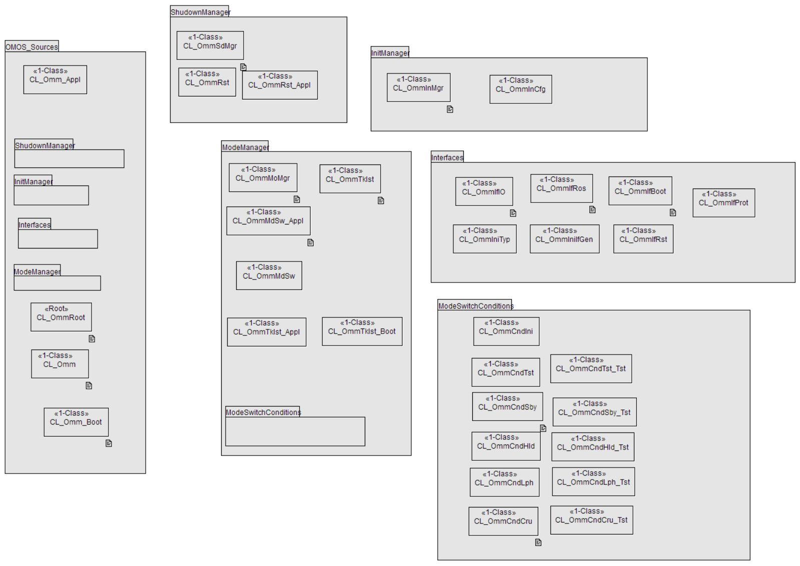


Figure C.4.: OMOS diagram “Omm_Packages_SRC” (version 3).

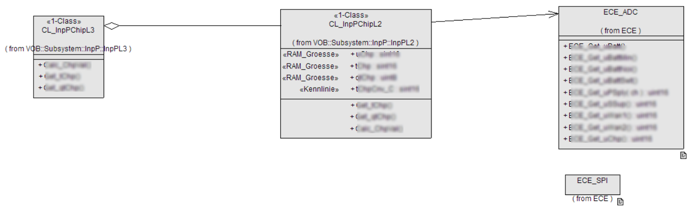


Figure C.5.: OMOS diagram “InpP_Chip”.

C.3. OMOS diagram “InpP_PSply”

OMOS diagram “InpP_PSply” is depicted in Fig. C.6.

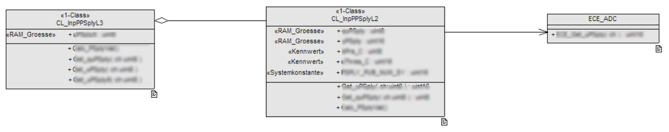


Figure C.6.: OMOS diagram “InpP_PSply”.

- Purpose of this diagram: Get input values from power supply.
- Layout findings: Flow from left to right (layer 3 to layer 2 to ECE); see Section 3.6 for a discussion on assigning class symbols to layers.

C.4. OMOS diagram “OutP_ObjectModel”

- The first analysed version of this diagram is shown in Fig. C.7.

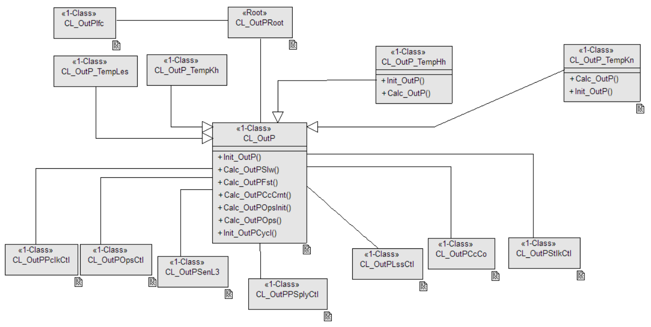


Figure C.7.: OMOS diagram “OutP_ObjectModel” (version 1).

- The second analysed version of this diagram is shown in Fig. C.8.
  - Layout changes in comparison to the first version:
    * New classes: CL_OutPIfcInt, CL_OutPPkHldL3, CL_OutPPkHld-L3_MN, CL_OutPLssCtl_Mn.
    * Class CL_OutPPkHldL3 (output for peak and hold injection) and CL_OutPPkHldL3_MN added in centre, i. e., in close proximity to

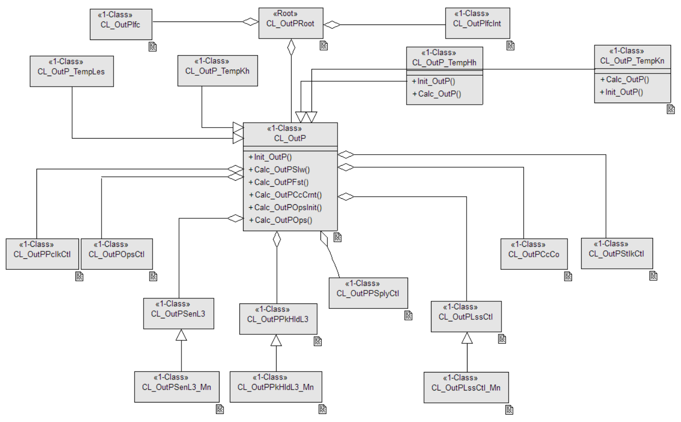


Figure C.8.: OMOS diagram “OutP_ObjectModel” (version 2).

symbols of semantically related classes (see Section 3.6 for a discussion on grouping semantically related class symbols in close visual proximity).

- The third analysed version of this diagram is shown in Fig. C.9.

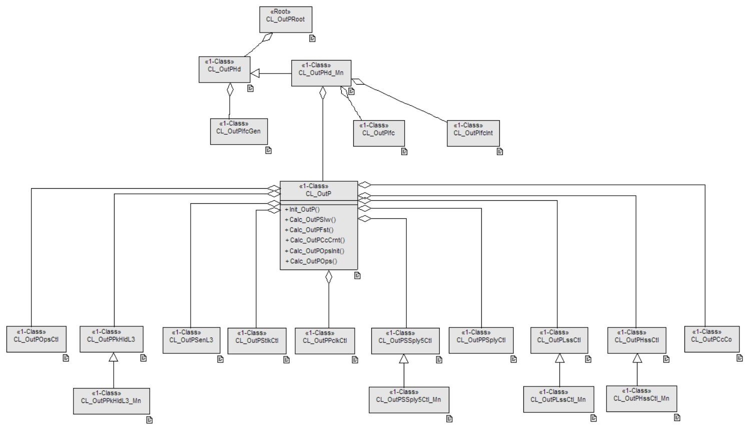


Figure C.9.: OMOS diagram “OutP_ObjectModel” (version 3).

  - Layout changes in comparison to the second version:
    * New: Class CL_OutPHssCtl (*high*-side current sensor controller) and CL_OutPHssCtl_Mn. It was positioned next to CL_OutPLssCtl (*low*-side current sensor controller) and CL_OutPLssCtl_Mn which is a sign for semantic grouping.

    * Class CL_OutPPclkCtl was moved to centre of the diagram, i. e., closer to symbols of semantically related classes (see Section 3.6 for a discussion on grouping semantically related class symbols in close visual proximity).
    * Class CL_OutPPkHldL3 and CL_OutPPkHldL3_Mn moved to right of CL_OutPOpsCtl (output safety controller).

- Layout findings:
  - Structure, especially horizontal ordering of classes identical to diagram “OutP_IfcGeneral” discussed in Section 3.5.1.

C.5. OMOS diagram “InpP_ObjectModel”

- The first analysed version of this diagram is shown in Fig. C.10.

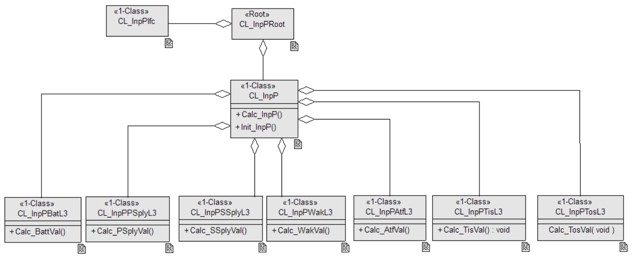


Figure C.10.: OMOS diagram “InpP_ObjectModel” (version 1).

- The second analysed version of this diagram is shown in Fig. C.11.

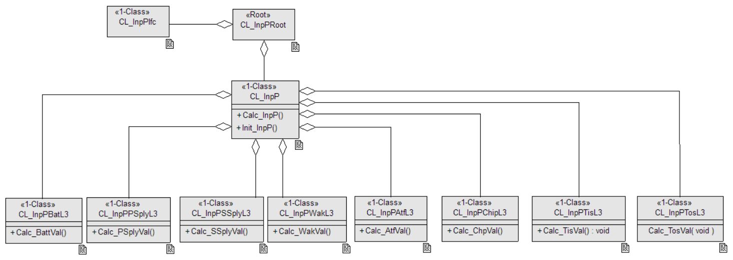


Figure C.11.: OMOS diagram “InpP_ObjectModel” (version 2).

- The third analysed version of this diagram is shown in Fig. C.12.

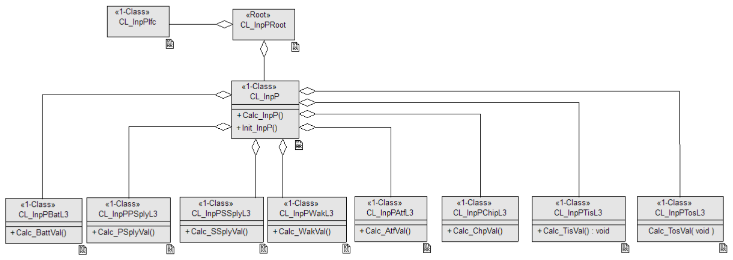


Figure C.12.: OMOS diagram “InpP_ObjectModel” (version 3).

- Layout findings: Layout, especially horizontal ordering of classes identical to diagram “InpP_IFC_General” discussed in Section C.6 and diagram “InpP_-IFC_GeneralExp” discussed in Section C.7.

C.6. OMOS diagram “InpP_IFC_General”

- Version 1: Does not exist.
- Version 2 is shown in Fig. C.13.

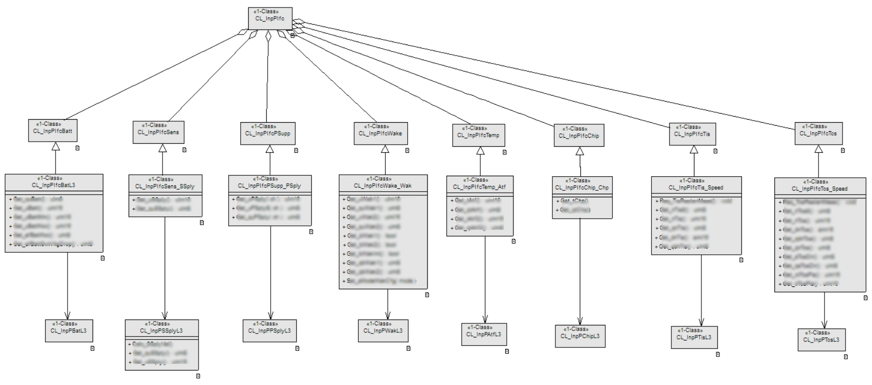


Figure C.13.: OMOS diagram “InpP_IFC_General” (version 2).

- Version 3 is shown in Fig. C.14.

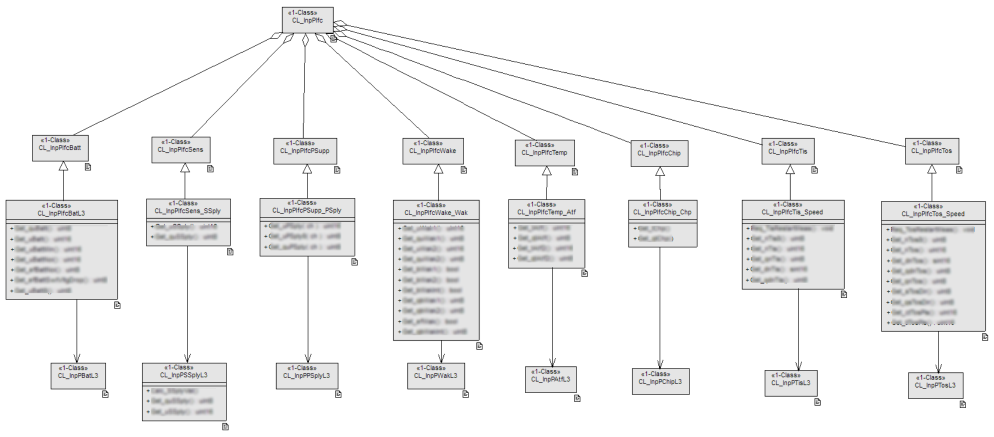


Figure C.14.: OMOS diagram “InpP_IFC_General” (version 3).

- Layout findings: Layout, especially horizontal ordering of classes identical to diagram “InpP_ObjectModel” discussed in Section C.5 and diagram “InpP_-IFC_GeneralExp” discussed in Section C.7.

C.7. OMOS diagram “InpP__IFC__General__Export”

- Version 1 is depicted in C.15.

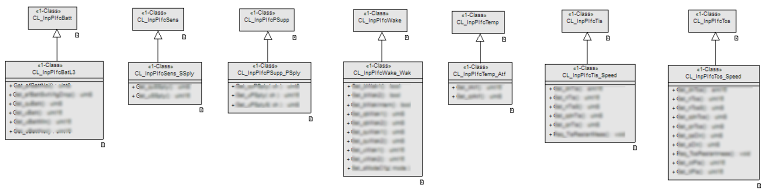

«1-Class» CL_InpPIfcBatt
«1-Class» CL_InpPIfcSens
«1-Class» CL_InpPIfcPSupp
«1-Class» CL_InpPIfcWake
«1-Class» CL_InpPIfcTemp
«1-Class» CL_InpPIfcTis
«1-Class» CL_InpPIfcTos
«1-Class» CL_InpPIfcBatL3
«1-Class» CL_InpPIfcSens_SSply
«1-Class» CL_InpPIfcPSupp_PSply
«1-Class» CL_InpPIfcWake_Wak
«1-Class» CL_InpPIfcTemp_Atf
«1-Class» CL_InpPIfcTis_Speed
«1-Class» CL_InpPIfcTos_Speed

Figure C.15.: OMOS diagram “InpPIfcExp” (version 1).

- Version 2 is depicted in: C.16.

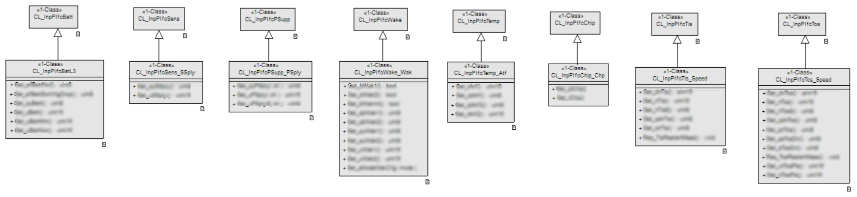

«1-Class» CL_InpPIfcBatt
«1-Class» CL_InpPIfcSens
«1-Class» CL_InpPIfcPSupp
«1-Class» CL_InpPIfcWake
«1-Class» CL_InpPIfcTemp
«1-Class» CL_InpPIfcChip
«1-Class» CL_InpPIfcTis
«1-Class» CL_InpPIfcTos
«1-Class» CL_InpPIfcBatL3
«1-Class» CL_InpPIfcSens_SSply
«1-Class» CL_InpPIfcPSupp_PSply
«1-Class» CL_InpPIfcWake_Wak
«1-Class» CL_InpPIfcTemp_Atf
«1-Class» CL_InpPIfcChip_Chp
«1-Class» CL_InpPIfcTis_Speed
«1-Class» CL_InpPIfcTos_Speed

Figure C.16.: OMOS diagram “InpP__IFC__General__Exp” (version 2).

  - Changes: Diagram renamed from “InpPIfcExp” to “InpP__IFC__General__-Exp.”

- Version 3 is depicted in: C.17.

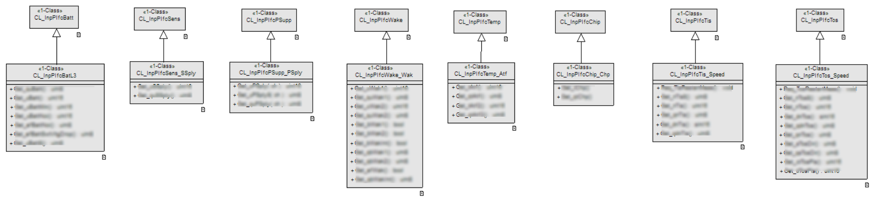

«1-Class» CL_InpPIfcBatt
«1-Class» CL_InpPIfcSens
«1-Class» CL_InpPIfcPSupp
«1-Class» CL_InpPIfcWake
«1-Class» CL_InpPIfcTemp
«1-Class» CL_InpPIfcChip
«1-Class» CL_InpPIfcTis
«1-Class» CL_InpPIfcTos
«1-Class» CL_InpPIfcBatL3
«1-Class» CL_InpPIfcSens_SSply
«1-Class» CL_InpPIfcPSupp_PSply
«1-Class» CL_InpPIfcWake_Wak
«1-Class» CL_InpPIfcTemp_Atf
«1-Class» CL_InpPIfcChip_Chp
«1-Class» CL_InpPIfcTis_Speed
«1-Class» CL_InpPIfcTos_Speed

Figure C.17.: OMOS diagram “InpP__IFC__General__Exp” (version 3).

- Layout findings: Layout, especially horizontal ordering of classes identical to diagram “InpP__ObjectModel” discussed in Section C.5 and diagram “InpP__-IFC__GeneralExp” discussed in Section C.7.

C.8. OMOS diagram “OutPStaLck”

- Software functionality: Control and monitor the electronic starter and steering locking device
- Version 1 and 2 are depicted in Fig. C.18.

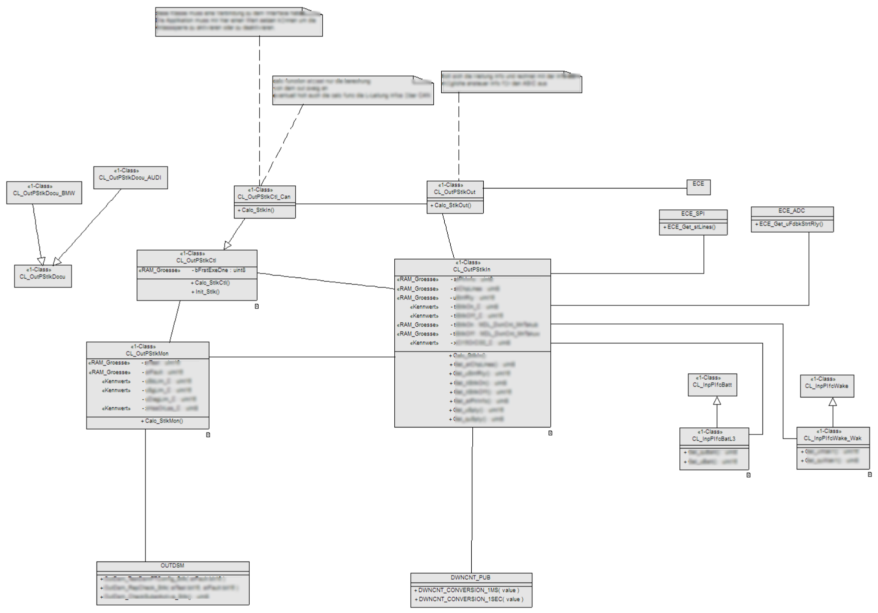


Figure C.18.: OMOS diagram “OupPStaLck” (version 1 and 2).

- Version 3 is depicted inFig. C.19.
  - Changes: Diagram renamed to “OutPStlk.”
- Layout findings: Layout similar to diagram “OutP__Lss” disucssed in Section 3.5.2 and diagram “OutP__Hss” discussed inSection 3.5.3. The layout of all those diagrams changed from version 2 to 3 in a similar way.

C.9. OMOS diagram “OutPSSply5V”

- Version 3 of this diagram is shown in Fig. C.20.
  - Changes: The diagram was added in version 3 of the analysed OMOS model.

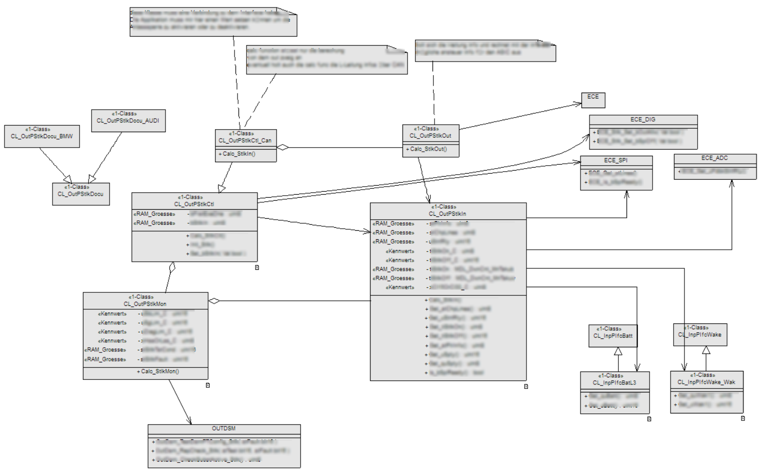


Figure C.19.: OMOS diagram “OupPStlk” (version 3).

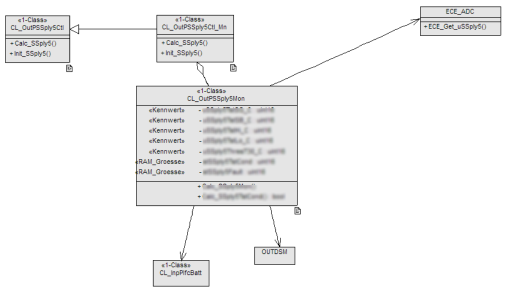


Figure C.20.: OMOS diagram “OupPPSply5V” (version 3).

- Layout findings: “CL_OutPSSply5Ctl” on same horizontal layer as “CL-_OutPSSply5Ctl_Mn”; see Section 3.6 for a discussion on assigning class symbols to layers.

C.10. OMOS diagram “IFC_OutP_internal”

- Version 2 of this diagram is shown in Fig. C.21.

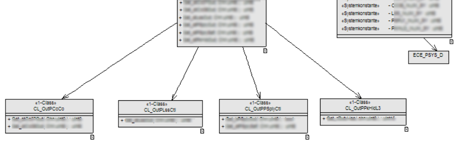


Figure C.21.: OMOS diagram “IFC_OutP_internal” (version 2).

  – Changes: The diagram was added in version 2 of the OMOS model.

- Version 3 if the diagram is shown in Fig. C.22.

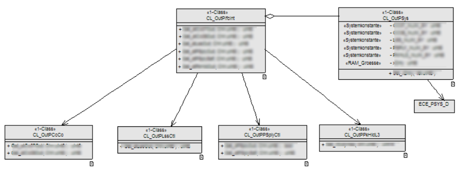


Figure C.22.: OMOS diagram “IFC_OutP_internal” (version 3).

  – Changes: Now attributes and operations were added to class “CL_-OutpSys.”

D. Development of tool support for visually creating and merging of OMOS models

A prototype OMOS modelling tool for creating UML models and OMOS diagrams (see Chapter 5) as well as a merge tool for differencing and merging UML models and OMOS diagrams (see Chapter 6) were implemented as a proof of concept for the research discussed here. The most important components of these tools will be outlined in this chapter.

The tools were implemented based on the Eclipse platform[1] as an Eclipse rich client application using the Java programming language. The implementation is based on the Eclipse Modelling Framework (EMF) which is an implementation of (parts of) meta-object facility (MOF, see Section 2.1.2.1) called Ecore which is part of the EMF [SBPM09]. Ecore and EMF are explained in more detail in Section D.4. Ecore substantially influenced the current MOF standard and lead to *Essential* MOF (EMOF), “a lightweight core of the metamodel that quite closely resembles Ecore” [SBPM09, p. 39]. Ecore models can be transformed into EMOF and vice versa [SBPM09, p. 129]. Ecore is an integral part of the *Eclipse* platform, and many other tools are based on EMF which provides a modelling framework and tools for a vast spectrum of application domains. EMF models are now becoming the standard facility for defining graphical user interfaces in the Eclipse platform. As explained in Section D.4, an implementation of the UML meta-model called Eclipse UML2[2] served as the basis for the implementation of the UML models underlying the OMOS diagrams.

The graphical parts of the prototype OMOS model editor and the model merge tools were implemented on the basis of the Eclipse Graphical Editing Framework (GEF)[3].

[1]http://www.eclipse.org/ (access date: 4/12/2012)
[2]http://www.eclipse.org/uml2/ (access date: 4/12/2012)
[3]http://eclipse.org/gef/ (access date: 4/12/2012)

The reason for implementing the OMOS model editor and the merge tool on top of Eclipse instead of using the Ameos modeller which is used by Bosch to create OMOS models is that a rich set of freely available software tools and frameworks (for instance, EMF, UML2, or GEF) exist for the Eclipse platform. Futhermore, Ameos is implemented in C and C++, therefore, reusing existing tooling from the Java-based Eclipse ecosystem is not trivial.

D.1. The part of the UML meta-model relevant for OMOS

The following UML meta-model elements are relevant for OMOS. They represent the building blocks OMOS models can be built from. The elements on the first level shown in the list below are instance of MOF class *Class* (see previous section). The first-level elements are defined by the UML meta-model. Their instances represent actual model elements.

The second-level elements below represent UML meta-model properties (i. e., instances of MOF class *Property*) of the first-level elements. They are either primitive, unstructured properties representing *attributes*, for instance, the *name* attribute of UML meta-model class *Parameter*, or structured properties representing *references* to other model elements, for instance, UML meta-model class *Operation* defines reference *ownedParameter* pointing to class *Parameter*. Some of the properties listed below are not directly defined by the classes defined below but by their (abstract) super classes. In such cases the name of the property is prefixed with the name of the UML meta-model class that actually defined the property. The complete set of UML meta-model classes and properties relevant for OMOS, including super classes which were not listed here, is presented in Section D.1.1.

Derived properties (see Section D.4.1.1) were not taken into account in the following list of OMOS-relevant UML meta-model elements because they are not important regarding to model comparison because their values are automatically calculated and thus cannot be changed manually by modellers.

- class *Model*
  - A model is a package (see class *Package* below). A OMOS model has exactly one *Model* instance, the model's root element.
- class *Package*
  - Name (value of attribute property *NamedElement::name*)

  - Profile applications (value of reference property *Package::profileApplication*)
  - Element imports (external model elements like primitive types (Boolean, String, etc.)) (value of reference property *Namespace::elementImport*)
  - Classes (value of reference property *Package::packagedElement* of type *Class*)
  - Packages (value of reference property *Package::packagedElement* of type *Package*)
  - Associations (values of reference property *Package::packagedElement* of type *Association*)

- class *Class*
  - Stereotypes (values of reference property *Element::appliedStereotypes*)
  - Generalizations (values of reference property *Classifier::generalization*)
  - Attributes (values of reference property *StructuredClassifier::ownedAttribute)*; attributes are of type *Property*, therefore, the attributes and reference defined above for class *Property* are of interest for OMOS.
  - Operations (values of reference *Class::ownedOperation*)

- class *Property*
  - Name (value of attribute property *NamedElement::name*)
  - Stereotypes (value of operation (query) *Element::getAppliedStereotypes*)
  - Type (value of reference property *TypedElement::type*)
  - Visibility (value of attribute *NamedElement::visibility*)
  - Staticness (value of attribute property *Feature::isStatic*)
  - Ordering (value of attribute property *MultiplicityElement::isOrdered*)
  - Uniqueness (value of attribute property *MultiplicityElement::isUnique*)
  - Lower bound (value of reference property *MultiplicityElement::lowerValue*)
  - Upper bound (value of reference property *MultiplicityElement::upperValue*)
  - Default value (value of reference property *Property::defaultValue*)

- class *Operation*
  - Name (value of attribute property *NamedElement::name*)
  - Abstractness (value of attribute property *BehavioralFeature::isAbstract*)
  - Parameters (value of reference property *Operation::ownedParameter*)

- class *Parameter*
  - Name (value of attribute property *NamedElement::name*)
  - Type (value of reference property *TypedElement::type*)
  - Direction (value of attribute property *Parameter::direction*)
  - Ordering (value of attribute property *MultiplicityElement::isOrdered*)
  - Uniqueness (value of attribute property *MultiplicityElement::isUnique*)
  - Lower bound (value of reference property *MultiplicityElement::lowerValue*)
  - Upper bound (value of reference property *MultiplicityElement::upperValue*)
  - Default value (value of reference property *Parameter::defaultValue*)
- class *Generalization*
  - Generic class (value of reference property *Generalization::general*)
  - Specific class (value of reference property *Generalization::specific*)
- class *Association*
  - Association ends (value of reference properties *Association::ownedEnd* and *Association::navigableOwnedEnd*); association ends are of type *Property*, therefore, the attributes and reference properties defined above for class *Property* are of interest for OMOS.
- class *Profile*
  - Name (value of attribute property *NamedElement::name*)
- class *Stereotype*
  - Name (value of attribute property *NamedElement::name*)
- class *ProfileApplication*
  - Applied profile (value of reference property *ProfileApplication::appliedProfile*)

Since the implementation of the prototype software tools discussed in this chapter uses the Eclipse UML2 implementation (see above), the part of UML meta-model used for the tools could not be modified in order to reduce it to the required model elements only (i. e., remove superfluous elements not required for defining OMOS models). Instead, the tools were configurable with regards to the UML meta-model elements whose instances should (not) be take into account (see above).

D.1.1. Listing of the UML meta-model elements relevant for OMOS models

Following the discussion on the UML meta-model elements relevant for OMOS above, Listing D.1. provides the parts of the UML meta-model which are relevant for OMOS models (i. e., UML class models) using a properly defined syntax conforming to the meta-object facility (see Section 2.1.2.1). The textual syntax (EMFText) used to present the UML meta-model is explained in Section D.4.3.

```
import("ecore")

package uml uml "http://eclipse.org/uml2"
{

class Model extends Package
{
  // has no OMOS-relevant features
}

class Package extends Namespace, PackageableElement,
    TemplateableElement
{
  unordered containment reference ProfileApplication profileApplication
      (0..-1) opposite applyingPackage;
  unordered containment reference PackageableElement packagedElement
      (0..-1);
}

class ElementImport extends DirectedRelationship
{
  unordered reference PackageableElement importedElement (1..1);
  unordered reference Namespace importingNamespace (1..1) opposite
      elementImport;
}

class Class extends EncapsulatedClassifier, BehavioredClassifier
{
  containment reference Operation ownedOperation (0..-1) opposite class
      ;
}

class Generalization extends DirectedRelationship
{
  unordered reference Classifier general (1..1);
  unordered reference Classifier specific (1..1) opposite generalization;
}

class Association extends Classifier, Relationship
{
  reference Property memberEnd (2..-1) opposite association;
}

class Property extends StructuralFeature, ConnectableElement, DeploymentTarget
{
  unordered containment reference ValueSpecification defaultValue (0..1);
  unordered attribute AggregationKind aggregation = "none" (1..1);
  unordered reference Association association (0..1) opposite memberEnd;
}

class Operation extends BehavioralFeature, ParameterableElement, TemplateableElement
{
  unordered reference Class class (0..1) opposite ownedOperation;
}

class Parameter extends ConnectableElement, MultiplicityElement
{
  unordered attribute ParameterDirectionKind direction = "in" (1..1);
  unordered containment reference ValueSpecification defaultValue (0..1);
}

abstract class MultiplicityElement extends Element
{
  unordered attribute Boolean isOrdered = "false" (1..1);
  unordered containment reference ValueSpecification upperValue (0..1);
  unordered containment reference ValueSpecification lowerValue (0..1);
}

abstract class ValueSpecification extends PackageableElement, TypedElement
{
  // has no features
}
```

```

abstract class LiteralSpecification extends ValueSpecification
{
  // has no features
}

class LiteralInteger extends LiteralSpecification
{
  unordered attribute Integer value = "0" (1..1);
}

class LiteralString extends LiteralSpecification
{
  unordered unsettable attribute String value (0..1);
}

class LiteralBoolean extends LiteralSpecification
{
  unordered attribute Boolean value = "false" (1..1);
}

class LiteralNull extends LiteralSpecification
{
  // has no features
}

class LiteralUnlimitedNatural extends LiteralSpecification
{
  unordered attribute UnlimitedNatural value = "0" (1..1);
}

enum ParameterDirectionKind
{
  0 : in = "in";
  1 : inout = "inout";
  2 : out = "out";
  3 : return = "return";
}

abstract class Type extends PackageableElement
{
  // has no OMOS-relevant features
}

```

```
abstract class TypedElement extends NamedElement
{
  unordered reference Type type (0..1);
}

abstract class ConnectableElement extends TypedElement,
    ParameterableElement
{
  // has no OMOS-relevant features
}

enum AggregationKind
{
  0 : none = "none";
  1 : shared = "shared"; // not used for OMOS
  2 : composite = "composite";
}

enum VisibilityKind
{
  0 : public = "public";
  1 : private = "private";
  2 : protected = "protected";
  3 : package = "package"; // not used for OMOS
}

abstract class Feature extends RedefinableElement
{
  unordered attribute Boolean isStatic = "false" (1..1);
}

abstract class StructuralFeature extends Feature, TypedElement,
    MultiplicityElement
{
  unordered attribute Boolean isReadOnly = "false" (1..1);
}

abstract class BehavioralFeature extends Namespace, Feature
{
  unordered attribute Boolean isAbstract = "false" (1..1);
  containment reference Parameter ownedParameter (0..-1);
}
```

```
abstract class Classifier extends Namespace, RedefinableElement , Type,
    TemplateableElement
{
  unordered attribute Boolean isAbstract = "false" (1..1);
  unordered containment reference Generalization generalization (0..−1)
      opposite specific;
}

abstract class StructuredClassifier extends Classifier
{
  containment reference Property ownedAttribute (0..−1);
}

abstract class EncapsulatedClassifier extends StructuredClassifier
{
  // has no OMOS−relevant features
}

abstract class BehavioredClassifier extends Classifier
{
  // has no OMOS−relevant features
}

abstract class RedefinableElement extends NamedElement
{
  // has no OMOS−relevant features
}

class Profile extends Package
{
  unordered derived volatile transient reference Stereotype
      ownedStereotype (0..−1);
  unordered reference ElementImport metaclassReference (0..−1);
}

class ProfileApplication extends DirectedRelationship
{
  unordered reference Profile appliedProfile (1..1);
  unordered attribute Boolean isStrict = "false" (1..1);
  unordered reference Package applyingPackage (1..1) opposite
      profileApplication;
}

class Stereotype extends Class
```

```
{
  // has no OMOS-relevant features
}

abstract class Element
{
  unordered operation Stereotype (0..-1) getAppliedStereotypes();
}

abstract class NamedElement extends Element
{
  unordered unsettable attribute String name (0..1);
  unordered unsettable attribute VisibilityKind visibility = "public"
      (0..1);
  unordered derived unchangeable volatile transient attribute String
      qualifiedName (0..1);
}

abstract class Namespace extends NamedElement
{
  unordered containment reference ElementImport elementImport (0..-1)
      opposite importingNamespace;
}

abstract class ParameterableElement extends Element
{
  // has no OMOS-relevant features
}

abstract class PackageableElement extends NamedElement,
    ParameterableElement
{
  // has no OMOS-relevant features
}

abstract class TemplateableElement extends Element
{
  // has no OMOS-relevant features
}

abstract class Relationship extends Element
{
  // has no OMOS-relevant features
}
```

```

abstract class DirectedRelationship extends Relationship
{
  // has no OMOS-relevant features
}

abstract class DeploymentTarget extends NamedElement
{
  // has no OMOS-relevant features
}

datatype Integer "int"

datatype Boolean "boolean"

datatype String "java.lang.String"

datatype UnlimitedNatural "int"

}
```

D.2. OMOS model editor

A OMOS model editor prototype was implemented as part of the research presented in this thesis (see Section 5.2). The editor allows to create OMOS models and diagrams. It implements the semi-automatic approach for laying out OMOS diagrams discussed in Chapter 5 and in this chapter. The editing capabilities provided by the editor are similar to those provided by other UML modelling tools. However, a unique feature of the editor is that it allows to create OMOS diagrams using the semi-automatic layout approach. The editor's capabilities are discussed in this section.

D.2.1. Creating class symbols

By means of creating symbols in diagrams, modellers can either add an existing class (which is already part of another diagram) to this diagram or create a new class. When a new class is created, the modeller selects the package to which it should belong. It is not possible to display a class more than once in a diagram. There is hence at most one class symbol of each class in a certain diagram. The position of the class symbol in the parent-child hierarchy has to be defined when a class symbol is added to a diagram. To do so, the modeller drags the class symbol to the left or right border of another class symbol. Doing so adds the new class symbol to children of the existing class symbol's parent class symbol (or, if the existing class symbol is a root one, the new class symbol is added to the diagram's root class symbols). The new class symbol is inserted before or after the existing class symbol which it gets dragged to — it is inserted on the left-hand side (i. e., before the symbol), if it was dragged to right border, and inserted at the right-hand side of the existing class symbol (i. e., after the symbol), if it was dragged to the symbol's left border.

If the UML class underlying the added class symbol already exists in the UML model, the model will not be modified. If the class is newly created, it will, of course, be added to the UML model.

D.2.1.1. Relocating class symbols

A OMOS diagram's class symbols can be relocated. A class symbol can be moved to a new parent node or reordered within its current parent's children. Similar to adding a class symbol to a diagram, this is done by dragging a class symbol to the left or right border of another class symbol. Moving class symbols in a diagram does not modify the underlying UML model.

D.2.1.2. Details visibility

For a class symbol, the OMOS editor prototype provides functionality for modellers to define which attributes and operation of a symbol's class are displayed for this symbol. They could chose between displaying all attribute and operations, only public ones, or none at all (then, only the class's name and its stereotypes are shown for the class symbol).

D.2.2. Creating connections symbols

In order for OMOS modellers to depict association or inheritance (UML) relationships, the diagram editor offers to create connections symbol depicting those relationships. Starting from a class symbol, a new relationship could be created by dragging the yet "dangling" end of the connection (which was created by the diagram editor to depict the relationship) to another class symbol. If there already exists a relation of the same type as the to-be-created one, the user can select one from a list of existing relationships (those already depicted in diagram are highlighted and cannot be selected). If an existing one is selected, no new *model* element is created, but the existing relationship is displayed in this diagram. It is not possible to display a certain relationship more than once in a certain diagram. There is hence at most one connection symbol of each UML relationship in a diagram.

D.2.2.1. Moving connection ends

The ends of connection symbols can be moved from one class symbol to other class symbol. Doing so, of course, changes the diagram, but also changes the underlying UML model. All other connection symbols in other diagrams depicting the changed model element will be *deleted* from the respective diagrams because the model changed and thus these connection symbols are outdated since the relationship they depicted no longer exists. The modified connection will be updated only in the diagram where the modeller modified the connection.

D.2.2.2. Automatic update of diagram symbols when model elements are updated

Diagram symbols are hence automatically updated if the underlying UML model changes. For instance, moving a generalisation connection's end from one class symbol to another will update the underlying UML model and by doing so *all* connection symbols depicting the old generalization will be deleted.

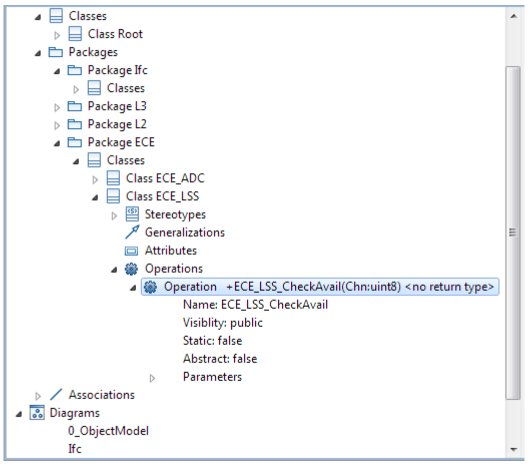
D.2.3. The model tree

All model elements are displayed in a modelling tree (see Section 5.2) shown in Fig. D.1.

The model tree allows modellers to navigate to diagrams and diagram symbols which visualise the respective model element. OMOS modellers can do so by making use of the *list of diagrams* which is discussed next.

D.2.3.1. List of diagrams depicting model elements

For each model element, a list of diagrams containing a symbol which depicts this element is provider by the modelling tool.



```
  Classes
    Class Root
  Packages
    Package Ifc
      Classes
    Package L3
    Package L2
    Package ECE
      Classes
        Class ECE_ADC
        Class ECE_LSS
          Stereotypes
          Generalizations
          Attributes
          Operations
            Operation   +ECE_LSS_CheckAvail(Chn:uint8) <no return type>
              Name: ECE_LSS_CheckAvail
              Visiblity: public
              Static: false
              Abstract: false
              Parameters
  Associations
Diagrams
  0_ObjectModel
  Ifc
```

Figure D.1.: OMOS model tree example.

D.2.3.2. Packages, attributes, and operations

Creating classes and association and inheritance relationships can only be done in a visual way by adding class symbols to diagrams. Because package assignment diagrams — the only kind of OMOS diagrams displaying package symbols (see Section C.1) — were not implemented by the modelling tool prototype, packages can be created directly in the model tree, however, they cannot be displayed in OMOS diagrams.

Creating, modifying, and deleting packages, attributes, and operations is accomplished via the model tree. Depending elements will also be deleted automatically. For instance, when a package is deleted, all its class and sub-packages and their depending elements are deleted. Since the model is modified, diagrams depicting deleted elements will be affected too (see Section D.2.4).

D.2.3.3. Deleting diagram symbols and model elements

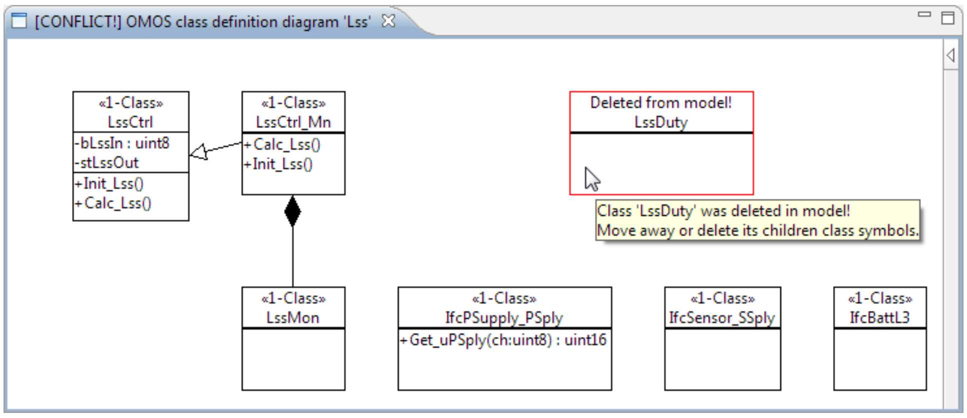
When a symbol is deleted in a diagram, the modeller can chose whether only the symbol or, in addition to the symbol, the underlying model element(s) are deleted. Deleting only the diagram symbol does not modify the model element it depicted. However, deleting also the model element(s) will delete all the model elements belonging to the depicted symbol from the model. This will also remove all symbols depicting the removed model elements from all diagrams. When a class is removed from the model, all its attributes, operations, stereotype assignments, association and generalisation relationships are removed, too.

The ability to automatically update a diagram's layout when its symbols were updated — directly in the diagram editor and/or by altering the underlying UML model — and still preserve the layout features of this diagram is one of the advantages of the approach to laying out OMOS diagrams developed by this research (see Chapter 5).

D.2.4. Diagram and model updates and automatic re-layout of all affected diagrams

In order to avoid orphaned model elements (i. e., classes and association and inheritance relationships) which are not depicted in any diagram any more but is still part of the underling model, deleting a symbol from a diagram which is not displayed in any other diagram will also remove the symbol's model element(s).

When a class is deleted from the model and thus all its class symbols are removed from all diagrams depicting the class, the deleted symbols' children class symbols are *not* deleted. Instead, in each diagram in which the deleted class symbol has children class symbols, it is replaced by a place-holder symbol which acts as a temporary parent symbol of these children symbols (see class symbol "LssDuty" in Fig. D.2). Each diagram which has such a place holder is marked with a error. The user then has to move the children class symbols to other parent class symbols or delete them too. This approach has been chosen in order to avoid recreating parts of a diagram in case a class (which was depicted in this diagram) is deleted from the model.


[CONFLICT!] OMOS class definition diagram 'Lss'
«1-Class» LssCtrl
-bLssIn : uint8
-stLssOut
+Init_Lss()
+Calc_Lss()
«1-Class» LssCtrl_Mn
+Calc_Lss()
+Init_Lss()
Deleted from model! LssDuty
«1-Class» LssMon
«1-Class» IfcPSupply_PSply
+Get_uPSply(ch:uint8) : uint16
«1-Class» IfcSensor_SSply
«1-Class» IfcBattL3

Class 'LssDuty' was deleted in model!
Move away or delete its children class symbols.

Figure D.2.: OMOS diagram with an error resulting from deleting class "LssDuty" from the model.

With respect to diagram editor tooling, diagrams have to be updated when the states of their diagram symbols are updated. For the OMOS diagram editor, symbol updates can occur because of the following reasons:

- Purely graphical updates affecting only a certain diagram, but not the underlying UML model. Those updates are interactively initiated by the modeller by modifying the graphical properties of symbols of a certain diagram, for instance, relocating a class symbol. The updates only affect the symbols of the respective diagram, but no other diagrams.
- Graphical updates which affect the underlying UML model and thus affect all diagrams depicting these model elements.
  - A connection symbol's end is relocated (this update does affect the underlying UML model since a diagram contains at most one symbol of a certain class; moving a connection end to another class symbol, therefore, also relocates the underlying model element(s) to a different class).
  - Deleting a class or connection symbol from a diagram and the respective class or relationship from the model. This will affect all diagrams depicting the respective model element.
- When modellers use the model tree to directly modify the UML model, the resulting model updates which affect all diagrams depicting the modified model elements. For instance, when the parameter list of an operation is modified, all class symbols depicting this operation have to be updated because their widths

might have changed. The width update in turn will require the respective diagrams to be newly laid out because changing the width of a class symbol requires rebalancing the class symbols. This in turn might cause modifications of the class symbols' position and the positions of connections.

Since a diagram's layout is only calculated and relevant when the diagram is actually opened in the modelling tool, only currently opened diagrams are affected by model updates and have to be laid out again.

D.3. Semi-automatic OMOS diagram layout details

This section discusses the semi-automatic approach for laying out OMOS diagrams presented in 5.

D.3.1. Overview

First, two meta-models for describing OMOS diagrams and an overview of the symbols used in OMOS diagrams are presented.

D.3.1.1. Depicting classes

Classes are depicted as rectangles. Each rectangle is divided into three compartments containing textual labels. The class's name and the names of the stereotypes assigned to it are displayed in the first compartment. The second and third compartment show labels of the class's attributes and operations, repsectively. Depending on which class details are shown for the symbol, these two latter compartments may be empty (i. e., show no details), depict only public attributes and operations, or all of them. In this section, these labels are referred as the *class details labels*.

D.3.1.2. Depicting connections

Connections visualise UML relationships between (two) UML classes, they are presented as poly-lines. A connection may have a decoration figure at its beginning and/or end: an open arrowhead for navigable associations, a filled diamond for composition associations, and an unfilled triangle for generalisations. For associations whose association's name is defined and/or whose upper or lower bound does not equal one, a text label depicting the name and/or upper and lower bound is part of the decoration, too.

D.3.1.3. A meta-model for OMOS diagram layout calculation

A meta-model used by the OMOS diagram editor and the OMOS merge tool to automatically calculate the layout of OMOS diagrams according to the approach defined in Chapter 5 has been defined as part of the research presented here. It describes the directed acyclic graph of class symbols and the set of connections belonging to each class symbol used to describe the principle structure/hierarchy of an OMOS diagram without describing the concrete graphical properties (i. e., position and size) of its symbols. The graph defines the class symbols' parent-children hierarchy (each parent class symbol has an ordered list of direct children symbols, all belong to the layer directly below its parent's layer) and the layers (or ranks) of the class symbols which are derived from the class symbol layout hierarchy manually defined by modellers. Like all meta-models provided by this research, this meta-model is based on the meta-object facility (see Section 2.1.2.1).

- Class symbols
  - Parent class symbol.
  - Children class symbols (ordered collection/sequence).
  - Inter-layer connections
    - Adjacent layer connection
      - Source connections (ordered collection/sequence).
      - Target connections (ordered collection/sequence).
    - Non-adjacent layer connections
      - Connections passing on left-hand side (ordered collection/sequence).
      - Connections passing right-hand side (ordered collection/sequence).
  - Intra-layer connections
    - Connections to adjacent class symbols
      - Source connections (ordered collection/sequence).

        · Target connections (ordered collection/sequence).

    * Connections to non-adjacent class symbols

        · Source connections (ordered collection/sequence).

        · Target connections (ordered collection/sequence).

This meta-model is used for creating the principal class symbol hierarchy and to define the routing of connection symbols. It is, however, not used to visually depict OMOS diagrams. This is done using another meta-model which is discussed in the next section.

D.3.1.4. A meta-model for OMOS diagram layout

In addition to the meta-model for describing the principle structure/hierarchy of class symbols discussed in the previous section, a meta-model (used by the OMOS model editor and the merge tool) for visualising (i. e., laying out) OMOS diagrams in an automatic manner was defined as part of the research presented here. This meta-model, too, is based on the meta-object facility (see Section 2.1.2.1).

- Class symbols
  - Reference to depicted class from the underlying UML model.
  - Size and position.
  - Bounding box (including the bounding boxes of connection decorations, the horizontal lines/segments of (incoming and outgoing) connections from/to non-adjacent class symbol's, and the vertical lines/segments of passing-by connections). (The bounding box is required for calculating the size of the layer to which the class symbol belongs.)
  - Reference to incoming, outgoing, and passing-by connection symbols.
- Connection symbol
  - Reference to depicted relationship from the underlying UML model.
  - Source and target location and bounding boxes of source and target decoration.
  - Bend point locations.

- Attribute symbol
  - Reference to depicted attribute from the underlying UML model.
- Operation symbol
  - Reference to depicted operation from the underlying UML model.
- Place holder symbol for deleted class symbols (see Fig. D.2).
- Diagram conflict state (derived from presence of place holder symbols; not related to merge conflicts).

D.3.2. Automatic layout of class symbols

This sections discusses the algorithm for laying out OMOS diagrams in an automatic way based on the meta-model for describing the principle class symbol hierarchy discussed in the previous section.

D.3.2.1. Calculating the sizes of class symbols

The calculation of the size of each class symbol consists of two steps: Step one calculates the size of class symbol. Step two calculate size of class symbol's *bounding box*. The latter step is required for visually separating the horizontal diagram layers. The directed acyclic graph of class symbols and a set of connections belonging to each class symbol serves as the input for both steps.

D.3.2.1.1. Calculating a class symbol's width and height:

- Input: A class symbol, its connections (separated in connections leaving at the top or bottom and at the left or right side), and its class details labels.
- Output: The given class symbol with its width and height assigned.
- Algorithm: Calculate class symbol width and height
  1. Assign given class symbol's width and height to default value (50 pixels and 80 pixels).
  2. Calculate the width and hight of the class details labels (since they influence the symbol's size).

3. Assign new width/height to the maximum of current width/height of the class details labels, i. e., if necessary, the class symbol is enlarged such that the text of all labels is displayed properly.
4. The size of the decoration of a class symbol's outgoing and incoming connections (for which the class symbol acts as its source or target, respectively) is taken into account. Inter- and intra-layer connections between non-adjacent class symbols connect to the class symbol at top or bottom; they therefore may influence the width of the class symbol. Intra-layer connections between adjacent class symbols connect to the class symbol at its side; they therefore may influence the height of the class symbol.
   a) Calculate the width and height of the class symbol's connections (default minimum size: 15 pixels wide and 15 pixels heigh).
   - The class symbol's new width is the maximum of (1) the current width and (2) the maximum of (a) the sum of the widths of all connections connected at the top of the class symbol and (b) the sum of the widths of all connections connected at the bottom of the class symbol.
   - The class symbol's new height is, therefore, the maximum of (1) the current height and (2) the maximum of (a) the sum of the heights of all connections connected at the left side of the class symbol and (b) the sum of the heights of all connections connected at the right side of the class symbol.

D.3.2.1.2. The bounding box of class and connection symbols: The reason why bounding boxes are required for laying out class and connection symbols is the balanced layout of class symbols. The bounding box is required to calculate the horizontal position of the diagram's class symbols. A class symbols bounding box takes into account all (left/right-hand side) decorations (of intra-layer connections between adjacent class symbols) and passing-by line segments; they add to the class symbol's width.

The bounding boxes of connections and their decorations are taken into account to prevent connection decorations from crossing with line segments of other connections. This approach ensures that if case line segments cross, they should cross the line instead of the decoration in order not to compromise readability of the decoration (which might be a label). This approach also prevents (straight vertical) line segments of passing-by connections from crossing the decorations of intra-layer connections between adjacent class symbols. It also prevents (straight vertical) line segments of intra-layer connections between non-adjacent class symbols from crossing with decorations of inter-layer connections between class symbols on adjacent layers (passing by connections do not have decorations).

D.3.2.2. Calculating a class symbol's position

This step in the OMOS layout approach determines the horizontal position of the diagram's class symbols. It creates a balanced tree of class symbols as outlined in Fig. D.3 according to the manually defined class symbol hierarchy:

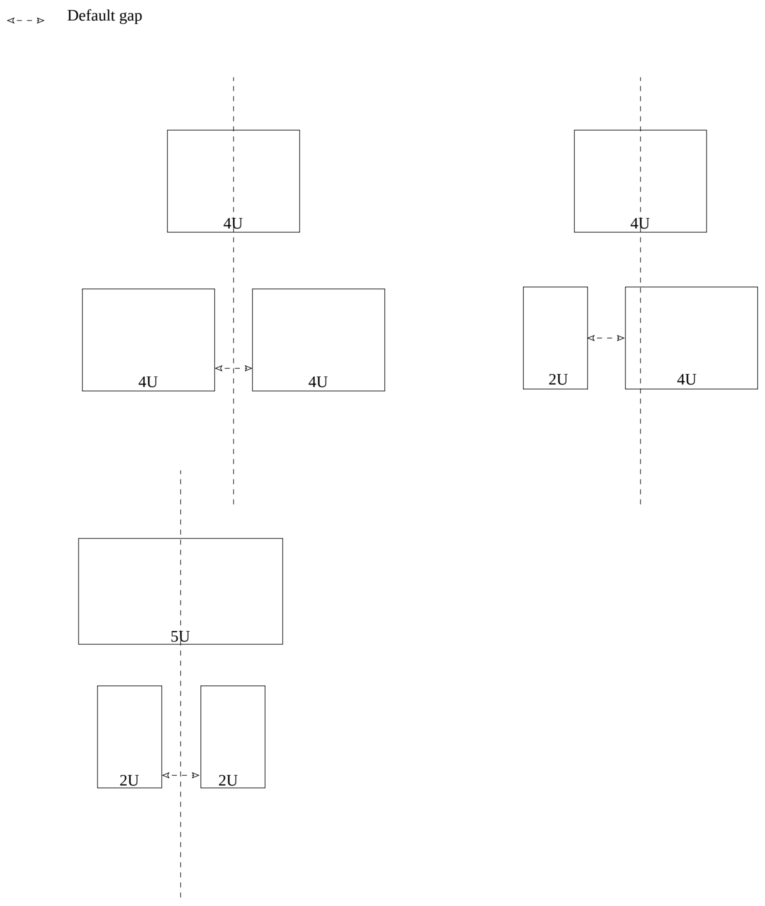


Figure D.3.: Balanced hierarchical class symbol layout.

D.3.2.2.1. Calculating a class symbol's horizontal (x) position

- Input: A diagram for which the width and height of all its class symbols assigned (see above).
- Result: The x position of all the diagram's symbols has been assigned.
- Algorithm: see Algorithm D.1.

Algorithm D.1 Laying out class symbols horizontally in a balanced manner.

```
Input: Class symbol cs
Input: Left x position leftX
Output: Class symbol tree's right x position
  if cs has childre class symbols then
    Layout children class symbols starting at given (leftX) x position
    {See "Laying out children class symbols horizontally" below.}
  end if
  $bbWidth$ := width of cs's bounding box
  if cs has no children class symbols then
    $treeWidth := bbWidth$
    $x := leftX$
  else
    {Position class symbol centred above its children:}
    $subtreeWidth$ := maximum of all class symbols' bounding box widths in the
    sub-tree formed by cs's children class symbols
    if $bbWidth > subtreeWidth$ then
      {cs is wider than sub-tree; balance sub-tree's class symbols:}
      $rightShift := bbWidth/2 - subtreeWidth/2$
      Recursively move sub-tree class symbols to the right by rightShift pixels.
      $x := leftX$
      $treeWidth := bbWidth$
    else
      {cs is smaller than sub-tree symbols; no children balancing needed, just po-
      sition it at the centre above children:}
      $x := leftX + (subtreeWidth/2) - (bbWidth/2)$
      $treeWidth := subtreeWidth$
    end if
  end if{Set cs's x position.}
  cs.x := $x$
  return  leftX + treeWidth;
```

Algorithm D.2 Laying out child class symbols horizontally.

```
Input: Ordered list of class symbols
Input: Left x position $leftX$
Output: None.
  $x$ := $leftX$
  for $cs$ ← each of given class symbols do
    {Assign $cs$'s horizontal (x) position:}
    $x$ := horizontalLayout($cs$, $x$)
    {Add default horizontal gap:}
    $x$ := $x$ + 50 {
  end for}
```

D.3.2.2.2. Calculating a class symbol's vertical (y) position

- Input: A diagram for which the width and height of all its class symbols is set (see above).
- Result: The *y* position of all the diagram's symbols is set
- Algorithm:
  - y := 0
  - For each layer
    - For all class symbols belonging to this layer:
      - Assign the *y* position to each class symbol.
    - *y* := maximum of (1) layer gap (40 pixels) and (2) the maximum height of the layer's class symbols' bounding box heights.

D.3.3. Automatic layout of connection symbols

D.3.3.1. Sorting connections symbols

Input:

- Class symbol graph (parent-child hierarchy; each parent class symbol has an ordered list of direct children symbols (all belong to the layer directly below its parent's layer)).

- Each class symbol's connection groups (each group contains all connections going to/coming from the same class symbol)
  - Outgoing inter-layer connection groups (outgoing: class symbol (source) on higher layer than the other (target) class symbol's layer).
  - Incoming inter-layer connection groups (incoming: class symbol (target) on lower layer than the other (source) class symbol's layer).
  - Outgoing intra-layer connection group to adjacent class symbol (single) (outgoing: class symbol (source) whose position order within the layer is lesser than the other (target) class symbol's, i. e., class symbol on left-hand side).
  - Incoming intra-layer connection group from adjacent class symbol (single) (incoming: class symbol (target) whose position order within the layer is greater than the other (source) class symbol's; i. e., class symbol on right-hand side).
  - Outgoing intra-layer connections groups to non-adjacent class symbols (outgoing: class symbol (source) whose position order within the layer is lesser than the other (target) class symbol's, i. e., class symbol on left-hand side).
  - Incoming intra-layer connections groups from non-adjacent class symbols (incoming: class symbol (target) whose position order within the layer is greater than the other (source) class symbol's; i. e., class symbol on right-hand side).
- Notes:
  - The input is a directed acyclic graph of class symbols and a set of connections belonging to each class symbol.
  - The layer (or rank) of class symbol is known due to its position in the class symbol hierarchy/graph.
  - The class symbols' size and position are not relevant at this stage.

Determining the sort order of connections:

- Calculate passing by connection groups (Note: there are only incoming ones since a passing-by connection is presented as a vertical line (i. e., passing-by segment)).

- Sort the connection groups.
  - Outgoing inter-layer connection groups (result: sorted group list): The outgoing inter-layer connections of a source class symbol are sorted according to the target class symbol's order within its layer. The connection group with the leftmost target class symbol becomes the first in the list and so on. Since target class symbols can belong to different layers, their order could be identical to the order within their layers. Then, the connection whose target class symbol is on a higher layer is ordered before the connection going to a target class symbol on a lower layer.
    - Two criteria for ordering connection are applied: the horizontal order of source class symbol serves as the first ordering criterion. A second criterion is applied if the first one yields identical results for different target class symbols. Then, connections are sorted according to their target class symbol's layer: a connection going to a symbol on a higher layer is sorted before a connection whose target class symbol is on a lower layer.
    - Rational: The reason for this approach to ordering connection symbols is to avoid connection crossings as much as possible.
  - Incoming inter-layer connection groups: Same algorithm as applied for outgoing inter-layer connection groups of source class symbol is applied; but instead of using the target class symbol for determining the sort order, the source class symbol is used.
  - Outgoing intra-layer connection group to adjacent class symbol: Only a single group of connections has to be taken into account since there is at most one adjacent class symbol on the same layer for which the class symbol has outgoing connections. The sort order of the connections of this group is discussed below.
  - Incoming intra-layer connection group to adjacent class symbol: Only a single group of connections has to be taken into account since there is at most one adjacent class symbol on the same layer for which the class symbol has incoming connections. The sort order of the connections of this group is discussed below.
  - Outgoing intra-layer connections groups to non-adjacent class symbols: The outgoing intra-layer connections of a source class symbol are sorted

according to the target class symbol's order within its layer (in the same way as used for the respective source class symbol). The target class symbols' layer order can only be greater (i. e., further to the right) than the (source) class symbol. The connection group whose target class symbol has with lowest greater layer order (i. e., is closest to the source class symbol) is first in the list and so on.

  - Incoming intra-layer connections groups to non-adjacent class symbols: The same algorithm as applied for outgoing intra-layer connection groups is applied to non-adjacent class symbols of source class symbol. However, instead of using the target class symbol for determining the sort order, the source class symbol is used.
  - Passing-by connection groups: They are ordered in the same way as outgoing inter-layer connection groups discussed above.

- Sorting within connection groups: The original order is kept, but the generalisation connection (there can at most be one per pair of class symbols, i. e., in a connection group) comes first (during the actual diagram layout it will be positioned closer to the centre of the source/target class node).

Result:

- The connections in each class symbol's connection groups are sorted according to the criteria shown above.

D.3.3.2. Laying out connections symbols

The layout process of connection symbols is based on the layout of the class symbols (described above) which has already been done when connections symbols are laid out. The calculation of the class symbols' width and hight already took into account the connection symbols belonging to the class symbol. Therefore the connection can be positioned at the class symbol without the need to resize it. Furthermore, the gaps between layers, i. e., the gap between the bottom and the top of class symbols on adjacent layers, are calculated such that connections do not overlap with class symbols.

D.4. Design and implementation of MOF-based model comparison

This section discusses the details of the meta-object facility (MOF) which are relevant for differencing and merging MOF-based models.

D.4.1. Introduction

MOF provides elements[4] for defining (see Fig. D.4):

- Packages (used for scoping elements);
- Classes (used to define the structure and behaviour of instances of classes, i. e., objects)
    - Data type properties (to define primitive, unstructured properties of classes);
    - Class type properties (for referencing instances of classes, i. e., for building networks of objects, i. e., models);
    - Operations (for declaring behavioural aspects of objects);
    - Inheritance (for reusing/extending existing classes).

To quote from section 2.3 of the Human-Usable Textual Notation Specification [OMG04]: *"The Meta-Object Facility (MOF) specifies a small but complete set of modeling concepts that can be used to express information models. [...] There are a number of essential concepts used in MOF modeling. A Package is used to encapsulate a collection of related Classes and Associations. [...] Classes exist in the commonly-used sense of the word, describing an object and its properties. These properties are represented through Attributes and References, which can be inherited using a multiple-inheritance system based on that of CORBA IDL. Attributes have a name and a type, selected from the CORBA type system1. This includes a range of types from basic types such as integers, strings, and booleans to more complex types such as enumerations, and through to structured types. In addition, attributes have both upper and lower limits on the number of times that they can appear within a*

[4]All these elements are MOF elements. Even though UML meta-model has elements with similar names, they are not the same. The MOF elements are, in (meta-) modelling terms, one level below (M3) the UML meta-model elements (M2). All elements of the UML meta-model are instances of elements defined by MOF.

class instance. An Association is used to represent a relationship between instances of two classes, each of which plays a role within the association. Associations can have the additional property of containment; an association represents a containment relationship if one of the participant classes does not exist outside the scope of the other. A Class participating in an association can also contain a Reference to the association. A reference appears much like an attribute, but reflects the set of class instances that participate in the Association with the containing class instance."

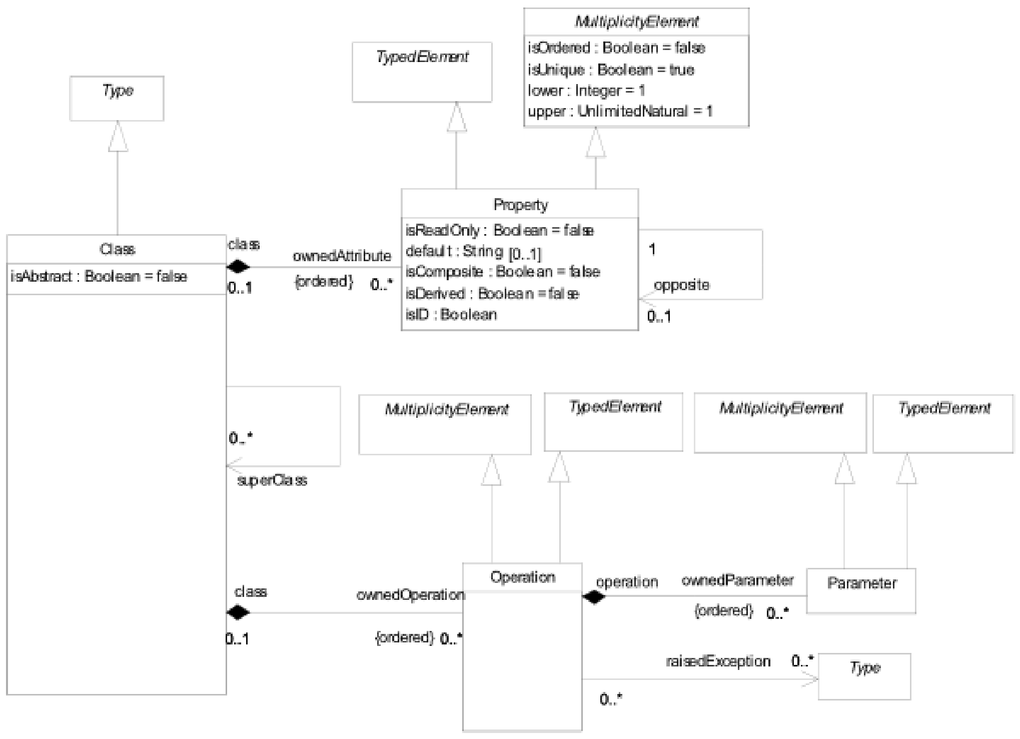


Figure D.4.: Relations, attributes, and operations of MOF classes ([OMG06a, Fig. 12.2, p. 33]).

The UML Infrastructure Specification describes the relation between UML and MOF as follows [OMG10a, p. 14]: *"[...] UML is defined as a model that is based on MOF used as a metamodel [...]. Note that MOF is used as the metamodel for not only UML, but also for other languages such as CWM. [...] An important aspect that deserves mentioning here is that every model element of UML is an instance of exactly one model element in MOF."*

D.4.1.1. Property features

Properties (instances of MOF class *Property*) have several features (which, due to the recursive nature of meta-meta-models, are instances of MOF class *Property* too) which are important for merging models. These features are discussed next and referred to in other sections discussing the model merge process.

D.4.1.2. Attribute and reference properties

As explained in Section 2.1.2.1, there exist two categories of properties (instances of MOF class *Property*, see [OMG06a, p. 36]): *Data type* (or *attribute*) *properties* (see [OMG06a, p. 17, p. 36]) represent values of simple, unstructured types. These types do not represent model elements. For instance, the *name* property of *NamedElement*, a MOF class (instance) which is defined by the UML meta-model, is such an unstructured one as its value is a string. The second property category are *class type* (or *reference*) *properties* (see [OMG06a, p. 36]) representing references to model elements. The *ownedOperation* property (defined by MOF class instance *Class* in the UML meta-model) is such a one, its values are operations, i. e., instances of (UML) class *Operation* (defined by the UML meta-model).

D.4.1.3. Name and type

Each property has a *name* and a *type*. The name is used to distinguish different properties (belonging to the same instance of a MOF class, for instance, UML's *Operation* class). The type is important for distinguishing between *attribute* and *reference properties* and for defining which values an instance of the property can hold as explained above.

D.4.1.4. Single- and multi-valued properties

Properties can either be *single-* or *multi-valued*. For instance, in the UML meta-model, the *NamedElement::name* property is a single-valued one, while the *Class::ownedOperation* property is a multi-valued one since a class in a UML model may have many operations. Whether a property is single- or multi-valued depends on the values of its *lowerBound* and *upperBound* features.

D.4.1.5. Order and unordered properties

Another important feature of properties is their *ordering.* For ordered properties the order of their values is relevant. For instance, the *ownedParameter* of UML class *Operation* is ordered (see [OMG10a, p. 97]) because in UML the order of an operation's parameters matters (as it often does for programming languages too). The *packagedElement* property of UML class *Package* is not ordered (see [OMG10b, p. 110]) because the order of the classes belonging to a package is not relevant.

For models which are instance of MOF-based meta-models, MOF defines how (the values of instances) of properties can be accessed and how they behave [OMG06a, p. 16]: *"If the Property has multiplicity upper bound of 1, get() returns the value of the Property. If Property has multiplicity upper bound >1, get() returns a ReflectiveSequence containing the values of the Property. If there are no values, the ReflectiveSequence returned is empty."*

The value of a single-value property hence is simply its value, i. e. either a attribute value (for attribute properties) or a model element (for reference properties). Depending on whether the property is ordered, the value of a multi-valued property is either an instance of *ReflectiveCollection*s or an instance of *ReflectiveSequence*s [OMG06a, pp. 24]. Both instances provide access to the actual values of the property (either an attribute value for attribute properties or a model element for reference properties). *ReflectiveCollection*, which is used for unordered properties, simply allows to add, access (traverse), and remove (by value) its values. In addition to the value-based access provided by *ReflectiveCollection*, its specialisation (sub-class) *ReflectiveSequence* used for ordered properties allows — since the order of values is relevant — *index-based* access, addition, and removal of values.

D.4.1.6. Default values and unset properties

In addition to setting (manipulating) a property's value, it is also possible to unset it [OMG06a, pp. 17]: *"If the Property has multiplicity upper bound of 1, unset() atomically sets the value of the Property to its default value for DataType type properties and null for Class type properties. If Property has multiplicity upper bound >1, unset() clears the ReflectiveSequence of values of the Property."*

An important feature for properties with respect to comparing model elements is the ability to determine whether the value of a property is set or not (unset). Models created from MOF-based meta-models thus allow to detect whether a property's value is set [OMG06a, pp. 17]: *"If the Property has multiplicity upper bound of 1, isSet() returns true if the value of the Property is different than the default value of that property. If Property has multiplicity upper bound >1, isSet() returns true if the number of objects in the list is > 0."*

D.4.1.7. Unique and non-unique properties

MOF also provides a feature for meta-models to define the *uniqueness* of multi-value properties. While unique properties may hold a certain value not more than once, their non-unique counterparts may hold a certain values multiple times. The UML meta-model defines four non-unique attribute properties — *OpaqueExpression::body*, *OpaqueBehavior::body*, *FunctionBehavior::body*, and *OpaqueAction::body*; all of them are unordered and none of them is relevant for OMOS models (see Section D.1). No non-unique reference properties are defined in the UML meta-model because it does not make sense for UML reference properties to reference the same model element more than once by the same property.

D.4.1.8. Containment properties and the model element containment hierarchy

MOF allows to define *containment* reference properties; this is accomplished by using Property's *isComposite* feature. This way containment hierarchies can be defined. Model elements belonging to a containment reference are "owned" by the model element to which this reference belongs. For instance, in UML each model element, except for the root *Model* element, must be owned by exactly one other model element, called its container (see [OMG10a, p. 74]). The elements of a UML model thus define a strict containment hierarchy. For example, packages (instances of UML class *Package*) contain classes (instances of UML class *Class*), classes contain operations (instances of UML class *Operation*) and so on. The containment of operations inside classes is defined by reference property *ownedOperation* defined for *Class* in the UML meta-model as mentioned above. This reference property is a (multi-values) containment reference.

Non-containment references are used to reference model elements without containing them. In UML, an attribute's type (reference property *TypedElement::type* of UML class *Attriubte*) is, for instance, defined by referencing another type (for instance, another class) in the model. Which model element actually contains the referenced type (model element) does not matter for defining the attribute's type.

D.4.1.9. Bidirectional properties

MOF allows to define *bidirectional* reference properties; this is accomplished by Property's *opposite* feature (see [OMG10a, pp. 33]). Since the values of bidirectional properties reference each other, bi-directionality makes only sense for reference properties. Then, each model element referenced from the opposite property also references the referencing model element with its property (which is the opposite of the former property).

D.4.1.10. Derived properties

MOF allows to define so-called *derived* properties via the *isDerived* feature (see [OMG10a, pp. 36]). The values of such properties are derived from the values of other (non-derived) properties. For instance, the *qualifiedName* property defined for UML meta-model class *NamedElement* (see [OMG10a, p. 71]) is a derived property, its value is created from the *name* property's value of the *NamedElement* and the *qualifiedName* value of the container element — which is, again, calculated in the same way. The values of derived properties cannot be changed directly. They are thus not relevant with respect to comparing model elements.

D.4.2. Implementation

The software tools implemented as part of this research use Ecore [SBPM09], the defacto standard implementation of the meta-object facility (MOF), for implementing tools for UML class model and diagram comparison and merging. Therefore, Ecore terminology (with respect to element names and features) will be used to discuss the implementation of UML model and diagram comparison. Since Ecore and MOF are closely related, the element names are similar to those defined by MOF (see Section D.4.1.1).

No matter what UML or diagram element types have to be compared, the type's structure is defined by means of MOF/Ecore concepts. The inner structure of those elements is then described by the MOF/Ecore meta-model itself: An *EClass* has structural features (of abstract type *EStructuralFeature*), which are either attributes (instances of *EAttribute*) or references (instances of *EReference*). An *EClass* can also have super types (instances of *EClass*).

Therefore, when dealing with a UML or diagram model element (i. e., an instance of the UML or diagram meta-model), its structure is described in terms of MOF/Ecore elements. For instance, *Class* from the UML (meta-model) has an *EAttribute* (inherited from its super-class *NamedElement*) called *name* which defines that each Class has a name of type *String*:

```
unordered unsettable attribute String name (0..1);
```

Therefore, instead of comparing a *Class*'s specific attribute *name* and other specific attributes, its sufficient to compare *all* its attributes. A custom comparison for class names is not necessary. It is thus not even required for the model comparison to be aware that it deals with instances of UML meta-class *Class*. Thus, model comparison can be based on the information defined by the meta-meta-model since it provides the mechanisms for accessing model elements (defined by a meta-model).

D.4.3. Introduction to EMFText as a concrete syntax for Ecore

A textual syntax called EMFText[5] is used to present Ecore-based meta-models in this thesis. It is a concrete syntax for Ecore. The following relation between EMFText keywords and Ecore elements exists:

EMFText keyword	Instance of Ecore element
`package`	`EPackage`
`class`	`EClass`
`abstract class`	`EClass` with attribute “abstract” set to “true”
`interface`	`EClass` with attributes “abstract” and “interface” set to “true”
`enum`	`EEnum`
`datatype`	`EDataType`
`attribute`	`EAttribute`
`reference`	`EReference`
`operation`	`EOperation`

Table D.1.: Mapping from EMFText keywords to Ecore elements.

[5]http://emftext.org/ (access date: 4/12/2012)

Instantiating an Ecore package (instance of `EPackage`):

```
package <package name> <namespace prefix> <namespace prefix>
{
        // ...
}
```

Example:

```
package Ecore ecore "http://www.eclipse.org/emf/2002/Ecore" { ... }
```

Instantiating an Ecore class (instance of `EClass`):

```
[abstract] class <class name> [extends <supertype [, <supertype>]*]
{
        // ...
}
```

Example:

```
abstract class ENamedElement extends EModelElement {...}
```

Instantiating an Ecore attribute (instance of `EAttribute`):

```
attribute <type> <attribute name> [= "<default value>"]
        [(<lower multiplicity>..<upper multiplicity>)];
```

Examples:

- `attribute EString name = "not set" (1..1);`
- `attribute EBoolean serializable = "true" (0..1);`

Instantiating an Ecore reference (instance of `EReference`):

```
[containment] [unordered] reference <type> <reference name>
    [(<lower multiplicity>..<upper multiplicity>)]
    [opposite <name of opposite reference>];
```

- `containment`: Referenced element is contained by referencing element, the latter is the parent element of the former.
- `opposite`: Defines the other/opposite end of a bi-directional reference.

Examples:

- `reference EPackage eSubpackages (0..-1) opposite eSuperPackage;`
- `reference EPackage eSuperPackage (0..1)opposite eSubpackages;`
- `reference ETypeParameter eTypeParameters (0..-1);`

D.4.4. Foundations of MOF/Ecore (meta-) models and model comparison

From a *meta*-modelling point of view, the main concept for creating meta-models with Ecore [SBPM09] are *EClass*es (see Fig. D.5 taken from the EMF website[6]). They define concepts (or "things") in the domain described by the meta-model, for instance, UML classes, packages, operations, associations, generalization, etc. from the software modelling domain. From a *modelling* point of view, Ecore-/MOF-based models consists of model elements which are instances of *non-abstract* and *non-interface EClass*es (called *Class*es in MOF) defined by the meta-model underlying the model. The model elements present actual *instances* of the concepts defined by their meta-model. For instance, package *Transportation* and class *Car*.

A model element is thus similar to the concept of an *entity* in information modelling [SM88], i. e., it describes a "thing" in the domain defined by the meta-model the entity belongs to. One model element can be distinguished from another, even if they have identical types (i. e., are instance of the same *EClass*) and identical structure (i. e., the values of their features are identical) — for instance, the same name (if such a feature was defined by the meta-model).

For comparing models, the state of model elements is most important (see Section D.5.1). On the meta-model level, the state an instance of a EClass can hold is defined by the *EClass*'s *EStructuralFeature*s, i. e., its attributes and references (and those inherited from its super *EClass*es) which are instances of *EAttribute* and *EReference*, respectively. Hence, in order to compare model elements, the values of their features have to be compared. For instance, when comparing two versions of a Class, the two values of its feature "name" (and other features) have to be are compared.

Since MOF/Ecore are *object-oriented* modelling approaches, in addition to their state, the behaviour of *EClass*es can be defined too — since the object-oriented

[6]http://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/package-summary.html (access date: 4/12/2012)

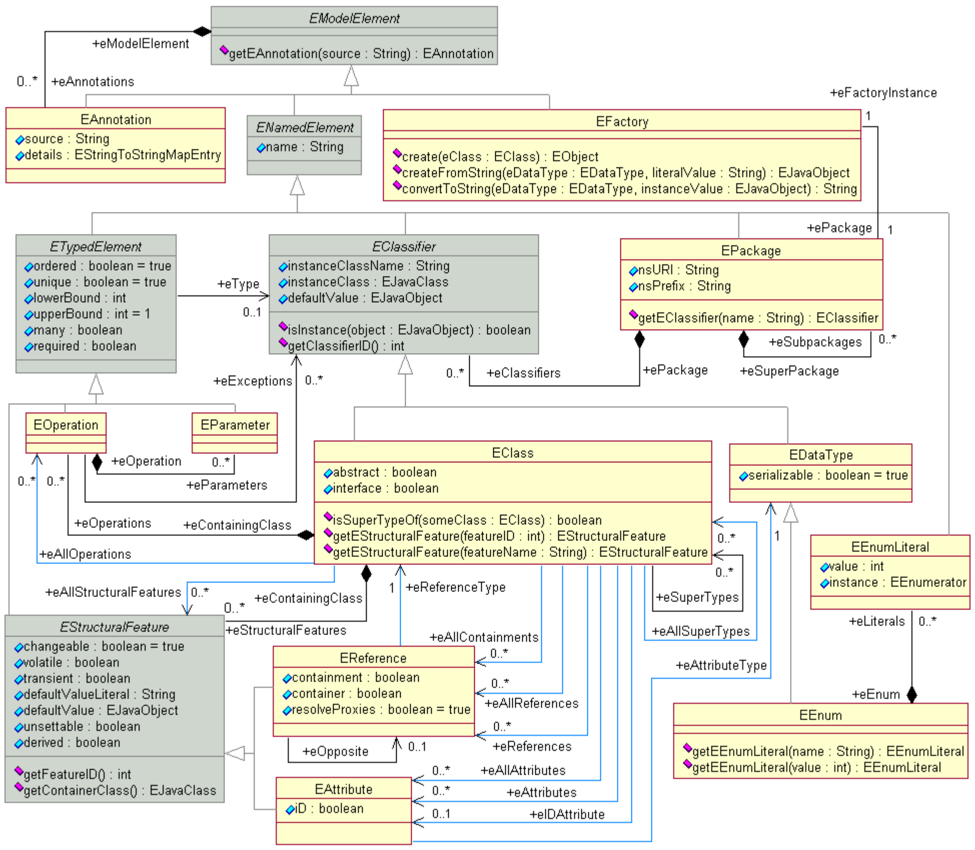


Figure D.5.: Relations, attributes, and operations of Ecore's classes.

paradigm advocates for coupling state and the behaviour which operates on this state [Mey97]. In Ecore, behaviour is defined by means of *EOperations*. In contrast to features, behaviour is irrelevant for comparing model elements since it does not hold state.

Thus, with respect to defining a model's state, features are most important as their values are persisted and preserved throughout the life-cycle of a model, i. e., a model's elements are created by a computer program, stored (to, for instance, a disk or database), restored back into a computer program and so one. Instances of operations are not persisted since they "live" on a model's *definition* (i. e., meta-model) level whereas instances of features "live" on the *instance* level.

In contrast to instances of *EClass*es, instances of *EDataType*s and *EEnum*s are not considered model elements in the sense that they represent entities in the domain defined by the meta-model. Merks et al. [SBPM09, p. 116] define them as concepts without an inner structure: "*data types represent a single piece of 'simple data'.*" The can only become part of a model as the value of attributes (i. e., instances of *EAttributes*) belonging to model elements (instances of EClasses). For instance, the name of an Operation belongs to the Operation (model element) in the sense that it defines the Operation's state, but the name itself is not a model element on its own.

An instance of an *EDataType* or *EEnum* (which is a special kind of *EDataType*) is identified by its value. For instance, there is no difference between the "name" attribute's value "C" for all the Classes named "C" — all the Classes have the same name.

The values of attributes may only be instances of data types (instances of *EClass EDataType*) and (its subclass) enumerations (instances of *EClass EEnum*). This is a consequence of *EClass EAttriubte*'s "eAttributeType" reference which is of type *EDataType* (of which EEnum is a subclass):

```
derived unchangeable volatile transient
        reference EDataType eAttributeType (1..1);
```

For instance, UML's *NamedElement* defines an *EAttribute* "name" of type String, and the visibility of *NamedElement* instances like packages, classes, or operations is defined by *NamedElement*'s "visibility" *EAttribute* which is of type *VisibilityKind* and is an instance of *EClass EEnum* (see Section D.1):

```
abstract class NamedElement extends Element
{
        unordered unsettable attribute String name (0..1);
        unordered unsettable attribute VisibilityKind visibility
                = "public" (0..1);
        // more features
}

enum VisibilityKind {
        0 : public = "public";
        1 : private = "private";
        2 : protected = "protected";
        3 : package = "package";
}
```

The possible values of attributes whose type is an instance of *EEnum* are hence defined by the meta-model — these values are the enumeration's *EEnumLiterals*.

In contrast to *EAttributes*, *EReferences* are used to define structured state as they reference instances of *EClass*es, i. e., model elements. This is a consequence of *EClass EReference*'s “eReferenceType” reference being of type *EClass*:

```
derived unchangeable volatile transient
        reference EClass eReferenceType (1..1);
```

For instance, the UML meta-model defines for EClass Package an EReference called “packagedElement” which is of type PackageableElement which is an EClass also defined by the UML meta-model:

```
class Package extends
        Namespace, PackageableElement, TemplateableElement
{
        unordered containment reference
                PackageableElement packagedElement (0..-1);
        // more features
}

abstract class PackageableElement
        extends NamedElement, ParameterableElement
{
}
```

D.4.5. Ecore structural features

Similar to the meta-object facility (MOF), Ecore defines EStructuralFeature, EAttribute, and EReference as the state-defining property of meta-models. There details are discussed in this section:

D.4.5.1. EStructuralFeature

- name (EAttribute): The name of the feature.
- eType (EReference): The type of the feature's value(s); it has to be an instance of an EClassifier, which is the common super-class of EDataType and EClass (which are the required types for EAttribute and EReference, respectively, see below).

- lowerBound (EAttribute): Defines the minimum number of values (multiplicity) hold by an instance of this feature; it has to be equal or greater than 0.
- upperBound (EAttribute): Defines the maximum number of values (multiplicity) hold by an instance of this feature; it has to be greater than 0 and greater or equal to the feature's lower bound. -1 is used to indicate unbounded upper bound multiplicity (only the upper bound of a feature may be unbounded).
- many (EAttribute): If a feature's upper bound multiplicity is larger than one, this feature is called *multi-valued* (see Section D.4.1.1) or multiplicity-many since it can hold more than one value. The actual values are stored in a list (called *EList* for Ecore and *ReflectiveCollection* or *ReflectiveSequence* for MOF [OMG06a, p. 28]).
- required (EAttribute): A feature's lower bound multiplicity is equal or larger than one.
- unique (EAttribute): The values of this feature are distinct, i. e. a single value may only occur once for an instance of a unique feature. Uniqueness is only relevant for multiplicity-many features.
- ordered (EAttribute): The order of values is relevant (in the domain defined by the meta-model). Ordering is only relevant for multiplicity-many features.
- changeable (EAttribute): Defines whether the feature's value can be set or not. If a feature cannot be set, its state cannot change. Unchangeable features are therefore irrelevant for model comparison.
- unsettable (EAttribute): These features have "*an additional possible state, called* unset*, that is distinguishable from that of being set to any value at all*" [SBPM09, p. 108]. Since the unset state is an additional possible state, unsettable features have to be taken into account for model comparison.
- transient (EAttribute): The values of transient features are not serialised when the model is persisted (i. e. stored in a file or database). Since models need to be de-serialised in order to compare them, transient features have to be taken into account for model comparison.
- derived (EAttribute): The values of derived features are calculated from the value(s) of other features. However, when comparing model elements, derived features *cannot* be ignored and thus have to be treated as ordinary features.

For instance, in Ecore, the “many” EAttribute defined by ETypedElement (a super-class of EStructuralFeature) is a derived feature: Its value is calculated from the value of the “upperBound” EAttribute (also defined by ETypedElement).
Derived features are often also transient and/or unchangeable. For instance, all derived features of the UML meta-model are also transient. (This was ascertained by querying all EClass instances of the UML meta-model for derived EFeatures.)

- volatile (EAttribute): Similar to derived features, volatile features do not have state directly associated with them. However, in contrast to derived features, volatile ones are not guaranteed to be directly derived from other features. Hence they have to be taken into account for model comparison.

D.4.5.2. EAttribute

The following properties are only defined for attributes, not for references (see below):

- iD (EAttribute): The value of this attribute can be used to identify the model element it belongs to in a model. Of all attributes of an EClass, only one may act as an identifying attribute. With respect to an EClass’s inheritance hierarchies (its direct and indirect super-classes), there can only be one identifying attribute within this hierarchy.
- eAttributeType (EReference): This derived reference casts the “eType” EReference inherited from EStructuralFeature into EDataType. This reference ensures that an instance of EAttribute can only reference instances of EDataTypes.

D.4.5.3. EReference

The following properties are only defined for references, not for attributes (see above):

- eReferenceType (EReference): This derived reference casts the “eType” EReference inherited from EStructuralFeature into EClass (similar to EAttribute’s “eAttributeType”). This reference ensures that an instance of EReference can only reference instances of EClasses.

- containment (EAttribute): Instances of the EClass (to which this reference belongs) contain the referenced model elements (instance of EClasses) in the sense of a whole-part relationship. "*[A]n object cannot, directly or indirectly, contain its own container; it can have no more than one container; and its life span ends with that of its container*" [SBPM09, p. 111]. Multiplicity-many containment references have to be unique since they cannot own model elements more than ones.
- eOpposite (EReference): This EReference is used to establish bidirectional associations, it refers to the EReference pointing into the opposite direction of the bidirectional association. Each EReference of such an association defines the other EReference as its "eOpposite." Since a model element can only have one container, the opposite reference of containment references (so-called container references, see next) must have an upper bound of one and, therefore, cannot be multiplicity-many references.
- container (EAttribute): If a containment EReference takes part in an bidirectional association (see above), the opposite EReference's "container" EAttribute's value is true. This EAttribute is a derived one.

Like for MOF properties, *null* values only allowed for single-valued Ecore features but not for many-valued ones [SBPM09, p. 11]. It follows that many-valued references cannot have "holes" (i. e., cannot hold null values).

Reflection-based model comparison MOF and Ecore define *Element* and *EObject*, respectively, which allow to access the type, i. e., meta-class (*EClass*), defining the model element at hand. Since they are the implicit root super-class of each model type, each model element's type can be accessed – no matter if the model element is an instance of a UML model element like a class, package, or operation, or if the model element is an element of the UML meta-model itself.

Instead of implementing model comparison based on the UML meta-model, this research implemented comparison directly based on the Ecore implementation of the meta-object facility (MOF). Since each model element's Ecore type is accessible, the whole meta-model is accessible without actually having to know the meta-model before hand. One can reflectively access all concepts defined in the unknown meta-model using MOF/Ecore concepts, be it *EClass*es, super types (again *EClasses*) thereof, features (*EAttribute*s and *EReference*s) and so on. Since the meta-model is build from instances of the MOF/Ecore meta-meta-model. So given a model which adheres to a certain known or unknown meta-model, for instance, the UML meta-model, full access to the meta-models elements (or concepts) is available by reflection using elements defined by the MOF/Ecore meta-meta-model:

```
modelElement.eClass.eAllStructuralFeatures
```

D.4.5.4. Assigning identifiers to UML model elements

Assigning identifiers to model elements can be achieved by means of the following two approaches: Ecore annotations or UML profiles (see Section 2.1.2.2). Annotations represent a simple mechanism for associating a set of annotation details (key-value pairs) to an annotation key (a string). UML profiles allow to extend UML meta-model elements. In order to add identifier attributes the UML model elements, a profile could be created that adds such an attribute to the UML meta-model's most basic EClass, namely Element.

Ecore annotations are defined in the Ecore meta-model by EClass EObject. Since EObject is the (implicit) super type of all EClasses and all their instances, any model element, i. e. any instance of an EObject, defined by any Ecore-based meta-model can have annotations (instances of EAnnotation) [SBPM09, p. 119] "*Annotations constitute a mechanism by which additional information can be attached to any object in an Ecore model.*"

Since Ecore and MOF are related, a concept similar to annotations exists for MOF too. This MOF concept is called MOF Extension: "[...] *it is sometimes necessary to dynamically annotate model elements with additional, perhaps unanticipated, information. This information could include information missing from the model, or data required by a particular tool. The MOF Extension capability provides a simple mechanism to associate a collection of name-value pairs with model elements in order to address this need.* [...] *They are included to reduce the need to redefine metamodels in order to provide simple, dynamic extensions*" [OMG06a, p. 27].

EMF Profiles [LWWC11] is a (prototypical) approach to adapting UML's profile concept to Ecore-based meta-models.

It was decided to use Ecore annotations (for implementing the software tools discussed in this chapter) because they provided sufficient mechanisms for assigning and assessing model element identifiers.

D.5. Foundations of model comparison

This section discusses the foundations of model comparison as used by the approach to merging OMOS models implemented as part of this research and discussed in Chapter 6.

D.5.1. Meta-models and state- and change-based model comparison

As explained in Section 2.1.3, OMOS *models* are ordinary UML models. based on the UML meta-model. UML, however, does not provide a standard for diagrams. It does define an approach — called UML Diagram Interchange [OMG06c] — for *exchanging* diagramming data *between* modelling tools. A proposal for standardising diagram definitions has been published [OMG11]; however, at the time of writing this (November 2012), is was still in beta version and has not been officially released. Of course a generic approach for defining the content of diagrams, i. e., providing a meta-model for diagrams, cannot provide the specific information required for OMOS diagrams proposed and implemented by this research. This is because OMOS diagrams define an explicit parent-child hierarchy for the class symbols they contain. So even though the (with respect to their layout features like the symbol's position and size) ordinary diagram symbols are used for OMOS diagrams, they contain information regarding the class symbols' parent-child hierarchy and the visibility of a attributes and operations.

Therefore, a specific meta-model for OMOS diagrams was developed by this research (see Section D.7.1.1). This meta-model is used by the prototypical modelling tool which was developed as part of this research to support creating OMOS diagrams (and the underlying UML models). The meta-model is also used for calculating differences between two versions of a diagram and for merging two evolved versions of a diagram (which is discussed in Section 6.4).

Of course, the UML meta-model and the meta-model for OMOS diagrams are completely different meta-models. However, both are defined using a meta-meta-model, namely the meta-object facility (MOF, see Section 2.1.2.1).

D.5.2. Preliminaries

For the approach to differencing and merging OMOS models presented in this thesis, the following preliminaries are expected to be met:

D.5.2.1. Stable meta-models and profiles

It is expected that the OMOS models involved in the merge process (i. e., the ancestor model and the two evolved models) are based on identical meta-models. This precondition holds since OMOS models are based on the UML meta-model. In case the meta-model would change, all three models would need to be migrated to the new meta-model before they are merged.

A similar precondition exists for UML profiles which are used by OMOS models. When a certain profile is applied in any of the models has, it has to be ensured that the other two models apply the same version of this profile (if they apply it at all). If the profile changes, all of the models applying it have to be migrated before they are merged.

The preliminaries above guarantee that the models can be merged and that the merged models will then be based on the same meta-model as the input models.

D.5.2.2. Valid models

It is expected that the OMOS models involved in the merge process are valid according to the fundamental rules defined by the meta-object facility (MOF), i. e., every model element (except for the root elements) has a container. Merged models will then adhere to the same fundamental MOF rules (Chapter 6).

D.5.3. Model element identification and matching equivalent model elements

When comparing models, there needs to be a way of determining whether a certain model element from one model exists in another model and, if it exists, get access to it in order to compare both elements. Thus, there needs to be some mechanism for identifying *equivalent* model elements. Most importantly, this mechanism has to support identifying equivalent elements across models. It is not sufficient to identify an element within its model. The element has to be identifiable in other models, too, in order to compare different versions of the element existing in the different models. Model elements hence need to be globally — meaning across models — identifiable in order to be able to access them in different models and compare them. The approach taken by this research to achieve this is discussed in the next section.

D.5.3.1. Model element identification and matching equivalent model elements

Names of elements might change and elements might move between owners. It is, therefore, insufficient to base the identification of elements on their fully-qualified names. Many meta-model's define name attributes for assigning names to model elements. For instance, class *NamedElement* from the UML meta-model defines a *name* attribute and meta-object facility's (MOF) meta-model defines a similar MOF class *NamedElement* — both classes represent super-classes of most (meta-) meta-model elements defined in MOF and UML. A fully-qualified name then is a concatenation of a model element's name and the name of its owner and so on until a root model element is reached (names are separated by a delimiter). For instance, "Model1::P1::C1" would be the fully-qualified name for class "C1" belonging to package "P1" in a UML model named "Model1."

The following two requirements have to be met by identification mechanisms suitable to enable model comparison:

1. A model element's identifier has to be constant for its whole lifespan.
2. It has to be globally unique such that when model elements are added to one of the to-be-compared models, their identifiers do not collide with elements from this model and from other to-be-compared models.

For identifier-based equivalence, accessing matching model elements can then be realised by a repository (one for each compared model) which provides access to model elements by means of their identifiers.

Regarding UML, there exist model elements which should be identified by their value rather than their identifier (value-based equivalence) in order to yield better merge results. For instance, the lower and upper bounds of instance of UML's *Property* class. Their equality is defined by the *ValueSpecification*'s concrete type's *value* reference property (concrete types are *LiteralInteger* for the lower bound whose value can only be 0 or >1 and *LiteralUnlimitedNatural* for the upper bound whose value can be -1, 0, or >1). Even if these elements have different global identifiers which are be assigned when the bounds were changed, they might still have the same value assigned and can, therefore, be regarded as identical. It follows that the value of the model element defines its "local identity." The identification is local because it is defined from the container element's point of view (for instance, from the UML *Property*'s *lowerValue* and *upperValue* containment reference properties' point of view), not from the viewpoint of the referenced element's type (for instance, *ValueSpecification*).

Nevertheless, UML model elements which, according to their "local identity" value, are equal but have different identifiers are *not* equivalent if other properties are assigned (for instance, the name or stereotype property for *ValueSpecification*). For the merge approach implemented as part of this research, only the lower and upper bound of UML properties are defined to be locally identifiable. For them it is very unlikely that their name or stereotypes will be assigned.

Of course, for each model element only one identifier can be used in the merged model. That is why the identifiers of model elements which have different identifiers, but are found to be locally identical have to be harmonised. The identifier as assigned for the common ancestor is then used for both evolved models, too. This approach guarantees that merge conflicts can be detected (see Chapter 6) because the same identifier is used for both evolved model elements.

D.5.3.2. External model elements

External model elements are not themselves taken into account during comparison. External model elements are those not contained in compared models, but referenced by element from those models. External model elements are thus not contained by any *containment* reference property of any of the model's elements, but only referenced by *reference* properties. They are, therefore, taken into account when the (reference) property values of internal model elements are compared because those elements can reference external ones.

D.5.4. Model element comparison

In terms of comparing model elements, the identification of values of *attribute* properties is based on the actual *values*. The comparison of *reference* properties' values cannot be based on their values because these values represent model elements from different models. These model elements are, of course, defined in terms of properties and, since they come from different models, the values of these properties might differ (because the model was modified). Therefore, the values of reference properties have to be compared based on the identifiers of the referenced model elements.

D.5.4.1. Collecting the to-be-compared model elements into model element repositories

Before the actual model comparison is performed, a so-called *model element repository* is create for each of the two compared models. It provides access to a model's element via their identifiers. Starting with a model's root element and recursively following the containment reference properties, all model elements are added to the model element repository. This way, only those elements which actually belong to the model by means of containment are handled during the model element comparison. External model elements (see Section D.5.3.2) which are not part of the model by means of containment are thus not compared themselves. External elements are thus only relevant when properties referencing them are compared.

When the two to-be-compared models are compared, added and deleted model elements are detected first. In a second step, the property values of equivalent model elements are compared. Property values are compared from the point of view of the model element to which the properties belong. Only these values are compared. In case the values are themselves model elements again (values of *reference* properties), property changes of these model elements are not taken into account for this specific comparison. The property values of these model elements will be or were already compared when the two equivalent model elements of this are/were compared. If the referenced model element was an external one, the values of its own properties won't be compared at all.

All detected changes are added to the *change set* (or *change repository*) which contains all changes detected when two models (i. e., the ancestor and an evolved model) are compared. The two change sets resulting from comparing the common ancestor model with the two to-be-merged evolved models are then used to create the merged model and to detect merge conflicts.

D.5.4.2. A meta-model for model element changes

Based on the property features defined by UML's meta-meta-model, the following types of model element changes can to be defined for and detected when comparing (based on their states which are defined by the values of the properties) two models which adhere to MOF-based meta-model (not only the UML meta-model):

- Model element addition change

  - Reference to the corresponding containment property value added change belonging to the added element's container model element.

- Model element deletion change
  - Reference to the corresponding containment property value removed change belonging to the deleted element's container model element.

- Model element relocation change
  - Reference to the corresponding containment property value removed change belonging to the relocated element's container model element in the common ancestor model.
  - Reference to the corresponding containment property value added change belonging to the relocated element's container model element in the evolved model.
  - Note: A relocation change is defined from the point of view of the contained (i. e., child) model element. From the container model element's point of view, a property value change exists which lists the child as added (for the new container in the evolved model) and removed values (for the old container in the ancestor model). Model element relocation changes are thus created when the values of *containment* reference properties are compared (see Section D.5.4.8).

- Unordered property value change (for single- and multi-valued unordered properties)
  - Fully-qualified name of the property.
  - Added values (collection of values).
  - Removed values (collection of values).
  - Note: For attribute properties, the values are instance of the property's type data type. For reference properties, the values are the identifiers of the referenced model elements. (The author decided against directly referencing the model elements because doing so would bind the changes to the compared models, i. e., in order to deal with the changes the two compared models would have to be available, too, because their model element need to be referenced. So, this indirection by means of referencing

model elements only by their identifier make the changes independent of the compared models). For consistency reasons, a collection of values is used even if the compared property was a single-valued one.

- Ordered property value change (for multi-valued ordered properties)
  - Fully-qualified name of the property.
  - Added values (pairs of added value and index representing the position at which the value was added).
  - Removed values (pairs of removed value and index representing the position from which the value was removed).
  - Reordered values (pairs of reordered value and pairs of indexes; each indexes pair represents a reordered value's old (common ancestor model) and the new (evolved model) index).
  - Note: As for unordered property value changes, the values of attribute properties are instances of the property's type data type. For reference properties, the values are the identifiers of the referenced model elements. The added, removed, or reordered value is part of the change information in order to allow for convenient change interpretation when merge conflicts are detected. It would, however, also be possible to access the changes' values directly in the respective models using the index information provided by the changes.

D.5.4.3. Detecting model element additions and deletions

The detection of added and deleted model elements is based on a collection of model elements belonging to a model. It is not based on the model's containment hierarchy as defined by the containment reference properties because model elements can be relocated in the compared model versions.

Since equivalent model elements are identified by their identifier (see D.5.3), a set of identifiers representing all model elements is created for the ancestor model (ID set A) and the evolved model (ID set E). Then, the differences of both sets are calculated in order to detect added and deleted model elements. For each model element which was detected as added or deleted, a corresponding model element change is created and registered at the change repository:

- Input: Ancestor model's set of model element identifiers (ID set A); evolved model's set of model element identifiers (ID set E).
- Detect deleted elements, i. e., those elements identifiers that remain when ID set E is subtracted from ID set A ($A \setminus E$).
- Detect added elements, i. e., those elements identifiers that remain when ID set A is subtracted from ID set E ($E \setminus A$).

After all added and deleted model elements were detected, the states of the model elements which exist in the ancestor and the evolved model are compared. This is done by comparing the values of their properties as explained next.

D.5.4.4. Detecting properties value changes

In order to detect property value changes and model element relocation changes, the property values of all model elements which exist in the ancestor *and* the evolved model are compared. The respective identifiers (see D.5.4.3) of these model elements are determined by calculating the union of the ancestor model's set of model element identifiers (ID set A) and the evolved model's set of model element identifiers (ID set E): $A \Cap E$. For each model element identifier contained in the resulting ID set, each property's values are compared for the corresponding ancestor and evolved model element.

For single-valued properties, the property's value represents at most one value (or *null* if no value has been assigned). The values of multi-valued properties are contained in either a *collection* or *sequence* (see [OMG06a, pp. 24]). A collection is used for unordered properties, and a sequence is used for ordered ones.

As explained in D.4.1.1, derived properties cannot be changed directly by modellers. That's why they do not need to be taken into account during model comparison.

When comparing attribute properties, the actual values of each property are compared. When reference properties are compared, it has to be taken into account that their values represent model elements. Since the compared property values belong to different models, by definition the model elements are different *instances.* Therefore, in order to determine if a value represents the same two model elements, the identifiers of the actually referenced model elements are used during model comparison.

The following sections explain how property values are compared and changes are detected (and registered at the change repository).

D.5.4.5. Comparing values of single-valued properties

A single-valued property's values can only be modified in one way in the evolved model: the value is changed. However, since MOF-based properties allow to distinguish whether a property's values was set or unset, the latter case is regarded as a special case and distinguished from assigning the property's value.

1. Input: Ancestor model version's property value; property value from evolved model version.
2. Compare values and, if the values are not identical, create and register (1) a *property value removed change* if the property's value has been assigned in the ancestor model version and (2) a *property value added change* if the property's value has been assigned in the evolved model version.

D.5.4.6. Comparing values of unique, unordered, multi-valued properties

1. Input: Collection of the property's values from ancestor model (*collectionA*); collection of the property's values from evolved model (*collectionE*).
2. Detect removed values, i. e., those values which exist in the ancestor model's property values, but not in the evolved model. They are represented by those values which remain when *collectionE* is subtracted from *collectionA* ($collectionA \setminus collectionE$). Create and register a *property value removed change* for each of the removed values.
3. Detect added values, i. e., those values which exist in the evolved model's property values, but not in the ancestor model. They are represented by those values which remain when *collectionA* is subtracted from *collectionE* ($collectionE \setminus collection\mathrm{A}$). Create and register a *property value added change* for each of the added values.

D.5.4.7. Comparing values of unique, ordered, multi-valued properties

Since there cannot be gaps (i. e., *null* values) in values of multi-valued properties (see Section 2.1.2.1), removing a value from its current position will move the values behind it one position to the left (i. e., its new index is its old index minus one), inserting a values at a certain position will likewise move the current value at this position and all values on its right-hand side by one position to the right (i. e., their new indexes are their old indexes plus one). For ordered features, three types of changes can be distinguish: value added, removed, and value reordered. Except for adding a value to or removing it from the end of the sequence of property values, these three types of changes will always cause the indexes of other values to change. Since the merge approach implemented by this research is based on the state of the to-be-merged models and not on the actual operations performed by the modelling tools (used to create the models), element re-orderings detected by the change detection algorithm presented below do not necessarily have to be the ones made by modellers in the modelling tool used to create the to-be-compared models. This is because the current state of the model versions is compared, not the course of editing actions (in the modelling tool) that lead to this state.

Reorder modifications are not necessarily minimal with respect to the distance between the value's index in model version A and its index in model version B. This is because value reorder detection starts from the beginning (index 0) of the sequence. In order to ensure the detection of minimal reorder modifications, the so-called *edit distance* has to be taken into account: "*The term 'edit distance' is sometimes used to refer to the distance in which insertions and deletions have equal cost and replacements have twice the cost of an insertion*" [Jac04, p. 190]. The minimal reorder problem faced here is similar to the *string-to-string correction problem* which refers to the minimum number of edit operations necessary to change one string into another. A single edit operation changes a single character (a property value in the case of model elements) of the string into another by deleting or inserting a character [WF74].

The approach taken here does not strive for the value reordered detection to be minimal because, as discussed above, the real course of user actions (performed by means of a modelling tool) which lead to the value reorder changes is not known.

The first two steps of comparing values of unique, ordered, multi-valued properties are identical to comparing values of unique, unordered, multi-valued properties (see Section D.5.4.6). However, when property value changes are created, information about the index of the added or removed values is added to the respective property value change.

1. Input: Sequence of values from the ancestor model; sequence of values from the evolved model.
2. Detect removed values, i. e., those values which are in the ancestor, but not in the evolved model.

3. Detect added values, i. e., those values which are not in the ancestor, but in the evolved model.
4. Detect reordered values, i. e., those values which exist in the ancestor *and* the evolved model, but whose position (i. e., index) changed. Only explicit reorder changes are taken into account. Reorder changes resulting from adding or removing values are not regarded as reorder changes and are, therefore, ignored.
   a) Create two new sequences: *sequenceA* := ancestor model values not including the values *removed* in the evolved model (as determined above); *sequenceE* := evolved model values not including the values *added* in the evolved model (determined above). As a result, two sequences which contain only those values that exist in both models and, therefore, may have been reordered in the evolved model are taken into account.
   b) Compare the indexes (i. e., positions) of *sequenceA* and *sequenceE*'s values. If a value's ancestor and evolved indexes differ, create a *property value reorder change.* Use the value's indexes from both *original* sequences, not from the sequences without added/removed values (i. e., *sequenceA* and *sequenceE*).

D.5.4.8. Detecting model element relocation changes

Model element relocation changes refer to model elements being moved to a different container model element and/or a difference containment reference property. Therefore, if a model element is relocated, it is removed from one containment reference property and added to another one. Then, the old (ancestor model) container is the property and container element which the model element was removed from, and the new (evolved model) container is the property and container element which the element was added to.

Therefore, when a compared property (no matter if it is a single- or multi-valued one) is a containment reference one, model element relocation changes can be detected (for the *contained* model elements, i. e., to property's values, *not* the *container* model element itself):

- If a value was removed from the property, but the model element it represents was not deleted from the evolved model, the ancestor containment part of the model element relocation change (see Section D.5.4.2) is created and registered.

- If a value was added to the property, but the model element it represents was not added to the evolved model, the evolved containment part of the model element relocation change (see Section D.5.4.2) is created and registered.

Reordering a model element in an *ordered* containment reference property (i. e., a property value reordered change exists), is not regarded as a containment change. It is a reorder change from the view point of the container element's property.

D.5.4.9. Comparing values of non-unique, multi-valued properties

The comparison of *non-unique* properties was not implemented as part of this thesis because it was not relevant for OMOS; neither the part of the UML meta-model relevant for OMOS (see Section D.1.1) nor the diagram models (explained below) has non-unique properties.

D.5.4.10. Type changes

Meta-model-based model (state) comparison approaches can also detect type changes of model elements. For instance, a UML modelling tools might allow for classes to become interfaces and vice versa. This means that the type of a model element can change. In MOF-terms, the model element's instance of its MOF *Class*, therefore, became a different one (in the example above, the type changed from *Class* to *Interface*).

With a comparison approach based on model element identifiers, type changes can of course only be recognised if the respective model elements will still have the same identifiers. So, the possibility of changing a model element's type then means that after the type change, the model element still has to be one single element in order to be recognised during model comparison.

Since the to-be-compared properties depend on the model element's type, the types of equivalent model elements have to be detected before the property values of are compared.

Since type changes are not relevant for OMOS models, they are not taken into account in this thesis.

D.6. Design and implementation of OMOS model merging

Based on the overview provided in Chapter 6, this section discusses the details of the approach to merging OMOS models implemented by this research.

D.6.1. Merge conflicts

This section provides an overview of the merge conflicts which can occur when models are merged.

D.6.1.1. Concurrent property value change conflicts

- The values of a property were changed in different ways in both evolved models.
- Each of the property's value changes is marked with a conflict. By doing so modellers can later decide about and resolve each value's conflict state individually (instead of dealing with the property as one single value).

D.6.1.2. Deletion-modification conflicts

- Conflict between a model element deletion change and ...
  - property value *added* changes (as will be explained below, property value *removed* changes do not conflict with the deletion change, they are considered non-conflicting changes);
  - a model element relocation change; or
  - property value *added* changes for *referencing* model elements (again, property value *removed* changes of referencing model elements do not conflict with the deletion change, they are considered non-conflicting changes).

D.6.1.3. Model element existence conflicts

- Concurrent model element relocation change conflict (i. e., two value added changes at different container model element); or

- Concurrent property change conflict for single-valued *containment* reference property (both contained elements from both evolved models, therefore, cannot be contained in the merged model as well as their direct and indirect contained elements).

D.6.1.4. Indirect model element existence conflicts

Modellers have to know about model elements which are not part of the merged model because some container element had a direct existence conflict. Therefore, model elements which themselves do not have model element existence conflict but belong to a container which has a direct or indirect model element existence conflict, are annotated with indirect model element existence conflicts.

D.6.1.5. Dangling reference conflicts

Each element (i. e., value) of a reference property which has an (indirect) existence conflict is marked with a dangling reference conflict because the referenced element does not exist in the merged model. Modellers can then decide that the conflicts of a sub-set of values are resolved. References to non-existent elements and the non-existence of the element itself are handled separately in order to allow modellers to express that it is fine for the referencing element not to reference elements with dangling reference conflicts, but still leave open whether the non-existence of the element is conflicting. Accepting the non-existence will also accept the inability to reference the element.

Dangling reference conflicts are derived from existence conflicts, they are *not* created when the initial merged model is created. When the modeller manually define that it is fine to not reference the dangling element, this information is explicitly created and persisted.

Conflicts have to be calculated for all model elements, not just for those which are part of the merged model. This is done to ensure that when model elements with existence conflicts become part of the merged model again their dangling reference conflicts are known (i. e., references to other model elements which might have been part of the merged model, but were manually removed during the merge conflict resolution phase).

D.6.2. Accepting and rejecting model changes and detecting merge conflicts

Each model change (model element added, deleted, or relocated change and each value property added, removed, or reordered change) have an *acceptance state.* The change is, therefore, either accepted or rejected. The details of accepting and rejecting changes will be discussed next.

The model merge approach developed as part of this research focuses on merging models adhering to MOF-based meta-models. It is based on the rules defined by MOF:

- Every model element must have a container model element (except for the model's root elements).
- Single-valued properties can have at most one value.
- Multi-valued properties with a limited upper bound must not have more values than defined by the upper bound (see Section D.6.4.6).
- Multi-valued *reference* properties cannot have "holes", i. e., every value of such a property must be a model element: *"Null is not a valid value within the list"* [OMG06a, p. 36]. Multi-valued *attribute* properties may contain *null* values *"to indicate the absence of a value"* [OMG06a, p. 11].

The values (state) of properties can of course only be assigned according to the rules defined by the meta-model of the (merged) model. Since this meta-model is based on MOF, the rules defined by MOF have to be adhered to by the (merged) model. Therefore, the process of deciding whether model element changes are conflicting is primarily based on the rules defined by MOF. Additional conflict types can be defined as will be demonstrated for merging OMOS diagrams in Section D.7.1.3. The merge process defined by this research can thus be used to merge models adhering to any MOF-based meta-model, not only UML.

Validating the correctness of the merged model's syntax in terms of additional rules specifically defined by the meta-model has to be validated by applying model consistency checking tools specific to a model's meta-model. The merge approach applied here guarantees that the fundamental syntax rules as defined by MOF are met. However, more complex syntax rules defined by the meta-model in addition to the MOF rules have to be validated by additional tooling. Such tools are usually part of the modelling tools used to create the models. For instance, UML modelling tools usually provide the possibility to check a UML model against the syntax rules defined by the UML standard Section 3.2.2.

The first step for creating the merged model is to determine which model element changes can be accepted in the merged model and which are conflicting with changes from the other evolved model. These merge conflicts are detected based on the two sets of changes resulting from comparing the common ancestor model with each evolved model.

A model element is only omitted from the merge model if it cannot become part of the merge model because it cannot be contained by its container. This is the case if:

- the model element was relocated to different containers in both evolved model versions;
- the capacity of the container element's containment property value is exceeded (this happens if the containment property is a single-valued one and two different model elements are "competing" to become the value of this property, or if the property is a multi-valued one with a limited upper bound, i. e., cannot contain an unlimited number of values);
- the model element's container element is not part of the merge model because of one of the reasons mentioned above (i. e., the container element has an existence conflict).

D.6.2.1. Concurrent modification conflicts

- Concurrent property value change conflict
  - The value of a property of a model element was changed in both evolved models in different ways, i. e., there exist different property value changes for this property.
  - Each property value gets marked with a conflict.
- Concurrent model element containment change conflicts with different new container elements.

D.6.2.2. Deletion-modification conflicts

In addition to the two conflict types above, deletion changes are also conflicting if the respective model element has property value changes and/or a container change, i. e., it was deleted in one evolved model and modified and/or relocated in the other evolved model.

However, a deletion change is not conflicting if the property changes represent only value *removed* changes. This approach guarantees that removing values from a model element which was deleted in the other evolved model does not cause conflicting deletions. However, depending on the actual changes, a *semantic* conflict might exist. Therefore, the changes have to be validated by modeller in during the second phase of the merge process when merge conflicts are solved (see Section 6.5.5).

As explained in Section D.6.3, model elements with existence conflicts are *not* part of the merge model. However the property value changes and/or container change made to the respective model element in the other model version are nevertheless accepted — since the model element is not part of the initial merged model, no property values are actually changed. This approach guarantees that changes made in the evolved model in which the model element was not deleted can actually be applied if this element becomes part of the merged model. For example, if the model element was relocated to a different container in this evolved model, the respective relocation change is accepted and this model element is no longer part of the old container element in the merged model (as it would if the container change was not accepted).

D.6.2.3. Accepting changes

All non-conflicting changes are *accepted*, i. e., the changes can be applied to the respective model element in the merged model. However, non-conflicting (i. e., accepted) changes can only by applied in the merged model if the respective model element exists in the merged model. The reason why model elements may not exist in the merged model is explained next.

D.6.2.4. Merge conflict repository

If conflicting changes exists for a model element, a merge conflict is created and registered for the respective model element in the *merge conflict repository.* Each conflict has a *conflict resolution state*. Until the modellers manually resolve the merge conflicts, each conflict's resolution state is *unresolved*.

D.6.3. Detecting merge conflicts

The objective of the conflict detection process is to detect changes (made in parallel in both evolved models) which contradict each other and thus cannot be applied to the merged model without further considerations by modellers. If changes are found to be non-conflicting, they are *accepted* and the respective model element will be changed accordingly in the merged model (if it is part of it). Depending on the type of conflict, conflicting changes are rejected or accepted as will be discussed below.

First, model element deletion changes are handled in order to decided whether the deletion can be accepted. Then, model element relocation changes are handled, and after that, property value changes are checked for conflicts.

Based on the change types defined in Section D.5.4.2, the following types of conflicts exists:

- *Concurrent property value change conflicts* indicate contradicting changes (made in both evolved models) of a property's value(s) of a model element.
- *Concurrent model element relocation change conflicts* refer to the fact that a model element was relocated in conflicting ways in both evolved models.
- *Deletion-modification conflicts* occur when a certain model element was deleted in one evolved model, but modified in the other one.

D.6.3.1. Analysing model element deletion changes to detect deletion-modification conflicts

The following approach is applied for analysing deletion changes to verify whether these changes can be accepted:

1. If the model element was only deleted in one evolved model, it is checked whether it was modified in the other evolved model version. A model element deletion change for model element e in evolved model eA is in conflict with the following types of changes made in the other evolved model eB:
    - All property value added, removed, or reordered changes of e in eB.

    - *e*'s model element relocation change.
    - All property value added, remove, or reordered changes of properties of other model elements referencing *e*.

2. If any of those changes exist, a deletion-modification conflict is created and registered at the conflict repository. The model element deletion change and each of the conflicting changes above is registered at the deletion-modification conflict.

In case a deletion-modification conflict exists, it has to be decided which changes are accepted and which are rejected. For the merge approach implemented at part of this thesis, it was decided that property value *removed* changes of the model element's properties itself and of any referencing model elements' properties are treated as conflicting with the model element deletion change, but do *not* prevent the model element from being deleted. The rationale behind this decision is that removing property values should not prevent the model element from being deleted because (1) property values were only removed from the to-be-deleted model element (in evolved model *eB*) and (2) elements which referenced it are no longer referencing it (in evolved model *eB*). Since the model element deletion change and its conflicting changes were registered as conflicting, the modeller has to verify whether the deletion can indeed be accepted or not (during the conflict resolution phase of the merge process, see Section 6.5.5).

The following rules for accepting and rejecting a model element deletion change and its conflicting changes are defined:

1. If the model element was deleted in both evolved models, i. e., each change set has a model element deletion change for this model element, accept both deletion changes.
2. If *e* has (1) property value added and/or reordered changes, (2) a model element relocation change, and/or (3) property value added and/or reordered changes referencing *e* of properties of other model elements in *eB*, then:
    - Reject the deletion change; and
    - Accept *all* property value (added, reordered, removed) changes of the *e* itself.
    - Note: Changes of *referencing elements* are handled when these elements are handled (see below).

3. Else, accept the deletion change and *all* property value removed changes of e itself.

The analysis of changes presented in the following sections is performed for model elements which were not deleted in any of the evolved models, i. e., model elements which either exist in the ancestor model element and both evolved versions, or were added to an evolved model version.

D.6.3.2. Analysing model element addition changes to detect addition conflicts

Whether a model element can be added to an evolved mode depends on the to-be-added model element's containment. If the container model element does not exist in the merged model, the added model element cannot be part of the merged model. For the merge approach implemented at part of this research, it was decided that model element addition changes are always accepted, even if the model element itself cannot be part of the merged model. In such cases, existence conflicts exist for the respective model elements and the model element addition changes are registered as conflicting. The reason for accepting the addition change is that when the existence change is resolved, the model element can then become part of the merged model.

D.6.3.3. Analysing model element relocation changes to detect concurrent model element relocation change conflicts

The following approach is applied for analysing model element relocation changes to verify whether these changes can be accepted:

1. If the model element has a relocation change in only one evolved model, it is accepted.
2. If the model element has relocation changes in both evolved models, accept both changes if the new containment references are identical (i. e., the model element was relocated to the same container element and the same containment reference property), else, both changes are rejected. In the latter case, a concurrent model element relocation change conflict is created and both changes are registered for it.

For the model merge approach implemented as part of this research, it was decided that model elements with rejected concurrent model element relocation changes should not be part of the merged model. This means that even if the model element relocation changes were rejected, the relocated model element is not contained by its ancestor model version's container element the in the merged model. This decision was taken because the container element as it was in the ancestor model version might have been non-conflictingly deleted in both evolved model elements and this won't be part of the merged model. Therefore, since a model element with a concurrent container change conflict cannot be part of the merged model, an existence conflict exists for the conflicting relocation changes.

D.6.3.4. Analysing property value changes to detect concurrent property value change conflicts

After deletion-modification conflicts and concurrent relocation conflicts were detected, property value changes are analysed to detect conflicting changes. The analysis is based on the different property types: For single- and multi-values properties, a concurrent property change conflict is created if the property values of both evolved models are not identical. The decision as of when value property changes are regarded as conflicting depends on the properties type. A definition of conflicting changes for the three different property types relevant for merging OMOS models is presented next.

D.6.3.5. Analysing property value changes of single-valued properties

For a single-valued property, a concurrent property value change conflicts exists if the property's value was changed in contradicting ways in both evolved models. The following approach is applied for accepting or rejecting property value changes:

1. Accept *all* property value removed changes, no matter if they were done for one evolved model only or for both (for a single-valued property, a value removed change means that the property's value was unset).
2. If there exist a property value added change for *only one* evolved model, accept it and create concurrent property value change conflict for the value property added and the respective value property removed change from the other evolved model (if it exists).
3. Else, if there exist a property value added change for *both* evolved models,
    a) Accept both changes if the added value is the same for both.

b) If not, reject both changes and create a concurrent property value change conflict for both value property added changes. If the property is a *containment* reference property, create an existence conflict for both model elements referenced by both value property added changes to indicate that these children elements cannot be part of the merged model because both changes were rejected.

D.6.3.6. Analysing property value changes of multi-valued, unique, unordered properties

For a multi-valued, unique, unordered property, a concurrent property value change conflicts exists if the property's value was changed in contradicting ways in both evolved models. The following approach is applied for accepting or rejecting property value changes:

1. Input: Two sets of property value added and removed changes.
2. Accept *all* property value removed changes (independently in which evolved model version they exist).
3. Accept *all* property value added changes (independently in which evolved model version they exist).
4. Create a concurrent property value change conflict for property value added/removed change only made in *only one* evolved model.

By applying this approach, all changes are accepted in the merged model, but the property gets a concurrent property value change conflict if different changes were applied to it in both evolved models.

D.6.3.7. Analysing property value changes of unique, ordered, multi-valued properties

The following conflicting changes per value and approach to accepting/rejecting changes can occur for unique, ordered, multi-valued properties:

- If value was only added in one evolved model, then accept this change and add it to the property's concurrent property value change conflict.
- If value was only removed in one evolved model, then accept this change and add it to the property's concurrent property value change conflict.

- If value was added (in both evolved models, but) at different positions or reordered (in both evolved models, but) to different positions, then it cannot be guaranteed that at least one new position (as defined in one of the evolved models) can be kept in merged model (because of other value reorderings, additions, removals; also, there cannot be holes in sequences). Therefore, accept both changes and add it to the property's concurrent property value change conflicts.
- If value was removed in one evolved model, but reordered in the other evolved model, then accept reorder change and reject value removed change and add both to the property's concurrent property value change conflicts. Again, it cannot be guaranteed, that the new value position (as defined in the evolved model) can be kept in the merged model (because of other value reorderings, additions, and removals, and because there cannot be holes/null values).

The following algorithm is applied for accepting or rejecting value property changes of unique, ordered, multi-valued properties:

- Input: Two sets of property value added, removed, and reordered changes.
- If not exactly the same addition *and* removal *and* reorder changes were made to the property in both evolved model's (i. e., if not all property values of both evolved model elements are identical), then create a (unresolved) concurrent property value change conflict for the property and assign the following property value changes to it:
  - Assign *all* addition and reorder changes to the conflict (reason for the conflict: because of other reorderings, additions, removals and because there cannot be holes, it cannot be guaranteed, that the new position can be kept in merged model).
  - Assign removal changes only made in one evolved model to the conflict.
- Accept *all* property value added changes (even if their positions differ, they are is marked with conflicts (see above) and, therefore, have to be taken into account during the conflict resolution phase).
- Accept *all* value reordered changes (even if their positions differ or if the value was removed, they are marked with conflicts (see above) and, therefore, have be taken into account during the conflict resolution phase).

- Accept property value removed changes which do *not* have corresponding conflicting reordering changes in the other evolved model (i. e., the value was removed in both evolved models or was not modified in the other evolved model).

D.6.4. Creating the (model elements of the) initial merged model

After the merge conflicts were detected, all non-conflicting changes were accepted, and all conflicting changes were rejected, the process of creating the initial merged model starts with the root model elements. First, the values of the root element's containment reference properties are merged. This process is recursive, i. e., the values of all containment reference properties of all the model elements handled for the root element are merged and so on. The approach for merging the property values is explained in the next sections.

As explained in Section 6.2.1, a fundamental rule inherent to all MOF-based metamodels is that each model element (except for the model's root element) has to belong to a container element. This element contains children elements by means of its containment references. Therefore, in order to create the merged model, the values of all containment properties (i. e., containment hierarchy) are merged first. This guarantees that all model elements can be referenced in the second merge step when the values of non-containment reference (and attribute) properties are merged.

In the first step, an empty copy of each model element which does not have an existence conflict is created, its identifier is set, and it is added to its container's respective containment property's values. The merged model elements are created starting at the root model element and following the *containment* reference properties.

In the second model merge step, the values of attribute and *non-containment* reference properties are merged. In case *non-containment* reference property reference a model element which has an existence conflicts it is, therefore, not assigned to the property's values in the merged model. The details of merging property values are explained in the next sections. As with comparing property values, the values of attribute properties represent attribute values while the values of reference property represent actual model elements. When dealing with model elements as property values, the identifiers of the model elements are used to access equivalent model elements from different models (i. e., common ancestor, evolved, and merged models). However, the actual model elements (not just the identifiers) are, of course, used as the values of these reference properties.

D.6.4.1. Merging single-valued properties

1. If the property is an attribute property and if concurrently property value change conflict exists, then *value* := property's value from ancestor model.
2. Else
   a) *value* := property's value from ancestor's model.
   b) Apply *accepted* property value removed change (if any exists) to *value*.
   c) Apply *accepted* property value added change (if anyexists) to *value*.
3. Assign *value* to merge model element's property.

If a concurrent modification conflict exists for an *attribute* property, the ancestor value is used in the merged model to ease manual conflict resolution for modellers. For instance, if the attribute property was the model element's name, it is helpful if at least the model element's name from the common ancestor is presented in the merged model during the conflict resolution phase. This benefit does not exist for *reference* properties because the referenced model element from the common ancestor might not exist in the merged model (for instance, it may have been deleted from both evolved models).

D.6.4.2. Merging unique, unordered, multi-valued properties

Merging the values of a multi-valued property whose values are unique and unordered means to assign the values (if any exist). The following possible changes and conflicts have to be taken into account:

- Possible changes (independently for both evolved models): Value added and removed changes.
- Possible conflicts: Deletion-modification conflict; in this case, the model element is not part of the merged model and thus the property value is not assigned. Concurrent modification conflicts (i. e., value changed in contradicting ways in both evolved models) cannot exist because removing and/or adding same *and/or* different values in both evolved models is not regarded as conflicting for unique, unordered, multi-valued properties. It is possible that there

exist semantic conflicts for the property's merged values, but this has to be decided by the modellers during the conflict resolution phase (see Section 6.5.3). *All* value added and value removed changes are, therefore, *non-conflicting*.

1. *collection* := property's values from *ancestor* model.
2. Remove all values from *collection* which have *accepted* property value removed changes by applying the property value removed changes.
3. Add all values to *collection* which have *accepted* property value added changes by applying changes by applying the property value added changes.
4. If the property is a reference one, remove all those values form *collection* which represent model elements with existence conflicts. This has to be done to ensure that only model element are referenced which actually exist in the merged model.
5. Assign the values for merge model element's property by copying from the values in *collection*.

As will be discussed in Section D.6.4.6, for properties with a limited upper bound, only the amount of values which fit into the merged model's collection can actually be assigned. The limit might have been exceeded because the values from the two evolved models were used. However, upper bound limits are not defined by the UML meta-model for any of the UML elements relevant for OMOS.

D.6.4.3. Merging unique, ordered, multi-valued properties

Merging the values of a multi-valued property whose values are unique and ordered means to assign the values from the evolved models (if they exist). The following possible changes and conflicts have to be taken into account:

- Possible changes (independently for both evolved models): Value added, removed, and reordered changes.
- Possible conflicts: Deletion-modification conflict; in this case, the model element is not part of the merged model and thus the property value is not assigned. As for unordered properties, concurrent modification conflicts cannot exist because removing, adding, and/or reordering the same *and/or* different values in both evolved models is not regarded as conflicting. It is possible that there exist semantic conflicts for the property's merged values, but this

has to be decided by the modellers during the conflict resolution phase (see). *All* value added, value removed, and value reordered changes are, therefore, *non-conflicting*.

Values of multi-valued, unique, ordered properties can be modify in the evolved models in ways which can be regarded as contradicting. This is because the *order* of the property's values is relevant. As explained in Section D.5.4.7, adding, removing, and reordering values in an evolved model will change the position of other values too. Therefore, the position of identical values in both evolved model's sequences can differ. Since the order of values is relevant, differing positions of identical values *could* be regarded as merge conflicts. If different values were added, deleted, and/or reordered in *both* evolved models, it is impossible to automatically merge the values in a manner that preserves *both* sequences' value order. Therefore, the merge approach presented here automatically merges the property's values from both evolved models by iterating over the sequences of both evolved values starting from their beginning (leftmost value) and using only the first occurrence of a value in the merged model. This approach may potentially not take into account the value's position in the other evolved model's sequence. All removed values are *not* part of the merged model. This means that values which have value removed changes either in one of the evolved models or in both models are excluded from the property's value in the merged model. This holds even if removed values have value reordered changes in the other evolved model.

The algorithm for merging the values of unique, ordered, multi-valued properties presented below is based on the *state* of of the two to-be-merged value sequences (taken from the evolved versions of to-be-merged model elements). Utilising the actual value property changes to merge the order property's values did not appear to be feasible because the actual positions at which values are inserted may be different than the position given by the property changes because of value added, removed, and conflicting reordered changes. However, care was taken for the merge algorithm to be in sync with the algorithm for accepting and rejecting value property changes and for detecting conflicts.

D.6.4.3.1. Merge algorithm: The merge algorithm uses *pivot values* for determining the position at which added values are inserted into the merged property's sequence. The pivot value of an added value is the first value left of the added value in this value's evolved property values which exists in the merged model's values. If the value was added at the beginning, before any non-added value, the beginning of the list represents the "pivot value."

1. Preparation
   a) Create a copy sequence (*sequence1*) of the property's values from evolved model 1. Remove all values with *accepted* value removed changes in evolved model 2.
   b) Create a copy sequence (*sequence2*) of the property's values from evolved model 2. Remove all values with *accepted* value removed changes in evolved model 1.
   c) If the property is a reference property, remove all values which represent model elements with existence conflicts from *sequence1* and *sequence2*. This has to be done to ensure that only model elements are referenced which actually exist in the merged model.
   d) Create an empty sequence (*sequenceM*) which represents the merged values.
2. Calculate the *sequence* of values to be used in merged model
   a) Merge values (from both evolved models) which were not added (, but were potentially reordered). To do so, copies of *sequence1* and *sequence2* are created and all values with addition changes are removed. The copies now only contain elements which already existed in the common ancestor model and were not removed in the respective evolved model.
      i. *value1* := value from beginning of *sequence1.*
      ii. Append *value1* to *sequenceM.*
      iii. *value2* := value from beginning of *sequence2.*
      iv. If *value1* equals *value2*, do not add it to *sequenceM* because this was already done above.
      v. Else, append *value2* to *sequenceM.*
      vi. Remove *value1* and *value2* from *sequence1.* The latter removal ensures that *value2* is only added once to the merged model in case *sequence1* contains this value too. If *value1* and *value2* are identical, only one removal is required.

        vii. Remove *value1* and *value2* from *sequence2*. The former removal ensures that *value1* is only added once to the merged model in case *sequence2* contains this value too. If *value1* and *value2* are identical, only one removal is required.

        viii. Repeat from the beginning until *sequence1* and *sequence2* are empty or the property's upper bound limit was reached (see Section D.6.4.6).

    b) Merge values which were added either in one of the evolved models or both (potentially at different positions). To do so, copies of *sequence1* and *sequence2* are created and all values without value added changes are removed. The copies now only contain element which were added in one or both evolved models.

        i. For *sequence1*...

            A. *value* := value from beginning of *sequence1*.

            B. Append *value* to *sequenceM* after the last value with a value addition change after (i. e., right of) *value's* pivot value (see above).

            C. Remove *value* from *sequence1* and *sequence2*. The latter removal ensures that *value* is only added once to the merged model in case *sequence2* contains this value too. This means that if *value* was added to the property's values in both evolved models (potentially at different positions), only its first occurrence is used.

        ii. Repeat the steps above for all remaining values of the *sequence2*. (These values were only added to the property's values in evolved model 2, but not in evolved model 1.)

            A. *value* := value from beginning of *sequence2*.

            B. Append *value* to *sequenceM* after the last value with a value addition change after (i. e., right of) *value's* pivot value (see above).

            C. Remove *value* from *sequence2*.

3. Assign the values of the merged model element's property by copying them from the values of *sequenceM* in their exact same order.

The approach for merging the values of unique, ordered, multi-valued properties has to following consequences:

- It is a asynchronous merge approach [Wes91] because evolved model 1's values are handled before evolved model 2's, and, if a value occurs in the property's values in both evolved model's, it is added to the merged model's property values at the first occurrence.
- Values with addition changes are grouped with respect to the evolved model they were added: By merging added values in the way described above (i. e., evolved model 1's values are merged before evolved model 2's values), the order of added values is (mostly) preserved in the way the values were added in the evolved models. However, for values which were added in both evolved models after different pivot values, the order from evolved model 1 is preserved in the merged model. Furthermore, if different values were added after the same pivot value (or at the beginning of the property's values), the added values from evolved model 1 are added to the right of their pivot value, while the added values from evolved model 2 are added after the added values from evolved model 1 belonging to the same pivot value.

Discussion of an alternative approach: Another possible approach (which was not implemented as part of this research) is to calculate the the *longest common sub-sequence* [Mai78] for the property's values from both evolved models. However, by doing so, only the values which occur in a property's values in both evolved models in the *same order* would be used in the merge model. (Added values could be merged by applying the pivot value concept used for the implemented approach above.) But because other existing values (i. e., values without addition changes) would not become part of the property's values in the merged model, this approach may potentially exclude values from the merged model which *are* included by the implemented approach.

D.6.4.4. Merging non-unique multi-valued properties

Non-unique multi-valued properties were not relevant for the merge approach implemented as part of this research since all multi-valued properties relevant for were defined as unique ones (see Section D.1).

D.6.4.5. Bidirectional properties

The bidirectionality of (reference) properties is not taken into account when their values are merged. Setting the values of a property is therefore handled independently for the opposite property. Thus, values of properties belonging to a bidirectional reference will be merged by applying the merge approaches presented above. For ordered reference properties, this will ensure that the values of both properties are in the ordered determined during the merge process.

D.6.4.6. Taking the upper bound of multi-valued properties into account during merging

For multi-valued properties with a limited upper bound, only a certain (limited) number of values can be included in the merged model's property values in order not to exceed this limit. This limit may be exceeded during the merge process because the values from the *two* evolved models were used.

If such a property is a containment reference property, existence conflicts have to be created for the model elements which "do not fit into the property's values." These model elements cannot be part of the merged model.

However, the UML meta-model does not define upper bound limits for any of the UML elements relevant for OMOS. The only multi-valued property with limited upper bound defined by the UML meta-model is reference property *DurationObservation::event* with an upper bound (limit) of two.

D.7. Design and implementation of differencing and merging of OMOS diagrams

Similar to the approach used for merging evolved versions of a UML model, a three-way merge approach is used for merging two evolved versions of a OMOS diagram. Like UML model elements, OMOS diagrams are compared and merged based on their states and state changes. Based on the possible state changes modellers can cause in OMOS diagrams by modifying them, an approach of differencing and merging OMOS diagrams was implemented as part of this research. It will be discussed in this section.

Each OMOS model has a collection of OMOS diagrams belonging to this model. This collection is not part of the UML model underlying the OMOS diagrams, but is maintained independently from the UML model. For the prototype modelling tool implemented as part of this research, the OMOS diagrams constituting a UML model are maintained in addition to the model itself in a *diagram collection* which is a multi-valued, unordered reference property containing diagrams.

D.7.1. OMOS diagrams changes and conflicts

Before the approach to detecting diagram changes and conflicts is discussed below, this section discusses diagram changes and conflicts. In the context of this research and for the prototype merge tool developed as part of it, diagrams are models. They are hence defined by means of a meta-model. Like the UML meta-model, this meta-model is based on the meta-object facility (MOF, see Section 2.1.2.1). That is why the approaches to differencing and merging UML models discussed previously in this chapter apply for diagrams as well. The meta-model for OMOS diagrams used in the context of this thesis is presented next.

D.7.1.1. A meta-model for saving, comparing, and merging OMOS diagrams

In this section, the meta-model used to persist, compare, and merge OMOS diagrams is presented. It is presented in a textual form called EMFText (which is explained in Section D.4.3). EMFText is a concrete syntax of Ecore which (as explained in Section D.4) is an implementation of MOF. The elements defined by the meta-model are thus fully compatible with the MOF-based approach for model comparison and merging discussed in this chapter.

No textual labels (for class names, attributes, association end names and multiplicities, etc.) are part of OMOS diagram meta-model above. The information are derived from the respective UML model element when the diagram is actually displayed (see also Section D.3). The meta-model for OMOS diagrams focuses on the information required to create/define OMOS diagram, use them as input for the actual diagram layout algorithm, and to store them. As will be discussed in the next section, the diagram meta-model also enables detecting the possible changes which can be made in a diagram when modellers modify it manually. Therefore this diagram meta-model represents the diagram's graph. As all meta-models presented in this thesis, it is based on MOF. The state of diagram models is, therefore, defined

```
@"import"("UML")

class Diagram
{
  attribute String GUID (1..1);
  attribute String name (0..1);
  containment reference ClassSymbol classSymbols (0..-1);
  containment reference ConnectionSymbol connectionSymbols (0..-1);
  ordered reference ClassSymbol rootClassSymbols (0..-1);
}

abstract class Symbol
{
  reference uml::Element umlElement (1..1);
}

class ClassSymbol extends Symbol
{
  reference ClassSymbol parentClassSymbol (0..1);
  ordered reference ClassSymbol childClassSymbols (0..-1);
  ordered reference ConnectionSymbol connectionSymbols (0..-1);
  attribute ClassSDetailsVisibility detailsVisibility (1..1);
}

class ConnectionSymbol extends Symbol
{
}

enum ClassDetailsVisibility
{
  0 : public = "public";
  1 : all = "all";
  2 : none = "none";
}
```

Figure D.6.: A meta-model for saving, comparing, and merging OMOS diagrams.

by the (attribute and reference) properties defined for the meta-model's classes. As will be explained below, the state (i. e., the values) of these properties is used to compare diagrams in the same way UML models are compared.

Class *Diagram* represents the root element of a OMOS diagram (model). A diagram has a name, a globally unique identifier (the diagram's name cannot be used as diagram's globally unique identifier because it may change), contains *all* class symbols (via containment reference property *classSymbols*) and connection symbols (via containment reference property *connectionSymbols*), and references its root layer class symbols (via the ordered reference property *rootClassSymbols*). Like all multi-valued properties in this diagram meta-model, they are *unique* properties because a certain diagram symbol can only appear once in a diagram.

By means of the *umlElement* reference, each *ClassSymbol* and *ConnectionSymbol* references the UML model element (i. e., a class or the inheritance or association relationship) it depicts. By means of this reference, all information from the UML model element(s) required to visualise the symbol are accessed. Since a certain diagram can only depict a certain UML model element at most ones, the *umlElement* also serves for identifying diagram symbols. Because each UML model element has a globally unique identifier (see Section D.5.3), the identifier of the UML element depicted by a diagram symbol serves as this symbol's identifier *within its diagram*. As a diagram's symbols only need to be identifiable within this diagram, using the UML model element's identifier will thus not compromise the (global) identifiability of the UML model element itself or other diagram symbols depicting this UML element in other diagrams.

It is not useful the assign *dedicated* globally unique identifiers to diagram symbols because the symbols can be deleted and recreated later on in the same diagram for the same evolved model version, i. e., in the same modelling cycle between two merge processes. It then has to be guaranteed that a diagram symbol is still identified as the same one because it still depicts the same UML model element.

In case it is not a root level one, a class symbol's *parentClassSymbol* reference property references the symbol's parent. Its *childClassSymbols* reference property references the symbol's children class symbols in an ordered manner. The reason why all *ClassSymbols* are contained by the *Diagram* root element and not by their respective parent *ClassSymbol* (i. e., the reason why *ClassSymbol::childClassSymbols* is not a *containment* reference property) is that it must be possible to delete a parent class symbol in a diagram without removing all its children class symbols from the diagram. As explained in Section 5.2 and shown in Fig. D.2, the deleted class symbol is then replaced with a place-holder symbol to allow the modellers to relocate "dangling" children class symbols to other class symbols or delete them as well.

The ordered reference properties *Diagram::rootClassSymbols* and *ClassSymbol::childClassSymbols* define a diagram's (horizontal and vertical) class symbol hierarchy.

A class symbol's *connectionSymbols* reference property references all the class symbol's incoming and outgoing connection symbols, i. e., inter- and intra-layer ones. However, it does not contain the class symbol's passing-by connections because (as explained in D.3) they are automatically determined because passed-by class symbols are not end points (source or target) of connection symbols and cannot be manually modified by modellers.

The *connectionSymbols* reference is an ordered one because the order is relevant for laying out connections and for merging them (as explained in Section D.3). Because class symbols can be moved and connections may thus change from one connection type to another — with respect to the connection types distinguished during automatic diagram layout, for instance, inter- and intra layer connections, not with respect to the actual UML relationship depicted by the connection, for instance, association or inheritance, this type does, of course, not change — it is not useful to separate connections by their type as done when the diagram is actually laid out (see meta-model for OMOS diagram editors in Section D.3).

The *detailsVisibility* attribute is used to define which attributes and operations of a *ClassSymbol*'s underlying class are depicted for this symbol.

Class *ConnectionSymbol* does not define references to the class symbols it connects to because this information can be derived from the connection symbol's UML model elements. Storing this information for *ConnectionSymbol*s would be redundant and is, therefore, not explicitly defined by the meta-model.

D.7.1.2. Diagram changes

As discussed in Section 5.2, modellers can modify OMOS diagrams by means of the following diagram editor actions:

- Add to and delete diagrams from a OMOS model.
- Change a diagram's name.
- Add and delete class and connection symbols to/from diagrams.
- Relocate a class symbol to a different parent class symbol.
- Reorder a class symbol within its parent's children class symbols.
- Change a class symbol's details visibility.
- Relocate a connection symbol's end to a different class symbol. This editor action will also modify the UML model itself because the association or inheritance relationship depicted by the connection symbol is modified itself since

a OMOS diagram can only contain one symbol of a certain class, so relocating a connection's end will connect it to a different class symbol and thus the model element underlying the connection will be added to a different class. As discussed in Section 5.2, this editor action will remove all other symbols of the connection from all other diagrams which depict it because the underlying UML model is updated.

Based on the diagram modification types presented above, the following types of diagram changes have to be detected when an evolved version of a diagram is compared to the common ancestor diagram version; the names of properties from the OMOS diagram meta-model (see Section D.7.1.1) which are affected by these changes are presented as well:

- Diagram added and deleted change (*diagram collection*).
- Diagram name change (*Diagram::name).*
- Class and connection symbol added and deleted change (*Diagram::classSymbols, Diagram::rootClassSymbols* or *ClassSymbol::childClassSymbols*, and *ClassSymbol-::parentClassSymbol* for class symbols, and *Diagram::connectionSymbols* and *ClassSymbol::connectionSymbols* for connection symbols).
- Class symbol relocation change and class symbol reordered change (*Diagram::-rootClassSymbols* or *ClassSymbol::childClassSymbols* and *ClassSymbol::parent-ClassSymbol* in case of a relocation change).
- Class symbol details visibility change (*ClassSymbol::detailsVisibility*).

The relocation of a connection symbol's end to a different class symbol (see the diagram editor actions described above) is not regarded as a diagram change since modifying connection symbols will always affect the underlying UML model. Therefore, whether an connection symbol was relocated can be detected by analysing the UML model changes of the connection's UML element. Details of the approach of comparing diagrams as implemented by this thesis are explained in Section 6.3.

D.7.1.3. Diagram merge conflicts

Similar to conflicting concurrent modifications of UML model elements, diagrams may be modified in both evolved models in conflicting ways. The following concurrent diagram modifications can occur:

D.7.1.3.1. Concurrent diagram modification conflicts:

- A diagram's name was changed in different ways for both evolved diagrams.

D.7.1.3.2. Concurrent diagram symbol modification conflicts:

- Concurrent class symbol relocation change conflict: A class symbol was relocated to different parent class symbols in both evolved diagrams.
- Concurrent class symbol reorder change conflict: A class symbol was reordered in different ways within its parent's children class symbols in both evolved diagrams.
- Concurrent class symbol details visibility change conflict: A class symbol's details visibility was changed in conflicting ways in both evolved diagrams.

Diagram conflicts are only detected for class symbols, not for connection symbols. Since the modification of connection symbols will *always* affect the underlying UML model, conflicting connection symbol modifications are detected based on UML model element conflicts. The diagram changes of connection symbols will then be annotated with conflicts because their underlying UML element have conflicts (see Section 6.5.3).

Concurrent diagram modification conflicts are detected by applying the same approach used for detecting conflicting changes for UML models discussed previously in this chapter.

D.7.1.3.3. Diagram symbol deletion-modification conflicts: A diagram symbols could be modified in one evolved diagram, but deleted in the other evolved one. The deletion change then conflicts with the following diagram modifications:

- Deleting a diagram conflicts with changing the diagram's name.
- Deleting a class symbol conflicts with:
  - relocating the class symbol to a different parent class symbol;
  - re-ordering the class symbol within its parent's children class symbols;
  - changing the class symbol's details visibility level; and
  - adding connection symbols to the class symbol, either by relocating an existing connection symbols end or by adding a new connection symbol to

the diagram. (Since property value *removed* changes are not conflicting with deleting the respective element, removing connection symbols from the class symbol is not regarded as conflicting with the class symbol's deletion.)

- Deleting a connection symbol conflicts with modifying the connection symbol. Since modifying connection symbols will always affect the underlying UML model, connection symbol modifications are detected though the connection symbol's UML model element's changes.

D.7.1.3.4. Conflicting diagram symbol deletion-addition changes: The detection of conflicting diagram changes also has to take into account deletion changes made to the underlying UML model (which was potentially modified by modifying diagram symbols). For class or connection symbols, the following type of conflict has to be detected: A class or connection symbol S is *added* to diagram D in an evolved model. In the other evolved model, S's respective model element(s) and all its diagram symbols (which were already part these diagrams in ancestor model) are deleted. Both evolved models and diagrams were changed in *seemingly* non-conflicting ways. However, the class or connection symbol added to diagram D has a conflict because its underlying UML model element does not exist in the merged model because it was non-conflicting deleted from the model. Furthermore, its diagram symbols were *seemingly* non-conflictingly removed.

Diagram symbol S added to diagram D can therefore not exist in the merged diagram because its underlying UML model element(s) is missing from the merged model.

D.7.1.4. Diagram merge conflicts resulting from UML model updates

Detecting diagram conflicts also has to take into account that the same UML model element can be depicted and thus potentially modified in several diagrams. This has to be taken into account when deciding whether diagram changes are conflicting or not.

D.7.1.4.1. Model change conflicts for diagram symbols — the duality between UML model elements and diagram symbols: If a diagram symbol was deleted in only one evolved model, then the question is whether the deletion change can be accepted. As explained above, it cannot be accepted if there are *diagram* conflicts

(i. e., changes in the other evolved diagram version). Another question is how UML model element changes and conflicts affect the diagram symbol's deletion. Is it conflicting if the model element was modified or if the model element has conflicts? Clearly the modifications of the UML elements must have been initiated from a diagram symbol (since those UML elements cease to exist if all their symbols are removed; i. e., when the last diagram symbol depicting a UML element is deleted, the UML element is deleted too). However, if more than one diagram depicts a certain model element, it cannot be detected in which diagram the UML model element modification were initiated. It might even be that the modification was done via a symbol that was removed afterwards. It is, therefore, difficult to decided for which of a modified UML model element's diagram symbols to accept or reject changes.

Therefore, if a diagram symbol's underlying UML model element(s) have conflicting changes and if this symbol has a deletion change, this change is only accepted if it exists in both evolved diagrams (i. e., the symbol was deleted in both diagrams). If not, the symbol's deletion is conflicting too. This way it is guaranteed that a diagram symbol is not (silently) removed from the merged diagram even though the UML model might have been modified via this symbol in one of evolved diagram versions.

If the diagram symbol's UML element(s) has *non-conflicting* changes, the symbol deletion change is accepted even if it was done in only one evolved diagram. This is feasible because if all symbols of the respective UML element's symbols were deleted in one evolved model, a model element modification conflict would exist (because the model element was deleted too). Therefore, at least one symbol must exist in each evolved model version.

D.7.1.4.2. Diagram symbol existence conflicts resulting from model element changes and existence conflicts: Diagram symbols can only be merged if the underlying UML model element exists in the initial merged model. In case of existence conflicts of their underlying UML model elements, the respective diagram symbols cannot become part of the respective initial merged diagrams. Then, a diagram symbol existence conflict exists for each of these diagram symbols too.

Existence conflicts can also occur for diagram symbols which have not been modified in any evolved diagram. They occur when the underlying UML model element was modified through symbols in different diagrams. For instance, a class could be conflictingly relocated to different packages in both evolved diagrams (i. e., the class has a container conflict). Then, the class cannot become part of the initial merged merged and the conflict has to be manually resolved. If the class again becomes part of the merged model (i. e., when the existence conflict was manually resolved), the diagram symbols, too, can become part of the receptive diagrams.

The following rules regarding diagram symbol existence conflicts can be drawn from the findings above:

- Class symbol:
  - The UML class the symbol references must exist in the merged UML model.
  - If the class symbol is not a root one (i. e., is not referenced by reference property *Diagram::rootClassSymbols*; see Section D.7.1.1), its parent class symbol (*ClassSymbol::parentClassSymbol*) has to exist in the merged diagram (i. e., it has no existence conflict).
- Connection symbol:
  - Both class symbols connected by a connection have to exist in the merged diagram (i. e., they have no existence conflicts).
- Association connection symbol:
  - The underlying UML association has to exist in the merged UML model (i. e., has no existence conflict), and
  - both UML properties belonging to the association have to exist (i. e., they have no existence conflicts), and
  - both UML classes referenced by the properties have to exist in the merged model (i. e., both must not have existence conflicts).
- Generalisation connection symbol:
  - UML generalisation has to exist in the merged UML model (i. e., has no existence conflict), and
  - both UML classes referenced by the generalisation have to exist in the merged model (i. e., both must not have existence conflicts).

Diagram symbol existence conflicts can also occur because a diagram symbol's existence depends on the existence of other diagram symbols in the respective diagram. Situations like the following have to be taken into account during diagram merging:

1. A connection symbol C is added between class symbols depicting class A and B to diagram D in an evolved model. The UML model elements underlying C were not created or modified, they already existed in the common ancestor model.
2. C's underlying relationship (UML model element) is modified in the other evolved model such that it now connects to at least on different class (i. e., class A and/or B are no longer referenced).
3. As a result, C no longer connects the class symbols depicting class A and B in the *merged version* of diagram D, but will have to connect to at least one other class. However, diagram D does not necessarily contain a class symbol depicting this class. Connection symbol C would therefore have at least one dangling end, i. e., it would not be connected to a class symbol on at least one end. Such situations must be avoided because the syntax of a diagram would be violated.

In such cases, the connection symbol addition change becomes a conflicting one because the underlying UML model changed and the connection symbol cannot be added to the diagram. The conflict's type then is a diagram symbol existence conflict.

D.7.2. Differencing OMOS diagrams

D.7.2.1. Matching equivalent diagram elements

As explained in Section D.7.1.1, a OMOS diagram is identified by a globally unique identifier (*Diagram::GUID*) which is assigned when the diagram is created. The diagram's symbols are also identifiable, the identifier is not globally unique, but only unique within this diagram. However, this local uniqueness is sufficient since only the contents (symbols) of a certain diagram are compared and since diagram symbols cannot be moved from one diagram to another.

D.7.2.2. Detecting diagram addition and deletion changes

Similar to merging UML models, in a first step, added and deleted diagrams are detected for each of the two evolved diagram models. To do so, the value of the evolved model's diagrams collection is compared to the common ancestor collection's value. As explained in Section D.5.4.3, a diagram addition or deletion change is registered for each added or deleted diagram, respectively.

Since diagram symbols are local to a diagram, i. e., they cannot be relocated to other diagrams; it follows that when a diagram was added or deleted, its diagram symbols must have been added or deleted as well. Addition or deletion changes are, therefore, also registered for each symbol of each added or deleted diagram respectively.

D.7.2.3. Comparing equivalent diagram model elements

After added and deleted diagrams were detected, equivalent diagrams which exist in the ancestor and the evolved model version are compared. A set of diagram symbols and their identifiers is created for each of the two diagram versions by following the containment reference properties of the diagram model.

D.7.2.3.1. Detecting diagram symbol additions and deletions: Based on the diagram symbols' identifiers of the two to-be-compared diagrams, added and deleted symbols are detected. A diagram symbol addition or deletion change is registered for each added or deleted diagram symbol, respectively.

D.7.2.3.2. Comparing equivalent diagram model elements: After added and deleted diagram symbols were determined, diagram model elements which exist in the ancestor and evolved diagram are compared. In a first step, diagram symbol relocation changes are detected — they can only occur for class symbols, but not for connection symbols because reference *Diagram::connectionSymbols* is the only containment reference property for connection symbols in the diagram meta-model (see Section D.7.1.1).

Finally the property values of all diagram model elements and the diagram element itself are compared including the properties of the diagram root element (for which only the *Diagram::name* property can be modified).

D.7.2.3.3. External model elements: The UML elements referenced by the diagram symbols (see *Symbol::umlElement* in Section D.7.1.1) are not taken into account when diagram are compared. The purpose of this reference property is to link a diagram symbol to the UML element it depicts and to identify the diagram symbol within its diagram. Therefore, the value of this property will not change.

The comparison this property will, therefore, not yield a value property change. And the referenced UML element is, of course, not compared as part of the diagram comparison; this is done during UML model comparison.

The referenced UML element is not taken into account when comparing equivalent OMOS diagrams because it is not *contained* by any of the diagram's elements. When the to-be-compared elements of both diagrams are collected (into an element repository) by following the *containment* reference properties of a diagram, the UML element is not taken into account.

D.7.3. Model consistency checking

The model merge approach presented in this chapter and implemented as part of this research does not take consistency rules defined by the actual meta-model into account when the *initial* merged model is *created.* However, in the case where such consistency rules are defined, the merged model can be checked for consistency once it has been created and repeatedly throughout the whole conflict resolution phase of the model merge process. Applying consistency rules to a model will reference the model elements which violate these rules, the respective merged model elements can thus be annotate with consistency errors and brought to the modeller's attention. It is then possible for modellers to analyse whether the violation of consistency rules was related to model changes and, if so, deal with those changes to make the respective model elements adhere to the consistency rules again.

Consistency rules for models defined by means of the meta-object facility (MOF, see Section 2.1.2.1) meta-models can be defined as part of the meta-model by means of the Object Constraint Language (OCL) [OMG06b]. The UML standard states the following about model consistency [OMG10a, p. 23]*: "The well-formedness rules of the metaclass, except for multiplicity and ordering constraints that are defined in the diagram at the beginning of the package sub clause, are defined as a (possibly empty) set of invariants for the metaclass, which must be satisfied by all instances of that metaclass for the model to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Most invariants are defined by OCL expressions together with an informal explanation of the expression, but in some cases invariants are expressed by other means (in exceptional cases with natural language)."*

Bibliography

[Aon06] Aonix. Ameos CASE Tool. http://www.aonix.com/, 2006. Access date: 4/12/2012.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[BDPP99] Giuseppe Di Battista, Walter Didimo, Maurizio Patrignani, and Maurizio Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size. In *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*, pages 297–310, London, UK, 1999. Springer-Verlag.

[BETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[BFN85] Carlo Batini, L. Furlani, and Enrico Nardelli. What is a good diagram? a pragmatic approach. In *Proceedings of the Fourth International Conference on Entity-Relationship Approach*, pages 312–319, Washington, DC, USA, 1985. IEEE Computer Society.

[BP01] B. Berliner and J. Polk. Concurrent Versions System (CVS). http://www.cvshome.org/, 2001. Access date: 4/12/2012.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 2nd edition, June 2005.

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[BT00] Stina S. Bridgeman and Roberto Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. *J. Graph Algorithms Appl.*, 4(3):47–74, 2000.

[Cav96] A L M Cavaye. Case study research: a multi-faceted research approach for is. *Information Systems Journal*, 6(3):227–242, 1996.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30:232–282, June 1998.

[DFM93] Ed Dengler, Mark Friedell, and Joe Marks. Constraint-driven diagram layout. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 330–335, August 1993.

[DP06] Brian Dobing and Jeffrey Parsons. How UML is used. *Communications of ACM*, 49(5):109–113, 2006.

[DSB98] Peta Darke, Graeme Shanks, and Marianne Broadbent. Successfully completing case study research: combining rigour, relevance and pragmatism. *Information Systems Journal*, 8(4):273–289, 1998.

[EGK$^+$04] Markus Eiglsperger, Carsten Gutwenger, Michael Kaufmann, Joachim Kupke, Michael Jünger, Sebastian Leipert, Karsten Klein, Petra Mutzel, and Martin Siebenhaller. Automatic layout of UML class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208, 2004.

[Eic02a] Holger Eichelberger. Aesthetics of class diagrams. In *Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis, pages 23–31. IEEE*, 2002.

[Eic02b] Holger Eichelberger. Evaluation-report on the layout facilities of UML tools. Technical report, 2002.

[Eic02c] Holger Eichelberger. Sugibib. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, 9th International Symposium, GD '02*, volume 2265 of *Lecture Notes in Computer Science*, pages 467–468. Springer, Springer, 2002.

[Eic03] Holger Eichelberger. Nice class diagrams admit good design? In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 159–167, New York, NY, USA, 2003. ACM Press.

[Eic05] Holger Eichelberger. *Aesthetics and automatic layout of UML class diagrams*. Ph.D. thesis, Fakultät für Mathematik und Informatik, Würzburg University, Germany, 2005.

[Eic06] Holger Eichelberger. On class diagrams, crossings and metrics. In Michael Jünger, Stephen Kobourov, and Petra Mutzel, editors, *Graph Drawing*, Dagstuhl Seminar Proceedings, 2006.

[EKE03] Markus Eiglsperger, Michael Kaufmann, and Frank Eppinger. An approach for mixed upward planarization. *J. Graph Algorithms Appl.*, 7(2):203–220, 2003.

[EKS03] Markus Eiglsperger, Michael Kaufmann, and Martin Siebenhaller. A topology-shape-metrics approach for the automatic layout of UML class diagrams. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 189–ff, New York, NY, USA, 2003. ACM Press.

[ELMS91] P. Eades, W. Lai, K. Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics '91*, pages 24–33, 1991.

[EMdK09] Harald Eisenmann, Juan Miro, and Hans Peter de Koning. MBSE for European Space-Systems Development. *INCOSE INSIGHT*, 12(4), December 2009.

[EvG03a] Holger Eichelberger and Jürgen Wolff von Gudenberg. Demonstration of Advanced Layout of UML Class Diagrams by SugiBib. In *Proceedings of Vissoft 2003, International Workshop on Visualizing Software for Understanding and Analysis*, pages 53–54, 2003.

[EvG03b] Holger Eichelberger and Jürgen Wolff von Gudenberg. UML class diagrams - State of the art in layout techniques. In *Proceedings of Vissoft 2003, International Workshop on Visualizing Software for Understanding and Analysis*, pages 30–34, 2003.

[Fei91] Peter H. Feiler. Configuration management models in commercial environments. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1991.

[FH01] Rudolf Fleischer and Colin Hirsch. Graph drawing and its applications. In Michael Kaufmann and Dorothea Wagner, editors, *Draw-*

ing Graphs: Methods and Models, number 2025 in LNCS, pages 1–22. Springer-Verlag, Berlin, Germany, 2001.

[Fra02] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[Fuj] Fujaba. Fujaba CASE Tool. http://www.fujaba.de/. Access date: 4/12/2012.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley Professional, January 1995.

[GJK$^+$03] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A new approach for visualizing UML class diagrams. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 179–188, New York, NY, USA, 2003. ACM Press.

[Gre02] James Grenning. Extreme programming and embedded software development. In *Embedded Systems Conference*, San Jose, CA, 2002.

[HJvdH06] S.A. Hendrickson, B. Jett, and A. van der Hoek. Layered class diagrams: Supporting the design process. In *Ninth International Conference on Model Driven Engineering Languages and Systems*, October 2006.

[HK99] Jungpil Hahn and Jinwoo Kim. Why are some diagrams easier to work with? Effects of diagrammatic representation on the cognitive intergration process of systems analysis and design. *ACM Trans. Comput.-Hum. Interact.*, 6(3):181–213, 1999.

[HN99] W. Hermsen and K.-J. Neumann. Object-oriented modeling concept for software of electronic control units in vehicles. *it+ti - Informationstechnik und Technische Informatik*, 5, 1999.

[HN00] W. Hermsen and K.-J. Neumann. Application of the object-oriented modeling concept OMOS for signal conditioning of vehicle control units. Technical report, SAE 2000 World Congress, March 2000, Detroit, MI, USA, 2000.

[HOS90] William H. Harrison, Harold Ossher, and Peter F. Sweeney. Coordinating concurrent development. In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, CSCW '90, pages 157–168, New York, NY, USA, 1990. ACM.

[IBM06] IBM. Rational Software Architect. http://www.ibm.com/, 2006. Access date: 4/12/2012.

[IL06] I-Logix. Rhapsody CASE Tool. http://www.ilogix.com/, 2006. Access date: 4/12/2012.

[Jac04] Nico Jacobs. *Relational sequence learning and user modelling.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Science, October 2004.

[JM03] Michael Jünger and Petra Mutzel. *Graph Drawing Software.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[Kos03] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.

[KTW97] Justus Klingemann, Thomas Tesch, and Jürgen Wäsch. Enabling co-operation among disconnected mobile users. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems*, COOPIS '97, pages 36–46, Washington, DC, USA, 1997. IEEE Computer Society.

[KW01] Michael Kaufmann and Dorothea Wagner, editors. *Drawing graphs: methods and models.* Springer-Verlag, London, UK, 2001.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. A generic difference algorithm for UML models. In *Proceedings of the SE 2005*, Essen, Germany, March 2005.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[Leb95] David B. Leblang. *The CM challenge: configuration management that works*, pages 1–37. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Let05] Kim Letkeman. Comparing and merging UML models in IBM Rational Software Architect. http://www-

128.ibm.com/developerworks/rational/library/05/712_comp/, July 2005. Access date: 4/12/2012.

[LLY06] Yi-Yi Lee, Chun-Cheng Lin, and Hsu-Chun Yen. Mental map preserving graph drawing using simulated annealing. In *APVis '06: Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation*, pages 179–188, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[LS87] H. J. Larkin and A. H. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:69–99, 1987.

[Lub03] Sam Lubbe. The development of a case study methodology in the information technology (IT) field: a step by step approach. *Ubiquity*, 2003, September 2003.

[LvO92] Ernst Lippe and Norbert van Oosterom. Operation-based merging. In *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, SDE 5, pages 78–87, New York, NY, USA, 1992. ACM.

[LWWC11] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. From UML Profiles to EMF Profiles and Beyond. In *Objects, Models, Components, Patterns - 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings*, pages 52–67, 2011.

[Mai78] David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978.

[MB02] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[MBZR03] T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution, 2003.

[MELS95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[Men02] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 1997.

[MF93] P. Moore and C. Fitz. Gestalt: Theory and instructional design. *Journal of Technical Writing and Communication*, (2):137–157, 1993.

[MGH05] Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2005. ACM Press.

[mic06] microTOOL. ObjectIF UML Modeler. http://www.microtool.de/, 2006. Access date: 4/12/2012.

[MKY06] David Mandelin, Doug Kimelman, and Daniel Yellin. A bayesian approach to diagram matching with application to architectural models. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 222–231, New York, NY, USA, 2006. ACM Press.

[Moo10] Daniel L. Moody. The physics of notations: a scientific approach to designing visual notations in software engineering. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 485–486, 2010.

[Nie04] Jürgen Niere. Visualizing differences of UML diagrams with Fujaba. In *Proceedings of the Fujaba Days 2004*, 2004.

[NoM06] NoMagic. MagicDraw UML. http://www.magicdraw.com/, 2006. Access date: 4/12/2012.

[Nor95] Stephen C. North. Incremental layout in dynadag. In Franz-Josef Brandenburg, editor, *Graph Drawing, Symposium on Graph Drawing, GD '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418, Passau, Germany, September 1995. Springer.

[Ohs02] Dirk Ohst. A fine-grained version and confguration model in analysis and design. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, ICSM '02, pages 521–, Washington, DC, USA, 2002. IEEE Computer Society.

[OMG03] OMG. *UML Notation Guide.* Object Management Group, 2003.

[OMG04] OMG. *Human-Usable Textual Notation (HUTN) Specification.* Object Management Group, http://www.omg.org/spec/HUTN/1.0/, 2004. Access date: 4/12/2012.

[OMG05] OMG. *MOF 2.0/XMI Mapping Specification.* Object Management Group, http://www.omg.org/spec/XMI/2.1/, 2005. Access date: 4/12/2012.

[OMG06a] OMG. *Meta Object Facility (MOF) Core Specification.* Object Management Group, http://www.omg.org/spec/MOF/2.0, version 2.0 edition, 2006. Access date: 4/12/2012.

[OMG06b] OMG. *Object Constraint Language (OCL), v2.0.* Object Management Group, 2006.

[OMG06c] OMG. *UML Diagram Interchange version 1.0.* Object Management Group, http://www.omg.org/spec/UMLDI/1.0/, 2006. Access date: 4/12/2012.

[OMG10a] OMG. *UML 2.3 Infrastructure Specification.* Object Management Group, http://www.omg.org/spec/UML/2.3/, 2010. Access date: 4/12/2012.

[OMG10b] OMG. *UML 2.3 Superstructure Specification.* Object Management Group, http://www.omg.org/spec/UML/2.3/, 2010. Access date: 4/12/2012.

[OMG11] OMG. *Diagram Definition 1.0 - Beta 2.* Object Management Group, http://www.omg.org/spec/DD/1.0/Beta2/, 2011. Access date: 4/12/2012.

[OWK03a] Dirk Ohst, Michael Welle, and Udo Kelter. Difference tools for analysis and design documents. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[OWK03b] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, New York, NY, USA, 2003. ACM Press.

[OWK04] Dirk Ohst, Michael Welle, and Udo Kelter. Merging UML documents. Technical report, University of Siegen, Germany, 2004.

[PCA02] Helen C. Purchase, D. A. Carrington, and J-A. Allder. Graph layout aesthetics in UML diagrams. *Journal of Graph Algorithms and Applications*, 6(3):255–279, 2002.

[PCM$^+$01] Helen C. Purchase, Linda Colpoys, Matthew McGill, David Carrington, and Carol Britton. UML class diagram syntax: An empirical study of comprehension. In Peter Eades and Tim Pattison, editors, *Australian Symposium on Information Visualisation, (invis.au 2001)*, Sydney, Australia, 2001. ACS.

[Pet95] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.

[PMCC01] Helen C. Purchase, Matthew McGill, Linda Colpoys, and David Carrington. Graph drawing aesthetics and the comprehension of UML class diagrams: An empirical study. In *APVis '01: Proceedings of the 2001 Asia-Pacific symposium on Information visualisation*, pages 129–137, Darlinghurst, Australia, 2001. Australian Computer Society, Inc.

[Pur04] Helen C. Purchase. Evaluating graph drawing aesthetics: defining and exploring a new empirical research area. In J. DiMarco, editor, *Computer Graphics and Multimedia: Applications, Problems and Solutions*, pages 145–178. Ed. Idea Group Publishing, 2004.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual.* Addison-Wesley Professional, 2nd edition, July 2004.

[RW98] Jungkyu Rho and Chisu Wu. An efficient version model of software diagrams. In *APSEC '98: Proceedings of the Fifth Asia Pacific Software Engineering Conference*, page 236, Washington, DC, USA, 1998. IEEE Computer Society.

[Sam02] Miro Samek. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems.* CMP Books, 2002.

[SB05] Markus Schweizer and Michael Benkel. Development of product families — an example from the automobile industry. In *Third Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER3)*, 2005.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks.

EMF: Eclipse Modeling Framework. Addison-Wesley, Boston, MA, 2. edition, 2009.

[SDNB04] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. Covamof: A framework for modeling variability in software product families. In *Proceedings of the Third International Conference on Software Product Lines (SPLC)*, pages 197–213, 2004.

[See97] Jochen Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In G. DiBattista, editor, *Proc. Graph Drawing, 5th International Symposium, GD '97, Rome, Italy, September, 1997*, volume 1353 of *LNCS*. Springer, 1997.

[Sel03] Petri Selonen. Set operations for unified modeling language. In *Proceedings of the Eight Symposium on Programming Languages and Tools, SPLST 2003*, pages 70–81, 2003.

[SG09] Andrew Stellman and Jennifer Greene, editors. *Beautiful Teams: Inspiring and Cautionary Tales from Veteran Team Leaders.* O'Reilly, Beijing, 2009.

[SM88] Sally Shlaer and Stephen J. Mellor. *Object-oriented systems analysis: modeling the world in data.* Yourdon Press, Upper Saddle River, NJ, USA, 1988.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understandings of hierarchical system structures. *IEEE Transactions in Systems, Man, and Cybernetics*, smc-11(2):109–125, February 1981.

[Sub] Apache Subversion. Subversion homepage. http://subversion.apache.org/. Access date: 4/12/2012.

[Sug02] Kozo Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*, volume 11 of *Series on Software Engineering and Knowledge Engineering.* World Scientific, 2002.

[Tam85] Roberto Tamassia. New layout techniques for entity-relationship diagrams. In *Proceedings of the Fourth International Conference on Entity-Relationship Approach*, pages 304–311, Washington, DC, USA, 1985. IEEE Computer Society.

[TBB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, 18(1):61–79, 1988.

[TH02] Steffen Thiel and Andreas Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72, 2002.

[TH03] S. Tilley and S. Huang. A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In *SIGDOC '03: Proceedings of the 21st Annual International Conference on Documentation*, pages 184–191. ACM Press, 2003.

[Vis] Visual Paradigm. Visual Paradigm for UML. http://www.visual-paradigm.com/. Access date: 4/12/2012.

[VS06] Markus Völter and Thomas Stahl. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, June 2006.

[WEK02] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yFiles: Visualization and automatic layout of graphs. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing, Vienna, Austria, September 23-26, 2001*, pages 453–454. Springer, 2002.

[Wes91] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 68–79, New York, NY, USA, 1991. ACM Press.

[Wes10] Bernhard Westfechtel. A formal approach to three-way merging of EMF models. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 31–41, New York, NY, USA, 2010. ACM.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21:168–173, January 1974.

[WHF93] Colin Ware, David Hui, and Glenn Franck. Visualizing object oriented software in three dimensions. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 612–620. IBM Press, 1993.

[Wika] Wikipedia. External validity. http://en.wikipedia.org/wiki/External _validity. Access date: 4/12/2012.

[Wikb] Wikipedia. Internal validity. http://en.wikipedia.org/wiki/Internal_validity. Access date: 4/12/2012.

[WL99] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family Based Software Development Process*. Addison-Wesley, 1999.

[WPCM02] C. Ware, Helen C. Purchase, L. Colpoys, and M. McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.

[Yin03] Robert K Yin. *Case Study Research: Design and Methods*, volume 5. Sage Publications, 2003.