Concurrent Requirements Engineering with a UML Subset based on Component Schema Relationships

vorgelegt von Diplom Informatiker Jan Wortmann
aus Bremen

Eingereicht an der Fakultät IV- Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

Promotionsausschuß:
Vorsitzender: Prof. Dr. Klaus Obermayer
Berichter: Prof. Dr. Herbert Weber
Berichter: Prof. Dr. Michael Goedicke

Tag der wissenschaftlichen Aussprache: 2 Juli 2001

Berlin 2001
D 83

The beginning of of wisdom is the definition of terms
[Socrates]

Abstrakt

Die vorliegende Arbeit widmet sich der konkurrenten Anforderungsanalyse für große Softwaresysteme am Beispiel der Flugsicherung. Eine Zerlegung der Gesamtaufgabe in kleinere Teile und die Darstellung von Teillösungen, sogenannten *Komponentenschemata*, die von jeweils einer Gruppe von Analytikern bearbeitet werden, erlaubt eine Durchführung des Projekts in einem vertretbarem Zeitrahmen. Das Vorgehen wirft jedoch die Frage auf, wie eine kohärente Beschreibung der Gesamtaufgabe auf Basis der unabhängig von einander entstandenen *Komponentenschemata* erzielbar ist, ohne Modellelemente bei der abschließenden Integration zu überschreiben.

Die in *Komponentenschemata* festgehaltenen Anforderungen werden auf Basis der objektorientierten Modellierungstechnik UML dargestellt. Eine Untersuchung des UML Metamodells identifiziert geeignete UML Sichten und *Konstrukte* für eine lösungsneutrale Aufgabenbeschreibung. Die in *Komponentenschema* modellierten *Dinge* der realen Welt heißen *Artefakte* und bestehen aus einem *Terminus* sowie einem *Konstrukt*.

Termini sind Wörter mit einer speziellen Bedeutung in einem gegebenen Fachgebiet. Eine konsistente Benutzung eines *Terminus* innerhalb eines *Komponentenschemas*, führt aber nicht automatisch zu der konsistenten Benutzung des *Terminus* in allen die Gesamtaufgabe beschreibenden *Komponentenschemata*. Eine *Korrespondenzvermutung* gilt daher für zwei *Artefakte* deren *Termini* das gleiche *Konzept* ausdrücken und das gleiche *Konstrukt* benutzen. Lexikalische Datenbanken und Ontologien verwalten die von *Termini* ausgedrückten *Konzepte* in einem *Konzeptgraphen* und tragen somit zur Aufstellung von *Korrespondenzvermutungen* bei.

Ausgehend von den Begriffen der Schemaintegration, detailliert die Arbeit die verschiedenartigen *Korrespondenzvermutungen* zwischen *Artefakten* zweier *Komponentenschemata* unter Verwendung der Modellierungstechnik UML an ausgewählten Beispielen der Flugsicherung. Die Arbeit zeigt Wege zur Ermittlung von *Korrespondenzvermutungen* auf, untersucht den Einfluß verschiedener Projektorganisationen auf die zu erwartenden *Korrespondenzvermutungen*. Ferner untersucht die Arbeit den nötigen Prozeß der von *Korrespondenzvermutungen* über *Korrespondenzzusicherungen* zur Integration der *Komponentenschemata* führt.

Die Arbeit stellt ein Werkzeugkonzept vor, das die konkurrente Anforderungsanalyse einer Untermenge der UML Konzepte auf Basis von Komponentenschemata wirkungsvoll unterstützt soll. Dabei wird insbesondere die Integration auf dem Markt befindlicher Produkte, wie Modellierungswerkzeuge, lexikalische Datenbanken, Ontologien diskutiert. Nach einer Bewertung der Modellierungstechnik UML anhand in der Flugsicherung durchgeführter Projekte faßt die Arbeit das erzielte Ergebnis zusammen und identifiziert den daraus erwachsenen Forschungsbedarf im Bereich des Continuous Software Engineering.

Acknowledgements

First of all I would like to thank Prof. Herbert Weber as the advisor of this thesis for the fruitful discussions throughout the entire project. I also would like to thank Prof. Michael Goedicke from Universität Essen, who spent the time and the efforts to accept the co-referate. I am especially greateful for his clear and structured comments.

I am greatfully indebted to my wife Luisa Callejón. Her motivation, her patience and her optimism help me to overcome the typical obstacles in the course of a dissertation project. Her enthusiams lead me through all those difficult moments and helped me to get back on the track.

Then, I also take the opportunity to thank my colleagues at the research group CIS at Technische Universität Berlin. The discussions with Susanne Busse and Markus Klar helped me to concretise the ideas presented in this work. Lutz Friedel for the excellent maintenance of our working environment and Michael Löwe and Ingo Claßen who lead me through the first time at CIS.

My graduate students Ute Schirmer, Dirk Kurzmann and Uta Glaß had a very important influence on this work. Their curiosity and their constant enthusiasm in the elaboration of their Diplomathesis gave me important input to go ahead in this project.

I also would like to thank Gerd Klawitter from the EUROCONTROL Experimental Centre for the opportunity to stay with his team. That was a unique possibility to get into the exiting world of air traffic management and has to be considered as the starting point for this thesis.

In particular I would like to thank Jerry Watson from EUROCONTROL for the terrific colaboration in the consultancy project in Brussels, that provided the initial problem statement for this work. Many thanks also to the other project participants Patrick Abbott, Marc Bourgois and Gerd Lauter.

Finally I would like to thank my friends Andreas, Rolf, Doris and Gaby for their patience their support and their believe. So many times they had to share the frustration of little advances in the thesis.

Contents

List of Figures

List of Tables

Abbreviations

AAS	Advanced Automated System
ATC	Air Traffic Control
ATFM	Air Traffic Flow Management
ATM	Air Traffic Management
ATMS	Air Traffic Management System
ATSU	Air Traffic Service Unit
C^3I	Command, Control, Communication and Intelligence
CMM	Capability Maturity Model
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
EATCHIP	European Air Traffic Control Harmonisation and Integration Programme
EUROCONTROL	European Organisation for the Safety of Air Navigation
FAA	Federal Aviation Agency
IEEE SRS	Software requirements specification following the IEEE-830 standard
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Modelling Technique
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OODA	Object-Oriented Domain Analysis
OOPA	Object-Oriented Problem Analysis
OOPL	Object-Oriented Programming Language
OORA	Object-Oriented Requirements Analysis
ORD	Operational Requirements Document
OOSRS	Object-Oriented Software Requirements Specification
SRS	Software Requirements Specification

StP	Software through Pictures
UML	Unified Modelling Language
UML SD	UML Sequence Diagram
UML SSD	UML Static Structure Diagram
UML UCD	UML Use Case Diagram

1 Introduction

Requirements engineering for industry-scale software systems marks the first and least formalized part within the entire software development cycle. The central outcome of requirements engineering is a software requirements specifications (SRS), as an agreed statement of software requirements for customers, end-users and software developers. The specification of user needs is one of the most complicated activities in industry-scale systems. The most challenging factors in this context are system size and characteristics. System size results in the necessity of performing the analysis concurrently. The entire problem statement has to be divided into smaller *initial partitions*, which can be elaborated concurrently by individual analyst teams. Otherwise requirements engineering cannot be concluded in a reasonable time frame. System characteristics comprise properties such as concurrency of execution and the predictability of output. Air traffic management, as the sample domain for this thesis, is a *command, control, communication and intelligence* (C^3I) application, where air traffic controllers maintain an orderly flow of air traffic. The system is permanently reacting to changes in its environment. System characteristics also incorporate its users and their various skills as the central input for the requirements. This situation leads to a critical problem in that the individual demand of a variety of system users with different technical backgrounds, have to be fulfilled.

The diversity of people affected by a future system as well as the limited knowledge concerning the structure of a future system make industry-scale requirements engineering projects so hard. Knowledge concerning the problem statement at hand is often not available in a structured form and mostly based on the individual expertise of business experts. The design of computational systems, however, demands for a precise and robust description of user needs that provides efficient and solution neutral input for designers. Furthermore software requirements specifications should also provide a solid basis for conformance testing.

Besides problem size and the variety of system users, modelling techniques for specification of user requirements play an important role. Applied modelling techniques give rise to the question of expected results. Each of them has its own advantages and, more importantly its restrictions. Consequently, the results of requirements specifications are influenced from four different directions, namely the computational complexity of the application domain, the skill of users as well as analysts gathering requirements and the modelling technique applied to describe the demand. Of the available modelling techniques, object-oriented techniques are rapidly gaining popularity in a variety of application domains. Of the various object-oriented analysis techniques, the Unified Modelling Language (UML) is becoming more and more popular in requirements engineering. From a scientific point of view the question of

efficiency of a given technique with respect to the expected results is extremely relevant. Project experience in air traffic management motivated this thesis and also gave rise to the studied factors in requirements engineering throughout this work.

1.1 Motivation and Objectives

This work is based on experience gained in series of requirements engineering projects in the air traffic management domain and therefore reflects a number of particularities with respect to organisational as well as technical aspects of this highly sensitive domain. This section introduces the empirical background and the objectives of this work.

1.1.1 Motivation: Requirements Analysis in the Air Traffic Management Domain

Implementation projects in the air traffic management domain recently suffered a number of costly failures. The Federal Aviation Agency's (FAA) Advanced Automated System (AAS) is one of the most spectacular failures in history of both air traffic management and software development [Gla97]. Proper software requirements specifications form the central prerequisite for the design of computational systems. But an adequate specification of requirements in air traffic management has been and still is a demanding challenge for a number of reasons. This subsection outlines the air traffic management domain under special consideration of its objectives, involved stakeholders, business activities and its operational infrastructure.

Complexity of Air Traffic Management Systems

Air traffic management (ATM), as a *command, control, communication and intelligence* (C^3I) application, provides controllers with efficient means to maintain an orderly flow of air traffic. The ATM domain basically incorporates two linked planning cycles with differing time horizons. Air Traffic Flow Management (ATFM) aims for a centralised regulation of air traffic before take-off. Smoothing peaks in individual air traffic service units (ATSU) and reducing holding patterns around major airports are the central objectives of this system. While ATFM regulates the air traffic before take-off, air traffic control actually realises the control of airborne traffic.

Controller Activities

In order to characterize the intricacy of air traffic management, we want to sketch the work of air traffic controllers. A controllers' central duty is maintaining an orderly flow of air traffic. Radar and the provision of flight data are the controller's basic aids in this context. Besides automated assistance, the controller's ability to construct a four dimensional mental image – latitude, longitude, altitude and time – on the basis of a two dimensional (radar) screen is the essential characteristic of this application domain. This ability documents the problems encountered by analysts gathering requirements for automated air traffic management systems. Besides the complicated creation of this mental image, there is little documentation on how controllers actually do their work. External observers are often quite surprised about the particular differences in the way each controller manages his daily routine.

This lack of well-documented business processes standardised on a European level further augments analysis complexity for analysts in the ATM domain. Although this paragraph can only provide a rough outline of controller activities, it emphasises the fundamental difficulty within the specification of controller demands.

Air Traffic Management Systems

Air traffic management systems (ATMS) comprise a number of demanding characteristics: they are distributed, are not operated under a central authority and are safety critical. Their availability of 99.99998% equals a maximum down time of 10 minutes a year. The relationship between pilots along with aircraft and controllers in particular is extremely complicated. The aircraft is tracked by a radar system and its identity is provided by a four bit octal code, the so-called SSR code. Today the SSR code space is far too small for the amount of airborne traffic, that means two aircraft may be assigned to the same code. At the same time the relationship between controllers and pilots is maintained via very high frequency (VHF) radio communication. In order to provide controllers with pilot intentions, a flight plan with a route from the departure to the destination aerodrome has to be submitted to ATC authorities. Flight data processing systems finally combine radar observations with route information and therefore assist controllers in assuring an orderly flow of air traffic.

EUROCONTROL and EATCHIP

From a legal as well as a technical point of view, the European ATM infrastructure is characterised by an enormous degree of heterogeneity, which led to the foundation of the European Organisation for the Safety of Air Navigation - EUROCONTROL in the 1960 and also initiated the European Air Traffic Control Harmonisation and Integration Programme (EATCHIP) in the mid 1980. The EATCHIP programme aims for a harmonisation and integration of the current European ATM infrastructure at three reference levels, tailored for the varying traffic density throughout Europe. The basis of this program is the definition of agreed requirements under the direction of EUROCONTROL. The resulting requirements specification should facilitate the integration of the currently nationalised ATM infrastructure. These requirements specifications are based on the objective of providing efficient and cost-effective means of managing the rapidly increasing rates of air traffic. The complex nature of controller activities, the heterogeneity and availability of operational systems, cannot be mastered without extensive requirements analysis. Providing thorough understanding of the intricate nature of this C^3I application is the objective of problem analysis in the ATM domain. This understanding, documented in a software requirements specification, provides the input to designers building the system and its users for an assessment of the conformance of the implemented system. The last point in particular makes intensive requirements analysis in the air traffic management domain such a crucial activity.

Requirements Specifications Based on Textual Descriptions

Requirements specifications in the ATM domain currently follow functional decomposition of the problem statement and usually comprise hundreds of thousands of lines of plain text statements. To exemplify this situation, we give a short example, taken from one of the EATCHIP operational requirements documents (ORD):

> "FDPD-FDMD-23 Manual modifications of relevant SFPL data shall trigger trajectory prediction re-calculation" [Eur97, p. 9-5]

Even when written in a concise and comprehensive manner, i.e. excluding the inherent ambiguity of natural language, the sheer amount of requirements statements in industry-scale systems is difficult to understand and their quality is hard to assess. Especially contradiction and completeness cannot easily be determined. Analysis complexity has been reduced in the EATCHIP context through a division of the overall problem statement into seven initial partitions. These Domains – in EUROCONTROL jargon – have been elaborated concurrently and each of them resulted in an operational requirements document comprising over 300 pages of written text. Due to concurrent elaboration, these initial partitions also give rise to the question of interfacing among them and with respect to global requirements impacting on the entire system. Sheer size of textual requirements does not facilitate a thorough understanding of the problem particularities in air traffic management.

Modelling Techniques in Requirements Engineering

Modelling techniques were an important innovation in the 1970 with the objective of accelerating problem understanding and improving communication between analysts, business experts and software designers. Compared to plain-text documents, abstract models reduce the ambiguity of natural language, regardless of the technique applied. Their comprehensibility, however, has always been under debate, by both domain experts as well as the scientific community. Applied modelling techniques range from semi-formal techniques, such as UML [BJR98], to formal ones like LOTOS [ISO8807]. Formal methods, however, suffer serious problems with respect to application size. Usually small problem statements require extensive efforts for modelling them precisely. Only carefully chosen critical parts of industry-scale applications can efficiently be modelled using formal methods. Currently object-oriented modelling techniques, like OMT [RBP+91] or UML [BJR98] are gaining acceptance in a variety of application domains and have also been used throughout our projects in the ATM domain.

UML in the ATM Domain

Today there is still little documented experience concerning the use of object models in requirements engineering for industry-scale applications. Especially in the air traffic management domain, which is traditionally related to functional decomposition techniques, object models are a rather novel approach. Within object-oriented modelling techniques the Unified Modelling Language (UML) is becoming an industry standard through its OMG certification. In our projects UML has been used to model the textual requirements in the original operational requirements documents.

1.1.2 Objectives of this Work

The experience gained in a series of projects in the air traffic management domain has been extrapolated to a slightly different approach. While the reference projects started on the basis of textual requirements specifications, this work takes complex use cases [JCJ+94, Wor96] as its central requirements basis.

The entire problem statement is split into an arbitrary set of initial partitions that are to be elaborated concurrently. A subset of the UML notation, comprising use cases, static structure and sequence diagrams, has been used for problem analysis purposes. The variety of UML views aims to support the entire software life-cycle. Therefore a specialised subset has to be tailored for requirements engineering, in order to avoid premature design and to facilitate the comprehensibility of the modelling techniques to involved stakeholders with varying IT background.

Component schemas are persistently stored in distributed repositories, corresponding to the viewpoint idea of Nuseibeh [NKF94]. Based on the results of projects in the air traffic management domain [Wor96, Eur98a], we discuss the impact of different project organisations on the specifications, under special consideration of the integration of the partitions towards an integrated minimal problem analysis model. In this context we define *correspondence*, *conflicts* and *dependencies* as relevant *component schema relationships* for partitions specified in the selected UML subset. We propose means for detecting *component schema relationships* and realise them in terms of the UML metamodel. Finally we detail organisational issues to resolve potential incoherence generated by component schema relationships. Within this general context we pursue the following objectives.

Evaluation of UML in Industry-Scale Applications

To date UML has been used in a number of small to medium sized projects, so the first objective is an evaluation of UML's principal suitability for requirements engineering in industry-scale systems. This objective also includes a brief comparison with other object-oriented analysis styles. UML offers a number of different views [RBJ99] defined on the basis of a common metamodel. These UML views have to be evaluated with respect to their suitability in requirements engineering, i.e. preserving a solution-neutral description of the problem statement at hand. Then available *constructs* within the selected UML views have to be examined under the same objective. In this context our attention is focused on constructs stemming from distinct programming languages. Finally we propose a UML subset tailored for requirements engineering. This objective is related to the demand for suitable modelling techniques for industry-scale problem statements, in order to facilitate thorough understanding of arbitrary problem statements and speed up the creation of expressive software requirements specifications. Furthermore the selected UML subset should also suit the diverse backgrounds of involved stakeholders from business users via analysts and domain experts to senior management. Note that the discussion of UML views and *constructs* is mostly based on the original 1997 metamodel, as submitted to the OMG [OMG97a, OMG97b, OMG97c]. The recent revision of the metamodel [Rat99] and the companion literature [BJR98, RJB99] only became available by the end of this work.

Concurrent Elaboration of Partitions

The second objective pivots around the question of concurrent requirements elicitation using the defined UML subset. Several analyst teams are assigned to given fractions of the overall problem statement and concurrently elaborate these *initial partitions* using the selected UML subset. Then the results, so-called *component schemas*, have to be compared to each other, leading to a definition of *component schema rela-*

tionships. Component schema relationships indicate potential threats to the consistency of the overall problem analysis model and therefore have to be evaluated carefully. Clarifying the nature and character of component schema relationships forms a major part within this objective. A comparison to concurrent design highlights the special difficulty of performing requirements elicitation concurrently. While concurrent design aims for the elaboration of partition contents on the basis of a stable abstract model with a defined partition boundary, the creation of model *and* boundary is the central objective of concurrent requirements analysis. Contents and environment of given initial partitions are not known at project begin in concurrent requirements analysis and therefore mainly represent an organisational unit for the assigned analyst team. Performing requirements analysis concurrently in a reasonable timeframe without compromising the quality of its results is a central motivation for this objective.

The Impact of Different Project Organisations

Clarifying the impact of project organisation on component schema relationships marks the third objective. Project organisation in our work comprises two major aspects: the organisation of analyst teams and project management, as well as the assignment of partitions to repositories. Although at first glance building *initial partitions* could be regarded as an integral project organisation activity, we leave out this question. The nature of initial partitions varies so much over application domains that common rationale for building them can hardly be expected. Analysing the influence of project *and* repository organisation on expected *component schema relationships* still gives rise to a sufficient number of questions in requirements engineering.

Managing Component Schema Relationships

On the basis of the previous objectives, strategies for the detection and management of *component schema relationships* have to be analysed under special consideration of the different project organisations. In this context, means for detecting, propagating and resolving *component schema relationships* have to be defined. Minimizing conceptual redundancy and assuring proper interfacing among partitions indirectly improves the quality of complex requirements specifications documenting industry-scale problem statements, as encountered in the air traffic management domain.

Improving the Quality of SRS

Evaluating *component schemas* marks the final objective of this work, where we compare them to an agreed software requirements specification standard [IEEE830]. This objective is intended to reveal potential shortcomings of *component schemas* with respect to ‘good’ software requirements specifications. *Component schemas* model structural as well as behavioural requirements in method-specific form. In this context we analyse which kind of requirements fall short and also propose enhancements to *component schemas* that allow persistent storage of previously unreflected requirements. The overall project efficiency and the quality of the resulting software requirements specifications should as a result be improved. This objective is highly relevant for outsourcing design and implementation of computational systems. In this context, software requirements specifications is to provide a basis for contractors to build the ‘right’ system and for clients a solid foundation for assessing the conformance of an implemented system with respect to the specified demand.

The formulated objectives stem from experience gained in a series of requirements analysis projects in the air traffic management domain. The problem statement, air traffic management in Europe, has been split into seven *initial partitions* by the project management. Each of these seven *initial partitions* has been specified entirely in terms of textual requirements, providing the initial requirements basis. Modelling the requirements statements of each *initial partition* in terms of *component schemas* provided a compact and robust description of the analysed fractions of the problem statement. The *component schemas* revealed vagueness and omission within given partitions, as well as improper interfacing between partitions. The amalgamation of the *component schemas* towards an integrated problem analysis model covering the entire air traffic management domain encountered a number of complications, stemming from shortcomings of UML, as well as insufficient CASE tool support. Nevertheless, *component schemas* provided a comprehensible outline of the detailed ORDs. Interfacing between partitions was able to be reviewed by the involved domain experts and an integrated minimal model, covering the entire application domain, was created. All examples used throughout this work relate to these projects, carried out in close collaboration with domain experts from the European Organisation for the Safety of Air Navigation – EUROCONTROL.

1.2 Outline of this Thesis

This section describes the organisation of this work and outlines the contents of individual chapter.

Chapter 2

Chapter 2 summarizes the related work. Requirements engineering techniques and object-oriented modelling provide the foundation of this work. Linguistic and ontological approaches clarify the use of language in *component schemas*. View integration techniques provide the basics for the definition of *component schema relationships*. Finally this chapter identifies missing bits with respect to the objectives.

Chapter 3

Chapter 3 analyses stakeholders, processes and techniques for concurrent requirements engineering. The basic terminology for models, viewpoints and project participants in requirements engineering is introduced here. Then the principal character of requirements is discussed. Different object-oriented analysis styles are briefly compared to each other, in order to identify the most effective one for requirements engineering purposes.
This chapter defines our perspective with respect to requirements engineering in general and identifies the modelling technique as well as the analysis style used throughout this work.

Chapter 4

Requirements engineering has to be conducted concurrently in industry-scale application domains, since the problem cannot otherwise be mastered in a reasonable time frame. Identifying the most important factors in industry-scale projects, involving several man years for requirements analysis, is the subject of Chapter 4.

Additional project participants, the process applied and modelling techniques in our reference projects are introduced. Analysing a problem statement concurrently gives rise to questions with respect to project and repository organisation. Especially project organisation significantly influences the *coherence* of the resulting requirements specifications. *Autonomy* enables analyst teams to elaborate their *initial partition* in isolation, without taking the knowledge of other analysts into account. On the one hand, *coherence* reflects two essential demands of project management. Budgetary and temporal control of the overall project progress is a must in modern management. On the other hand, the need of a complete and consistent problem analysis model, as a central prerequisite for the elaboration of a detailed and expressive software requirements specification, characterizes the second management demand. This chapter extends the perspective of requirements engineering towards concurrent requirements engineering, calling for a number of clarifications not covered in the requirements engineering literature.

Chapter 5

The use of UML in concurrent requirements engineering is the primary concern in Chapter 5. The chapter opens with a discussion of the nature of UML and its notation. Then UML is evaluated with respect to its suitability in requirements engineering. Under the objective of providing a solution-neutral problem description, we identify suitable UML views and discuss all *constructs* within these selected UML views. Avoiding premature design in software requirements specifications and providing comprehensive problem descriptions that suit a broad target audience, from system users to software designers, is the central concern of this chapter. This discussion is entirely based on an examination of the UML metamodel [OMG97a] defining the semantics of this modelling technique. Finally, selected UML views and the chosen *constructs* within these UML views establish our requirements basis. Semantics and notation of this UML subset applied in concurrent requirements analysis leads to our definition of *component schemas*.

Chapter 6

Natural language can be considered as the most important input to requirements engineering. Its inherent ambiguity is one of the major problems in this discipline. Therefore Chapter 6 analyses the character and occurrence of natural language in *component schemas*. *Terms*, *artefacts* and *concepts* provide the central prerequisite for the definition of *component schema relationships*, elaborated in the next chapter.

Chapter 7

The use of language and the differing perspectives of stakeholders are the central factors for emerging component schema relationships, discussed in Chapter 7. Based on the definition taken from the schema integration community, we define potential conflicts for our UML subset selected for *component schemas*. Incorporating linguistic aspects into this merely structural definition of conflicts leads to the notion of *component schema correspondence*, *conflict and dependency*. Examples of component schema relationships are taken from our projects in the air traffic management domain.

Chapter 8

Possible means for detecting and managing *component schema relationships* are discussed in Chapter 8. On the basis of a pragmatic model for assessing *component schema relationships*, we discuss the influence of the particular project organisation on the encountered *component schema relationships*. Automated as well as manual means may be used for the clarification of *terms* used in *component schemas*. The chapter closes with organisational aspects regarding the *notification*, *coordination* and *resolution* of *component schema relationships* among the involved analyst teams and the introduction of a process model.

Chapter 9

Tool support for concurrent requirements engineering using *component schemas* is the central topic of Chapter 9. In this context, linguistic support, communication infrastructure and CASE tool capabilities form the major components, leading to two potential solutions. An integrated solution combines all three components into a single tool, supporting the stakeholders in concurrent requirements analysis for industry-scale problem statements. The second solution is based on a combination of off-the-shelf tools.

Chapter 10

Our proposed solution for concurrent requirements engineering is elaborated in Chapter 10. We apply three different perspectives towards *component schemas* in industry-scale problem statements. The first perspective mostly deals with the expressive power of *component schemas constructs*. Here we have a closer look at the specific way given aspects of the problem statement have been reflected in *component schemas*. The specific problems encountered in our air traffic management projects provide the second applied perspective. Here we are mostly interested in the question of how existing tool support could improve given projects. Finally, we analyse the role of *component schemas* in requirements engineering projects. Overall project efficiency can be dramatically improved by using the information comprised in *component schemas* for the generation of software requirements specifications. On the basis of the IEEE Software requirements specification standard [IEEE830], we discuss requirements for how currently missing requirements may be represented in *component schemas* and propose a suitable outline for generating software requirements.

Conclusion & Appendix

Concluding remarks and future research areas mark the end of this work. *Component schema relationships* incorporating both structural and naming conflicts, are presented as suitable support for concurrent requirements engineering. On the basis of autonomous analysts teams, the coherence of a requirements specification divided into *initial partitions* can be assessed. *Component schema relationships* also realise the notification of the affected analyst teams and therefore drive the collaborative resolution of potential conflicts. Future research identifies the impact of design decisions on problem analysis models and the management of requirements for operational systems or subsystems as relevant aspects. The Appendix A presents the UML views and notation used in *component schemas*. Sample statements from the EUROCONTROL Operational Requirements Documents are gathered in Appendix B. These statements formed the basis for the creation of *component schemas*, modelling individual problem partitions.

Appendix C presents examples of conflicts following the classic distinctions of the schema integration community. Finally Appendix D incorporates the complete list of *component schema relationships*. Based on a matrix of the selected UML constructs for *component schemas*, samples from our air traffic management projects illustrate the specific character of the described *component schema relationship*.

2 Related Work

This chapter identifies related research topics in requirements engineering and summarizes relevant approaches for this work. On the basis of identified open problems in the related disciplines, the specific contribution of this work will be presented.

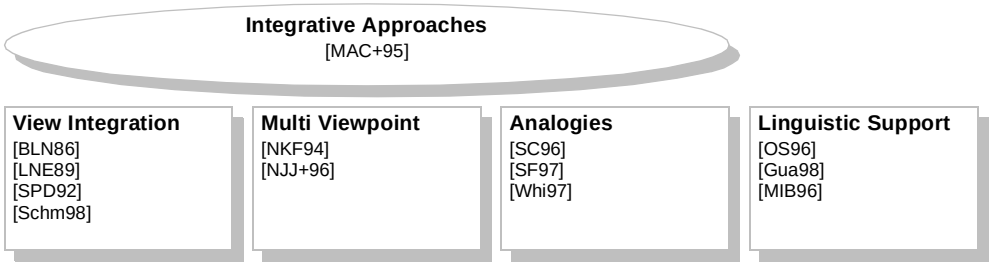
2.1 Research Directions in Requirements Engineering

Software requirements engineering is the science and discipline concerned with eliciting, analysing and representing demands for software systems. The scientific literature documenting the research activities in this field essentially pursues the following five research directions:

- Software & Systems Engineering
- Software Requirements Analysis & Specifications
- Software Requirements Methodologies & Tools
- Requirements & Quality Management
- Software Systems Engineering Process Models

This work essentially explores the use of object-oriented analysis techniques in concurrent requirements engineering, where the involved analyst teams elaborate *initial partitions* of an industry-scale problem statement in isolation. Building an overall problem analysis model on the basis of a set of self-contained *component schemas* gives rise to the consistency of the specification. Consequently the research directions *software requirements methodologies* and *systems engineering* form the most important aspects in this work. Figure 1 depicts individual research directions and references relevant for this works.

Figure 1 Research Directions and Influential References for this Work



Multi-viewpoint specifications and analogies represent the most important topics within the larger frame of requirements engineering. View integration and linguistic support represent contributions from other communities that have been adapted to the purposes of concurrent requirements engineering. View integration for instance provides the starting point for the discussion of conflicts between *component schemas* in this work. The linguistic support reflects the specific character of expert language in requirements specifications.

2.2 Requirements Engineering in Practice

Requirements engineering in practice has always been controversial in the design of industry-scale software systems. Although the need for precise specification of user needs is a widely accepted fact, the different means for their elicitation are in debate. Within the scientific community, requirements engineering dealt with a broad range from social studies [GL93] to formalised mathematical descriptions [DLF93]. Common points within this range are that requirements engineering aims for a gradual structuring of a problem statement from business objectives to a structured problem description, allowing cost-effective design and implementation of the envisioned software system. The first and most important question in this context deals with the concrete nature of user requirements. What is a requirement? How can we understand, describe and assess a demand? Especially taking into account that users often struggle to describe their need, the so-called say-do problem [GL93]. Even if we are perfectly able to elicit user demands, are we able to describe them properly, without leaving out and – equally important – inventing things? Problem descriptions on the other hand should never constrain design and implementation of software systems. Especially industry-scale projects cannot be efficiently managed on the basis of textual requirements alone. Thus the applied modelling techniques should always guarantee a compact and solution-neutral description of the problem statement at hand.

Popular Modelling Techniques

Consensus exists on the value of modelling techniques in requirements engineering and many scientists believe, that quality requirements specifications cannot be accomplished without suitable modelling techniques. Since the 1970 modelling techniques have been widely used in requirements engineering, with varying success. Functional decomposition techniques such as SADT [Ross77] and Structured Analysis [Orr81] became quite popular and are still used in industry-scale projects. These techniques suffer a number of disadvantages, especially the separate consideration of functionality and data is deemed quite problematic. Problem analysis models following these techniques are usually complicated to map to software designs, the frequently quoted impedance mismatch. The expressive power of given modelling techniques still is and always has been an important question in requirements engineering. In this context, we may ask which observable aspects of real-life systems can adequately be represented in a problem description using given modelling techniques.

The Requirements Engineering Process

Process models form another important research direction in this context, focusing on the order of activities for specific techniques. Sommerville [SS97], Davis [Dav93] as well as Thayer and Dorfmann [TD97] are the basic references in this context. Especially remarkable in this context is Sommerville's and Sawyer's adaptation of the *Capability Maturity Model* [PWC95] to requirements engineering [SS97]. The process model in this work is mostly inspired by Sommerville combined with model-based problem analysis, as proposed by Davis [Dav93]. *Component schemas* as our proposed modelling technique with their use case driven analysis style lead to adaptations of the requirements engineering process.

Project Organisation

Project organisation and its specific impact on the resulting requirements specification is hardly documented in the literature. Moody [Moo96] for instance points out the importance of stakeholder involvement in database design with respect to quality and maintainability of the resulting schemas. Forsberg and Mooz describe the project organisation within medium to industry-scale systems [FM96]. However, their findings are mostly related to the larger frame of systems engineering, outside the actual focus of this work. Stressing analyst's autonomy is a central aspect of industry-scale requirements engineering projects. Basing the overall problem description on a set of self-contained object-oriented analysis models representing *initial partitions* is the central idea throughout this work. A novel aspect of this work is the incorporation of the influence of different project organisation models on consistency and completeness of problem descriptions built from a set of *component schemas*.

2.3 Requirements Engineering Methods

Requirements engineering methods according to Kotonya and Sommerville [KoS98] guide the process of eliciting, structuring and formulating requirements. This can be regarded as a systematic way of producing compact and robust descriptions of the problem statement at hand. In this section we essentially concentrate on the different modelling techniques for the representation of requirements and their specific capabilities.

Popular Methods

Functional decomposition techniques developed in the late 1970 gained significant popularity and are frequently used in industry-scale projects. Their entirely functional perspective stresses system functionality but neglects suitable descriptions of data handled in a software system. Therefore Structured Analysis [Orr81] or SADT [Ross77] have often been complemented through the use of Entity-Relationship-Models [Chen76] for data specifications. In the last decade the object paradigm proposed the object metaphor as the sole principle throughout the entire software lifecycle. A number of technologies, including programming languages, database systems, middleware platforms and user interfaces currently follow the object paradigm. Object-oriented analysis techniques, such as OMT [RBP+91], OOram [Ree96] and finally the Unified Modelling Language (UML) [RJB99] make use of the object paradigm for the purpose of analysing given problem statements and providing a stable basis for the design of the software system later on.

Object-Oriented Analysis Styles

Within the variety of object-oriented analysis methods currently on the market, three major analysis styles can be distinguished. Class-based analysis in OMT stresses structural aspects of a problem statement and considers behaviour as a justification for analysis classes. Role-based analysis in OOram [Ree96] concentrates on the required behaviour of objects jointly solving a defined objective. In other words interfaces and object interaction are stressed in this technique. The third style ties the behavioural perspective of role-based analysis to the notion of business processes. Jacobson's *Object-Oriented Software Engineering* [JCJ+94] is the most prominent example for use case driven analysis. Textual descriptions of business processes mark the input to an object-oriented structuring of initial requirements expressed in use cases. The *Unified Software Development Process* finally combines Jacobson's use case driven analysis with the UML notation [JBR99].

Partitioning Industry-Scale Applications

Despite the individual differences of analysis styles, specific support for industry-scale projects is hardly documented in the literature. In this context we concentrate on the initial partitioning of given problem statements. Firesmith [Fir93] suggests a strict top-down approach, partitioning the problem statement on the level of interfaces. Although this approach is advantageous for system integration, stakeholder perspectives are completely left out and a centralised analyst team is the only possible project organisation. This approach also requires sufficient prior knowledge concerning the application domain. Otherwise the resulting *component schemas* do not share a comparable complexity and tend to describe the analysed partitions on an unbalanced level of detail. Multiple perspectives, as introduced by use case driven analysis, lead to another significant problem. Although the analysis complexity has been reduced, the importance of given real-world things for a number of business processes does not become apparent to the analysts. The rather self-contained perspective of use cases may lead to redundancies between *component schemas* and threatens the quality of the overall problem description. Redundancy and its inverse minimality also depend on the actual project organisation and the analyst's horizon, indicating the visibility of model elements to analysts outside the assigned team. Stressing the influence of partitioning and analyst's horizon on the quality of requirements specifications forms another significant contribution of this work.

2.4 Viewpoints

Viewpoint-oriented requirements engineering has been introduced from two completely different directions and technologies. The ViewPoint Framework [NKF94] marks the first important reference, while TELOS and ConceptBase [CB97] mark the other influential related work in this section.

2.4.1 The ViewPoint Framework

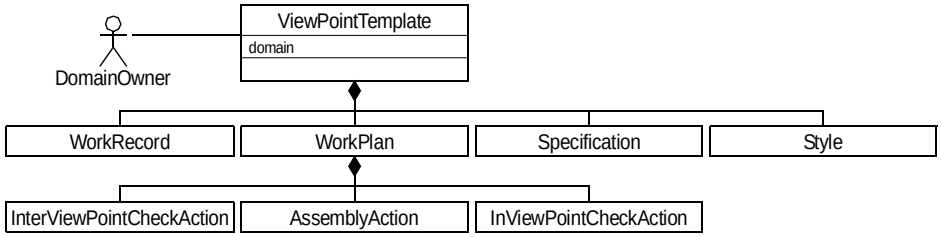
Nuseibeh regards viewpoints as encapsulated, distributed requirements specifications with the following properties:

> "ViewPoints draw together the notion of an actor, knowledge, source, role or agent with the notion of a view or perspective held by the former."
> [NKF94, p. 760]

Viewpoints and Templates

A so-called *viewpoint template* encapsulates the notation of the used modelling technique and a workplan defining legal transactions to a specification. Consequently a *ViewPoint* – in Nuseibeh's terms – is an instance of a viewpoint template and contains an elaborated specification, i.e. a problem analysis model. Viewpoint templates can be decomposed into the following *slots.* A viewpoint contains a *domain*, i.e. a dedicated area of concern, a *specification* describing the domain and a *work record* storing the transactions to the *specification.* The *style slot* incorporates the used notation and the *workplan slot* determines construction and consistency rules for a specification. So-called *in-viewpoint check actions* represent rules to control the consistency of a problem description in the *specification slot*, while *inter-viewpoint check actions* represent rules for *specifications* encapsulated in different ViewPoints. These rules are usually activated by so-called *viewpoint trigger actions*. Figure 2 depicts a ViewPoint template and its slots in the UML notation:

Figure 2 ViewPoint Template Specified in the UML Notation



Industry-scale applications can be divided into *initial partitions* and each of them is elaborated as a distinct ViewPoint. The consistency of the overall specification, encapsulated in several ViewPoints, can be examined through *check actions*. Individual ViewPoints may encapsulate different modelling techniques in their *style slots*. Therefore Nuseibeh distinguishes between *intra-viewpoint check actions*, testing the consistency of ViewPoints using the same style and so-called *inter-viewpoint check actions* for ViewPoints using different *styles*. Changes to the specification, i.e. the invocation of *assembly actions*, does not automatically trigger consistency rules. *Viewpoint trigger actions* therefore realise consistency checks on demand. This approach also decouples conflict detection from conflict resolution and therefore avoids unintentional overriding of requirements. The ViewPoint Framework provides highly relevant distinctions for the distributed nature of concurrent requirements engineering.

Deficiencies of the ViewPoint Framework

Considering multi-viewpoint specifications as distributed encapsulated objects is the central value of the ViewPoint framework. Decoupling detection and resolution of conflicts stresses analyst autonomy, since overriding requirements outside their assigned partition is prohibited. Unfortunately, the ViewPoint Framework remains

vague with respect to a number of details. Especially checking the consistency over distributed ViewPoints on the basis of a so-called *consistency-logic* leaves out important details. This first-order logic formalism does not solve the inherent ambiguity of expert language in the domain at hand. Statements in first-order logics do not reflect the specific meaning of expert terms in the application domain at hand. As a result, linguistic deficiencies concerning the use of *terms* in different ViewPoints lead to conflicts. Another problem is the formulation of *inter-viewpoint check actions* for ViewPoints using different notations. Adequate conflict detection can only be based on rigorous semantics of the involved modelling techniques. The majority of object-oriented modelling techniques, however, is not based on formal semantics. In this case these check actions can only detect the use *terms* in ViewPoints describing the overall problem statement. Elaborate conflict detection cannot be realised since the applied *construct* is not taken into account. This observation also holds for the semantics of the UML, which are defined on the basis of a metamodel.

2.4.2 TELOS and ConceptBase

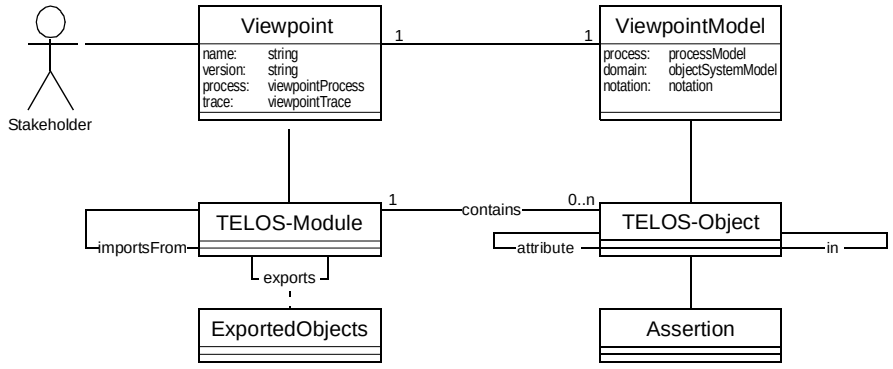
The research group of Prof. Jahrke at RWTH Aachen [NJJ+96] takes a different approach to multi-viewpoint specifications. The basis of their consideration is TELOS, realising all four layers of the ISO information resource dictionary standard [ISO10027]. ConceptBase, as a tool environment, realises TELOS on top of a deductive database [CB97]. TELOS objects can be regarded as four tuples containing the shaded constructs depicted in Figure 3. Specification techniques incorporating several views such as UML [OMG97a] can easily be realised in ConceptBase with a centralised repository. Unlike the ViewPoints framework [NKF94], TELOS specifications cannot be regarded as distributed objects: they remain projections of a centralised repository. This approach guarantees the consistency of the overall specification at any time. Centralised repositories with a rigorous metamodel implementation may lead to unintentional overriding of requirements and consequently do not promote the autonomy of analyst teams. Especially *inter-partition* and *global requirements* may lead to overriding of class specifications relevant in several partitions.

TELOS Modules

Introducing modules to TELOS realises the idea of several independent partitions on top of a centralised repository [MAC+95]. Initial partitions are mapped to TELOS modules, corresponding to modules in programming languages. The autonomy of analysts teams can be guaranteed while import and export interfaces realise the integration of *initial partitions* in a module representing the overall problem statement. Import / export relationships between TELOS modules further avoid unintentional overriding of requirements stemming from different partitions. Integrating two partitions C_1 and C_2 leads to the creation of a new TELOS-Module C_I, which imports the initial partitions to be integrated. Changes and enhancements to imported models are entirely encapsulated in C_I and therefore do not alter the imported partitions at all.

An increased traceability is the result of this approach, since the imported partitions remain completely unchanged and the involved stakeholders do not loose track of their models. Finally, agreed schemas representing given partitions can be exported to other partitions, facilitating the reuse of agreed and validated specifications.

Figure 3 UML Static Structure Diagram Depicting the TELOS Module Concept



Comparing ViewPoints and ConceptBase

Compared to the ViewPoint framework, the TELOS modules have a number of advantages. The meta-metamodel allows the definition of complex modelling techniques such as UML. Modifications to the realised modelling techniques can also be applied at run time, since the ConceptBase tool realises all four IRDS [ISO10027] levels. TELOS assertions may realise *in viewpoint check actions* in the Nuseibeh terminology. *Intra* and *inter-viewpoint check actions* can be regarded as *in-viewpoint check actions* with an additional parameter, the name of given modules / *initial partitions*.

The autonomy of TELOS modules is guaranteed through a hierarchic organisation of the name space. The names of class specifications are unique, since they are always prefixed with the module name. Another interesting detail in TELOS can be regarded both as advantageous and disadvantageous at the same time. All elements of a specification, from instance level to metamodel level in the IRDS terminology, carry a unique and immutable object identity. As a result, name clashes never occur and overlapping analyst perspectives never lead to uncontrolled overriding of requirements. Class specifications may carry the same name and still remain distinct objects. Homonym conflicts in complex specifications are automatically excluded via this approach. On the other hand, means for controlling the use of synonymous in specifications are not described and therefore the minimality of the resulting problem analysis model remains in question. Import and export relationships between TELOS modules cannot indicate conflicts between class specifications specified in distinct modules and the affected analyst teams cannot be notified. Furthermore, all rationale concerning the resolution of conflicts between imported partitions is entirely buried in the TELOS module. At the same time, the traceability of the overall analysis

model cannot be guaranteed, since the background knowledge used for the resolution of conflicts cannot be made explicit. Although TELOS overcomes strict top-down partitioning, import and export relationships alone guarantee neither minimality nor coherence of the resulting overall problem analysis model.

2.5 Linguistic and Ontological Approaches

Requirements specifications are heavily influenced by the inherent ambiguity of natural language, due to the involvement of stakeholders with different cultural backgrounds. Stakeholders are usually experts in a given application domain and each domain is characterized by its own expert language. Ortner and Schienmann regard requirements engineering as a gradual differentiation of natural language towards a so-called *normative expert language* [OS96].

Methodically-Neutral Conceptual Modelling

Based on this distinction, Ortner and Schienmann develop their concept of *methodically-neutral conceptual modelling*. In their perspective, modelling techniques already describe a problem statement in terms of a future solution and therefore do not suit the discussion among affected stakeholders. Given modelling techniques might distort the original requirements statement, since the modelling technique at hand does not offer adequate *constructs*. The lack of adequate data description in functional decomposition techniques is an example of this kind of 'distortion'. Therefore the authors distinguish between a *normative language construction* where the meaning of *terms* in a given application domain has to be clarified and a mapping of these reconstructed sentences to a given modelling technique, representing a *method-specific* representation of the *normative language* [OS96]. This mapping between a *methodically-neutral normative language* towards a *method-specific* one has been described for object-oriented modelling techniques [Sch97].

Although we totally leave out the reconstruction of a normative language in this work – *component schemas* specified in the UML notation form our requirements basis – the work of Ortner and Schienmann [OS96, Ort97, Sch97] provides important input for the definition of *component schema relationships*.

Thesauri and Dictionaries

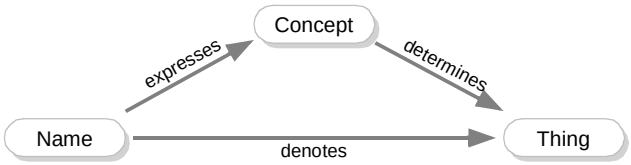
Thesauri and dictionary organisation represent the other central linguistic influence to this work. In this context the organisation of dictionaries and thesauri for a clarification of the meaning of *terms* used in a given *initial partition* is relevant. Alexander Hars [Hars97, HM97] proposes natural-language methods for the integration of data models. Based on a syntactic discrimination of valid *terms* in a given application domain, the similarity of *artefacts* in a set of specifications can be clarified. Syntactic aspects of *terms*, in the sense of Chomsky [Cho65], are combined with semantic categorizations, in order to resolve synonyms in data models.

Names, Concepts and Things

Figure 4 depicts a pragmatic approach for the treatment of the inherent ambiguity of natural language in database schemas, as proposed by Weber [Web82]. Taking the intension/extension dichotomy as a starting point for his consideration, he comes to

the following conclusions. Names in databases express *concepts* and denote *things*. Furthermore, a *concept* determines a real-world thing. When a set of *terms* express the same *concept* and *denote* the same *things*, these *terms* are called synonyms. In contrast to synonyms, *terms* expressing different *concepts* and *denoting* different *things* are called homonymous. In his article, Weber extends these basic distinctions include variables, functions and composite terms. However, Weber's *concepts* are atomic and consequently do not cover class specifications with behavioural and structural properties in *component schemas*.

Figure 4 The Relationship between Name, Concept and Thing



Conceptual Structures

A strategy similar to that used in thesauri can be found in the artificial intelligence community, namely in the work of John Sowa [Sowa84]. Based on three distinct constituents, reasoning mechanisms for complex problems are proposed. The first constituent is based on a dictionary comprising a classification of *terms* according to their syntactical category, such as *proper names*, *count nouns*, *adjectives* and *pronouns*. *Articles* and *prepositions* are ignored, they are stop words in terms of information retrieval. *Proper names* are categorised to *concepts*, i.e. the terms *'Norma'* or *'Jack'* are categorised to the *conceptual type* 'Person'. In other words these *conceptual types* provide the real-world meaning for a set of *proper names*.

Conceptual Type Lattice

The *conceptual types* recorded in the dictionary are organised in a type lattice, where a generalisation/specialisation hierarchy represents the subordination of terms. The general type `T` is the root of an acyclic directed graph. Due to the lattice character of the conceptual type hierarchy, given types may be related to several generalised types, for example the type `Animal` is characterised by the generalised types `MobileEntity` and `Animate`.

Conceptual Graphs

The third and final constituent in Sowa's work is the *conceptual graph*, encoding relations between *terms*, classified in the *conceptual type lattice*. Natural language sentences like *"Spike the dog is sitting in his basket"* can be expressed in conceptual graphs as depicted in Figure 5, p. 20. Formally *conceptual graphs* are bipartite graphs presenting relations between *concepts*. Although the distinction between *concepts* and *conceptual relation* seems arbitrary, *conceptual structures* provide all elements that can be found in an ontology introduced below. Sowa regards *conceptual structures* as a visual representation for first-order predicate logics.

Figure 5 Textual and Graphical Representation of a Conceptual Graph

[DOG:: Spike] → (STATUS) → [SIT] → (LOCATION) → [BASKET]

Ontological Approaches

Ontologies in the philosophical tradition, from Plato to Wittgenstein, deal with the nature and the organisation of reality. In the last decade, these theories have been adopted by computer science, mostly in the field of artificial intelligence. Their principles may also be used for the analysis and design of information systems. Ontological approaches can be roughly distinguished into two different directions. While the first direction focuses on the expressive power of given modelling techniques, the second tries to cope with the inherent ambiguity of natural language in information systems.

Deep Structure of Information Systems

Wand and Weber [WW93, WW95] base their evaluation of modelling techniques on an adaptation of Bunge's [Bun77] ontology, describing the nature of things in our perceived reality. Information systems, similar to natural language in transformational grammars [Cho65], can be distinguished into surface and deep structure phenomena. The surface structure comprises the visible part of information systems for its human users, like the user interface, reports and other system output. The deep structure, however, manifests the *real-world meaning* an information system intends to model. In other words, deep structure phenomena can be regarded as the specific way *how* computational systems encapsulate real-world meaning. This way is determined by the constructs offered by the modelling technique used. Consequently, comparing Bunge's ontology with the constructs of a given modelling technique may be used to measure its expressive power. Despite the fact that we are not assessing the expressive power of UML as such, we make use of the idea of a deep structure in a traditional linguistic sense. *Artefacts* in partial specification may be related to each other via a deep structure, i.e. the similar meaning expressed in terms of synonyms.

Conceptualisation and Ontology

Ontologies in the second research direction are mainly used to cope with the inherent ambiguity of natural language, in order to provide flexible and adaptable access to heterogeneous information sources on the Internet. Although there is still significant disagreement concerning the definition of ontological approaches, we follow the definitions proposed by Guarino [Gua98].

> "[...] an ontology refers to an *engineering artefact*, constituated by a specific *vocabulary* used to describe a certain reality, plus a set of explicit assumptions regarding the *intended meaning* of the vocabulary words. This set of assumptions has usually the form of first-order logical theory, where vocabulary words appear as unary or binary predicate names, respectively called concepts and relations. In the simplest case, an ontology describes a hierarchy of concepts related by subsumption relationships; in more sophisticated cases, suitable axioms are added in order to express other relationships between concepts and to constrain their intended interpretation." [Gua98, p. 2]

Categorisation of Ontologies

The clarification of a domain vocabulary on the basis of a logical theory can be regarded as a significant benefit of ontologies. With an ontology we can, for instance, deduce, that the term ‘board’ denotes a piece of wood in the carpentry domain. Unlike Sowa’s conceptual structures, which are built on top of less expressive first-order predicate logics, ontologies are usually based on description logics [BBM+89, CLN98]. Description logics are able to calculate the subordination of propositions on the basis of constraints and therefore enable automatic concept classification for knowledge bases at runtime. Ontologies according to Guarino can be grouped into the following distinct groups, comprising *top-level*, *domain*, *task* and *application ontologies* [Gua98]. Top-level ontologies describe the vocabulary of fairly general concepts such as *things*, *relations*, *space* and *time*. Domain and task ontologies describe the relevant expert vocabulary of given application domains, where *domain ontologies* focus on structural and *task ontologies* on behavioural aspects. Joint *structural* and *behavioural* vocabulary implemented in a given system forms an *application ontology.*

Comparing Component Schemas with Ontologies

Comparing Guarinos ontology categories to problem analysis models in requirements elicitation leads to interesting analogies. *Component schemas* incorporate the vocabulary of an application ontology, where terms describing structural aspects can be found in static structure diagrams and *terms* related to behaviour in sequence diagrams. In other words, the application ontology is an important outcome of requirements engineering. In concurrent requirements analysis, however, *component schemas* only comprise a *partial application ontology.* Due to the unrestricted use of language, concurrently elaborated *component schemas* potentially incorporate synonyms and homonyms. Consequently, integrating *component schemas* automatically leads to an application ontology. So the availability of a domain ontology directly facilitates the integration of *component schemas*, since terminological anomalies, such as synonymy, polysemy and homonymy can automatically be resolved.

2.6 View and Schema Integration

View and schema integration techniques form another influential research direction for this work. With its roots in the federated database community, the integration of distributed information sources is the main background for this problem. According to Spaccapietra [SPD92] view integration deals with the integration of schemas, described in a single modelling technique, that do not have associated extensions and no coded program using them. By contrast, schema integration deals with the integration of information sources, using different data models, to describe data which is actually stored in a database and support a number of application programs.

Methodologies for Database Schema Integration

Batini, Lenzerini and Navathe [BLN86] provide a first summary of this discipline in their 1986 article. Important results of this work are the integration process and the definition of a taxonomy of conflicts mostly related to the Entity-Relationship model [Chen76]. Semantic relationships, such as composition and generalisation/specialisation, have been left out in their work.

Attribute Equivalence in Schema Integration

Larson, Navathe and Elmasri [LNE89] provide a formalized conflict definition for the observation of Batini, taking their *canonical data model* ERC as a starting point of their considerations. Their definition is founded on a set-theoretic definition of equivalence of values of compared attributes. Similarity of attributes can be expressed in terms of so-called *correspondence assertions*. Although the canonical data model incorporates generalisation/specialisation relationships, an assessment of equivalence has not been discussed for these constructs.

Model-Independent Assertions

Spaccapietra, Parent and Dupont [SPD92] base their considerations on a so-called *Generic Data Model* for the integration of complex and multi-valued attributes. Their distinction of *value* and *reference attributes* solves this special deficiency of the previous approaches. But in this approach generalisation/specialisation has been left out.

Merging Inheritance Hierarchies

In his dissertation thesis, Ingo Schmitt concentrates on merging inheritance hierarchies [Schm98]. A combination of structural as well as schematic aspects can be used to calculate upward-inheritance in his generic integration model.

Our definition of *component schema relationships* takes this related work as an input and extends it to include the selected UML constructs in *component schemas*, not dealt with in the literature. Due to the absence of instances and the inherent ambiguity of names in this work, *component schema relationships* extend the classic definition of conflicts in the schema integration literature

2.7 Analogies

Analogies finally define the metrics for an assessment of the similarity of class specifications. Spanoudakis [SC96] develops his theory on the basis of the TELOS metametamodel and assures an integration to ConceptBase [CB97]. His principle metrics are the *identification*, *classification*, *generalisation* and *attribuation distance*. Whitemire [Whi97] takes a more complex approach, since his formalization is based on category theory. Grouping the similarity of class specifications into *internal*, *external* and *similarity of purpose* influenced this work. Despite its mathematical elegance, the theoretic foundation of Whitmire's approach is far too complex for this work. Both approaches require precise problem specifications, which are usually too detailed and strict for the participating business users. Analogies in the sense of Spanoudakis always require the use of the modelling tool ConceptBase. Research prototypes however, usually do not gain the necessary acceptance from senior management and therefore are hardly used in industry-scale projects.

2.8 Contributions of this Work

Based on a number of different research results, from requirements engineering as well as linguistics and view integration, this work concentrates on requirements in industry-scale problem statements, where sheer size demands efficient use of multiple analyst teams working in parallel. We adopt Nuseibeh's [NKF94] perspective towards specifications as self-contained entities. Requirements specifications documenting industry-scale problem statements are built from a number of so-called *component schemas* modelling *initial partitions* of the problem statement. This gives rise to the question of consistency and completeness of the resulting overall problem description. Unlike Nuseibeh, we use a concrete modelling technique, the Unified Modelling Language (UML), standardised by the Object Management Group in 1997 [OMG97a - OMG97c]. UML can be regarded as the most popular object-oriented analysis technique in industrial practice. Thus analysing the concrete nature of *component schema relationships* is the central concern in this work.

The Requirements Engineering Method

According to the current OMG standard [OMG97a - OMG97c] UML is a *model-based multi-viewpoint specification*, offering a number of different *views* relevant in the entire software development cycle. In order to guarantee solution-neutral character of requirements specifications based on UML views, *component schemas* incorporated a distinct UML subset tailored for requirements engineering. This subset incorporates UML's *static structure*, *interaction* and *use case view*. Within these UML views, all notational *constructs* related to distinct programming languages have been rejected. Tailoring UML for requirements engineering intends to improve the orthogonality of its *constructs* and increase the comprehensibility to business users. *Use case driven analysis*, as the suggested *analysis style* for UML, once again gives rise to the question of consistency and completeness of the overall specification, built from a set of *component schemas*.

Consistency and View Integration

Regarding *component schemas* as self-contained requirements specifications representing *initial partitions* calls for a concrete adaptation of Nuseibeh's viewpoint check actions. Due to the inherent complexity of natural language in use cases and the problematic notion of behavioural correspondence for sequence diagrams, *component schema relationships* have been restricted to static structure diagrams. The view integration community provided the first input for the definition of conflicts [BLN86] and also imparted the idea of *correspondence assertions* [LNE89, SPD92]. Due to the entirely intensional character of requirements specifications, the set-theoretic definition of *correspondence assertions* between attribute values stemming from different database schemas does not hold for this work. Moreover, UML constructs such as packages do not have a direct counterpart on the instance level and therefore induce a set of new conflicts.

Conflicts & Linguistic Support

The entirely intensional character of requirements specifications emphasises the importance of naming conflicts, which has mostly been left-out in the view integration literature. Besides the extended definition of structural conflicts, means for detecting naming conflicts have to be introduced.

In this context, linguistic support plays an important role, on different levels of sophistication. Simple data dictionaries may clarify the meaning of *terms* used in a given application domain. Ontologies [GG95] further incorporate a complete set of linguistic relationships between *terms* characterizing distinct application domains. Linguistic support on the basis of ontologies has the potential to improve the notion of *conflict* and *correspondence* between *component schemas*.

Correspondence

Structural conflicts induced by the used modelling technique and *naming conflicts* as deficiencies in the use of *terms* lead to our definition of *correspondence* and *conflict*. In order to avoid overriding of requirements, *conflict* and *correspondence* are regarded both as *relationships* between *artefacts* specified in different *component schemas*, so-called *component schema relationships*. The completeness of requirements specifications, as a central quality attribute, leads to the definition of *component schema dependency*, essentially used for the indication of *inter-partition requirements*. *Conflict*, *correspondence* and *dependency* are integrated in the UML metamodel, as the basis of our *component schemas*. Thus *component schema relationships* become first-class modelling concepts that can be realised in off-the-shelf CASE-tools realising the entire UML metamodel.

Requirements Engineering

With respect to requirements engineering as a whole, this work also contributes through the examination of organisational aspects for concurrent requirements engineering. Taking Moody's [Moo96] analysis of the impact of stakeholder perspectives on data modelling as our starting point, project organisation and the *horizon* of individual analyst teams leads to different categories of *component schema relationships*. Analysing the relationship between *component schemas* and software requirements specifications according to the IEEE-830 standard [IEEE830] is another significant contribution of this work. Making direct use of *component schemas* for the creation of software requirements specifications has the potential to improve overall project efficiency and increase traceability at the same time. This approach suits stakeholder participation in industry-scale problem statements. As a side effect, stakeholders do not loose track of their problem analysis models, if these models are directly used in software requirements specifications.

3 Requirements Engineering: Stakeholders, Processes & Techniques

Requirements engineering covers all activities involved in discovering, documenting and maintaining a set of requirements for the definition of software systems. Software Requirements Specifications (SRS) are an agreed statement of software requirements for customers, end-users and software developers. Software requirements specifications therefore facilitate the communication between domain experts focused on what the envisioned system shall do and software developers elaborating how the future software system shall realise the specified functionality. Finally an SRS supports the acceptance tests of the realised system, defining the basis for conformance testing [SS97]. This chapter outlines the classic principles of requirements engineering, while the following chapters focus on requirements engineering for industry-scale systems, such as Air Traffic Management (ATM).

The construction of industry-scale systems is always a challenge to the existing knowledge in software engineering. Requirements engineering as the first phase within the software life-cycle can be regarded as the most vulnerable part, since the cost for repairing erroneous requirements will explode later on in the system life-cycle. Fixing the impacts of poor requirements in the implementation phase can be 200 times more expensive compared to fixing them directly in the requirements phase. The long and painful story of software systems that never meet their requirements is not finished yet, and as you read this sentence new chapters are being added to this endless book. What are the reasons for cancelling projects after billions of dollars have been spent? Why are requirements so complex? Why are humans so incapable of formulating their needs?

Since requirements aim to define *what* a system shall do rather than *how* the desired functionality might be realised, we first introduce the main factors effecting industry-scale requirements engineering projects. First different project participants – *stakeholders* – are introduced and their individual responsibilities discussed. Then we introduce a *process model* and describe the individual phases and activities in requirement engineering. Afterward the individual *character of requirements* will be analysed. Finally, *modelling techniques* for structuring and modelling requirements are discussed. All factors discussed in this section provide an empirical basis for the evaluation of UML in industry-scale requirements engineering projects. A detailed analysis of the factors *stakeholders*, *process* and *techniques* under special consideration of their particular impact on requirements engineering projects are provided in the reminder of this chapter.

3.1 A Classification of Involved Stakeholders in Requirements Engineering Projects

Usually a number of different people with particular skills and responsibilities participate in requirements engineering projects, each of them with their own interests, cultural background and personal habits. The term *stakeholder* covers all relevant people in requirements engineering, apart from analysts and project management actually running the project.

> Definition 1 (Stakeholder) – The term *stakeholder* subsumes all people directly or indirectly affected by the envisioned system. ◊

In this section we want to outline the personal background and objectives of relevant *stakeholders*. Since hardly any project corresponds to another, only a very rough categorisation of *stakeholders* can be provided. Project managers, business users and analysts are common participants contributing to industry-scale projects.

Business Users

Business users are the first category of *stakeholders*. Usually they are the central source for detailed needs, that lead to a specification of demanded functionality. Their IT knowledge is usually limited, but they are experts in the particularities of the problem statement at hand. The envisioned system will also directly affect their daily routine. In other words, the future system shall assist the business users in an efficient and cost effective pursuit of their tasks. However, business users have rather personal perspectives with respect to the future system. Usually they only focus on their own role within an organisation and often do not have a detailed overview of the implications with respect to the overall system. Therefore requirements statements of business users tend to be specific with respect to their individual position within the business and tend to be vague with respect to overall implications. In air traffic management, experienced controllers play the role of business users.

Analysts

With respect to their professional background, analysts fall into two categories: experienced business users and experienced IT personnel. The first category usually represents senior staff in air traffic management in a responsible position, who usually have limited IT knowledge, but extensive insight into the overall business implications. This insight usually distinguishes analysts from business users. The second category represents analysts with an IT background and limited business know-how. Modelling experience and the capability of mapping analysis models to competitive designs are the central contribution of this group. Especially modelling experience dramatically facilitates analysis and negotiation of complex requirements.

Information Officer

Information officers, sometimes also called system administrators, are becoming more and more important, due to the fact that systems are hardly defined from scratch any more. Usually new systems integrate the functionality of existing ones and therefore operate closely with an existing information and communication infrastructure. Information officers have a clear IT background and have a high-level knowledge of operational systems. Their experience is crucial for embedding new systems in an existing infrastructure. Domain expertise of these *stakeholders* may contribute to

object-oriented domain analysis, i.e. identifying key abstractions of a dedicated domain, the so-called *Corporate Domain Model* [Moo96]. According to Moody modelling is the key skill of these *stakeholders*.

Project Managers

Project managers are responsible for the overall project pursuit. *In time*, *in budget* and *in quality* are their central concerns. So their specific interests lies in two different sectors: means for controlling project progress and assessing efforts for the development of the future system.

3.2 The Requirements Engineering Process

Kontoya and Sommerville identified the following groups of related activities in requirements engineering, largely resembling Boehm's spiral model [Boe87]. Their process model incorporates the phases *elicitation*, *analysis* & *negotiation*, *documentation* and *validation*:

1 *Requirements elicitation*
The system requirements are discovered through consultation with stakeholders, from system documents, domain knowledge and market studies;

2 *Requirements analysis and negotiation*
The requirements are analysed in detail. An approved negotiation process involving different stakeholders decides which requirements are to be accepted;

3 *Requirements documentation*
The agreed requirements are documented at an appropriate level of detail. In this work, the requirements are documented using UML's static structure and interaction view.

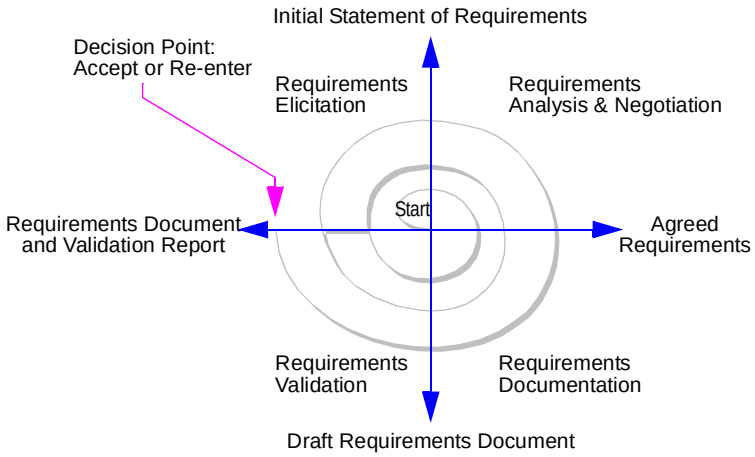
4 *Requirements validation*
A careful check of the requirements for consistency and completeness is the central activity within this phase.

Each of these phases leads to defined intermediate results. The outcome of requirements elicitation is for instance an initial statement of requirements. This draft can be thought of as a long list of requirements that have to be analysed in the following phase. Groups and priorities are to be introduced to the list of requirements in the analysis and negotiation phase. Vagueness, contradiction and omissions finally have to be detected in the negotiation phase.

Due to the generic character of this process model, the individual techniques used in the different phases have to be defined. In this context we have to stress that individual UML views do not necessarily fit into distinct phases of the requirements engi-

neering cycle. Moreover, the use case driven analysis style of *component schemas* combines at least *elicitation* and *documentation* of requirements in a single activity. This discussion is detailed in Section 4.3, pp. 52.

Figure 6 A Spiral Model of Requirements Engineering Processes [KoS98, p. 35]



3.3 Requirements: Character & Techniques

Over the years, computer science has proposed a variety of techniques for the specification of requirements. All of these proposals have individual advantages and shortcomings with respect to the specific character of requirements. Different techniques for eliciting and documenting requirements are introduced briefly in this section. Finally we want to introduce the IEEE standard for requirements, providing an agreed outline for software requirements specifications.

3.3.1 The Character of Requirements

In order to clarify the meaning of the term requirement, we first consult a dictionary. Webster's Collegiate Dictionary gives the following definition:

> "requirement**:** something required:
> **a:** something wanted or needed: NECESSITY
> ⟨production was not sufficient to satisfy military ~⟩
> **b:** something essential to the existence or occurrence of something else**:** CONDITION
> ⟨failed to meet the school's ~ for graduation⟩"
> [MW93, p. 995]

Both meanings, i.e. necessities as well as condition, hold in requirements engineering. Requirements specify user demands *and* define conditions for acceptance of the future product at the same time. While the second meaning, *condition*, in the sense of conformance testing remains outside the scope of this work, we want to concentrate on the first meaning, defining something wanted.

> Definition 2 (Requirement) – The term requirement refers to a user need or a necessary feature, function, or attribute of an envisioned system, that can be sensed from an external position. ◊

Requirements are roughly divided into *functional* and *non-functional requirements*. Functional requirements directly describe *what* a system shall do and appear as operations bound to class specifications in *component schemas*. Non-functional requirements comprise a variety of different aspects, which traditionally fall into temporal and non-temporal system aspects. *Temporal requirements* give values for the system response time and are budgeted to functions that carry out the response [OKK97]. *Non-temporal requirements* cover aspects such as availability and security and are budgeted to components [OKK97]. On the other hand, *design-related non-functional requirements*, or for short *design requirements* often comprise issues such as programming languages, middleware platforms, database systems, operating systems or network protocols.

Structural Requirements

Functional requirements, however, do not comprise information concerning the internal structure of data, relevant in a given application. In the worst case, missing data descriptions may even hinder proper integration of the implemented system. Therefore, we introduce so-called *structural requirements* as a third general requirements category. From a purists point of view, structural requirements may seem design related, since they need a certain level of formal rigor in order to be meaningful. Nevertheless, modelling techniques such as UML [OMG97a] or the Entity-Relationship Model [Chen76] still provide sufficient formal rigor, without actually prescribing a specific programming language. Another important reason for incorporating structural requirements is the ubiquity of information, already present in operational systems. Today there is hardly any situation, where systems may be designed from scratch. So specifying deltas between existing and required information demands precise specification techniques, without going into the details of an existing programming language. The integrated description of data and functionality is the central advantage of object-oriented analysis techniques and is detailed in Section 3.3.4, pp. 33.

Tacit Requirements

Besides the three categories of requirements, there is still a more fundamental question: How far is reality accessible to structured descriptions that may be realised in terms of computational systems. Goguen points out that domain experts are often not able to properly describe their daily routine. He calls this problem *tacit requirements* [GL93]. An important example for this category is user interfaces. Although it is clear what kind of information has to be displayed in order to assist system users,

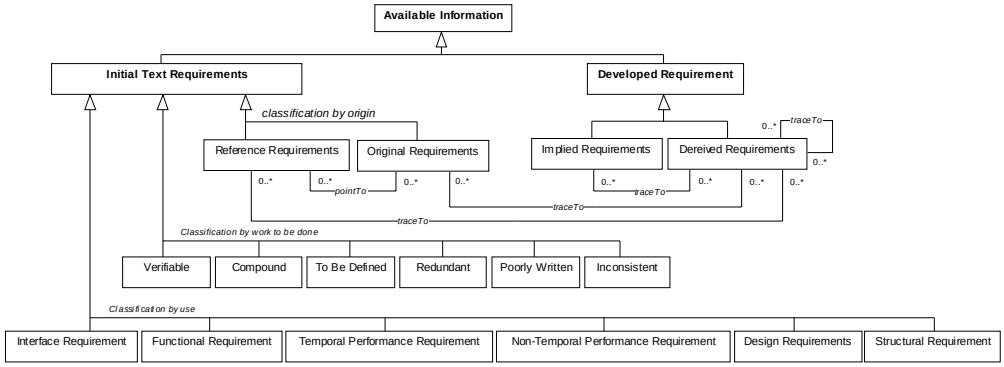
there is considerable space for interpretation, as to how required information can be displayed efficiently and comprehensively. Since user interface requirements are not in the scope of this work, we do not detail this aspect.

Taxonomy of Requirements

Classical requirements engineering often entirely relies on textual descriptions of functionality. Statements such as "*The system shall provide conformance monitoring"* provide examples for this approach. From a broader perspective requirements give rise to the question as to how they might be evaluated. Oliver et al. present the following taxonomy of requirements which we use for an evaluation of UML specifications below (see Figure 7).

Figure 7

A Taxonomy of Textual Requirements according to [OKK97, p. 106]



Initial versus Developed Requirements

Available information in arbitrary form primarily falls into the categories *initial* and *developed requirements*. The user need is either directly specified in terms of *original requirements* or may point to a clearly defined standard, so called *reference requirements* [OKK97]. Whenever original requirements can be further decomposed, they are called *developed requirements*. *Implied requirements* are related to original requirements, but the stakeholders did not directly indicate this need. In other words, the information is implied by other initial requirements [OKK97]. Omission and vagueness in the original statement are common reasons for this kind of requirements. Unlike *original requirements*, *implied requirements* can only be detected when reviewing *original* and *derived requirements*. In other words, this category of requirements represents analysts interpretations of analysts and therefore has to be carefully reviewed with business users and domain experts.

Attributes of Requirements Statements

Examining original requirements leads to a set of attributes, indicating the quality of individual requirements. These quality attributes also indicate potential work to be done, in order to improve the quality of the requirements specification. According to Oliver et al. [OKK97] requirements can be *compound*, *redundant*, *poorly written* and *inconsistent*. The elaboration of requirements can be deferred, so called *to be done* (TBD) or *to be resolved* (TBR) requirements. Requirements carrying these attributes are potential sources of trouble in later stages of the software life-cycle. Figure 7 depicts the attributes of these requirements. *Compound requirements* shall be split into individual derived ones. Redundancies have to be removed and poorly written

(i.e. incomprehensible) requirements have to be clarified. Contradictory information is an indication for inconsistent requirements, which also have to be resolved. Although this categorisation is directly related to textual requirements specifications, we use this taxonomy for an evaluation of *component schema* contents below.

Mapping Requirements to Systems

When none of the attributes in the previous paragraph is applicable to requirements we consider them to be well written. The statement is clear and concise and does not require additional work. Well written requirements, however, may define different aspects of the future system. The principle distinction in this context is the distinction of soft- and hardware components. Software related requirements, however, serve different purposes in the future system, i.e. they can be grouped into categories like *interface*, *functional*, *temporal* and *design* requirements [OKK97].

3.3.2 Requirements, Models and Modelling Techniques

User requirements are usually stated as plain text-statements in the form *"The system shall provide medium-term conflict detection"*. Especially industry-scale systems, comprising thousands of textual requirements statements, complicate a thorough understanding of its particularities. This understanding is quite difficult to reach on a plain-text basis alone. Modelling techniques with powerful abstraction mechanisms provide efficient access to complex plain-text requirements. In theory they should be comprehensive to all kinds of *stakeholders*, from users to software designers. Several different modelling techniques have been used in scientific as well as industrial practice, like SADT [Ross77], the Entity-Relationship Model [Chen76] and various object-oriented techniques [RBP+91, Ree96, RJB99]. Before going into detail with respect to the notations of given modelling techniques, we start with a rather primitive minimal model, providing a baseline for a discussion of various requirements engineering techniques.

> Definition 3 (Abstract Model) – An abstract model represents a piece of reality in terms of *elements*, *relationships* and a defined *system boundary*. *Elements* represent *real-world things* and *relationships* describe relations among *real-world things*. ◊

So the model divides the perceived reality into the *system* marking the primary matter of interest and the *system environment*. The *system environment* is formed by *elements* related to *elements* within the area of discourse but not directly part of the system. For the time being this distinction provides a sufficient model-theoretic basis.

In order to avoid lengthy discussions concerning the character of UML and its indivdual views we simply call it a *modelling technique*. The term *modelling technique* does not imply a formal definition of semantics. The term *method*, another frequently used term in the requirements engineering community, also does not correctly characterise the UML. A *method* implies a defined process, which has explicitly been excluded from the UML standardisation.

The UML standard is based on a 'semantics' document including an *abstract-syntax*, *well-formedness rules* and an extensive textual description. But there is significant dispute concerning the adequacy of this approach for definition of semantics. We believe that this definition can be regarded as *intuitive semantics*.

Definition 4 (Modelling Technique) – A *modelling technique* refers to a graphical notation with intuitive semantics. ◊

Intuitive semantics allow a correct use of the UML notation for stakeholders and also assure a common understanding of the described reality in a UML model. When we refer to individual syntactical elements of the UML we use the term *construct*.

Definition 5 (Construct) – A *construct* refers to a syntactical element of a modelling technique describing a selected *aspect* of a *concept* in the model world. ◊

We base our discussion in the remainder of this work on an analysis of the UML metamodel, i.e. we directly adopt the *intuitive semantics* of the UML standard. *Construct* always refers to a metaclass in the UML metamodel as depicted in Chapter 5.

3.3.3 A Taxonomy of Systems

While the beginning of this section discussed the particular character of requirements, this section is intended to characterize distinct differences of applications. Carving out particularities of individual applications or entire application domains outlines the applicability of relevant problem analysis techniques for the problem to be solved. Davis proposes the following five orthogonal axes for the categorisation of problems to be solved [Dav93]:

1 Difficulty of problem

2 Relationship between data and processing in time

3 Number of simultaneous tasks to be performed

4 Relative difficulty of *data*, *control* and *algorithmic* aspects of problem

5 Determinism versus non-determinism of the results

The relative difficulty of common applications incorporate only one or two of the listed attributes. *Data* and *control*, for instance, are the main characteristics of inventory control systems. Air traffic management, however, is one of the few application domains incorporating all three attributes. But its main characteristic still remains *control* and *algorithms*. The relative difficulty of the problem at hand has a strong impact on problem analysis notations, since they should suit the actual attribute. Missing out one of the attributes might cause omissions, inconsistencies or ambiguities in the resulting requirements specification.

3.3.4 Object-Oriented Problem Analysis Styles

In this subsection we want to briefly introduce a set of popular requirements engineering techniques. We distinguish between *functional decomposition* and *object-oriented* techniques. SADT [Ross77] is a prominent examples of functional decomposition techniques widely used in air traffic management. However, frequent criticism with respect to this technique stresses the problematic mapping of functional specifications to software designs later on. As a result, analysis and design specifications have no direct relationship and consequently traceability is hindered. The seamless transitions from requirements analysis to software design makes object-oriented techniques quite attractive for industry-scale projects, although this still has to be considered as an unproven claim.

Object-Oriented Requirements Analysis

Although there is still little experience documented in the literature with respect to industry-scale applications, this technique is rapidly gaining popularity in a variety of application domains. According to Berard [Ber93] and Firesmith [Fir93] object-oriented requirements analysis falls into the categories *application* and *domain analysis*. Domain analysis seeks for key abstractions in entire application domains, while application analysis concentrates on requirements engineering for a dedicated application within a larger domain. As a result, application development can be regarded as a life-cycle activity, which is not the case with respect to domain development.

Character of Requirements Engineering in the ATM Domain

Although a clear distinction between domain and application analysis cannot be drawn in our study case, its character has to be regarded as application development. Although an entire application domain is covered in this project, the objective is a definition of requirements for the implementation of a concrete air traffic management infrastructure.

Object-Oriented Analysis Styles

Today object-oriented analysis techniques may be divided into three different styles: *class-based*, *role-based* and *use case driven analysis*. OMT [RBP+91], OOram [Ree96] and Jacobson's Object-Oriented Software Engineering [JCJ+94] are prominent examples for these three styles. The following subsections briefly introduce the most important characteristics of these styles, concluding with a short evaluation of their individual advantages and trade-of. Since UML itself does not propose a defined analysis process, it may be used as a notation for class-based as well as use case driven analysis.

3.3.4.1 OMT: Class-Based Object-Oriented Analysis

OMT is the prototype for the *class-based analysis style* [RBP+91] and mainly focuses on structural aspects of the problem at hand. An OMT model according to Rumbaugh comprises the following three models: the *object*, *dynamic* and *functional model*. The OMT object model focuses on the invariant structure and corresponds to the static structure diagram in UML. The dynamic model comprises object interaction as well as state changes, corresponding to *sequence diagrams* and *statecharts* in

UML. Finally the functional model delimits the overall system functionality and is based on DeMarco's *data flow diagrams* [DeM79]. This subsection will primarily introduces OMT's process model and the main analysis outcome, the object model.

Problem Analysis in OMT

Various information sources, like company procedures, business plans, operational procedures, forms, screens and reports, have to be scanned for nouns. In the following step all nouns related to the problem at hand will be grouped to *tentative objects*, such as `Trajectory`, `Aircraft` or `Flight Plan`. Finally all spurious objects, i.e. objects which do not have a direct relationship to the application domain, have to be erased from the list of *tentative objects*. The resulting set of candidate objects are still unrelated to each other with respect to simple, as well as semantic relationships. Rumbaugh summarises the process as follows [RBP+91, p. 152]:

1 Identify objects and classes

2 Prepare a data dictionary

3 Identify associations (including aggregation) between objects

4 Identify attributes of objects and links

5 Organise and simplify object classes using inheritance

6 Verify that access paths exist for likely queries

7 Iterate and refine the model

8 Group classes into modules[1]

The creation of an OMT object model therefore heavily relies on a systematic linguistic treatment of the problem statement. As a result, structural properties dominate over behavioural ones. Rumbaugh suggests *"add operations to classes later as a by-product of constructing the dynamic model"* [RBP+91, p. 152]. This approach leads to three major conclusions:

OMT Summary

1 Behavioural properties are not reflected in the definition of the initial class hierarchy in the OMT object model. OMT neglects behavioural system properties in terms of object interaction.

2 OMT is not applicable for industry-scale projects, since classes form the initial partitions of the problem statement

3 OMT does not support stakeholder perspectives.

1 The equivalent UML construct is called *package*

3.3.4.2 OOram: Role-Based Object-Oriented Analysis

OOram [Ree96], as the main reference for *role-based analysis*, draws the analyst's attention to the interaction between real-world things. Actors accomplish a defined business objective, comparable to contracts in the ODP standard [ODP P2]. This approach leads to the following fundamental differences with respect to OMT:

1 Role-model collaborations mark initial partitions of the problem statement

2 Behavioural aspects shape the resulting analysis model.

OOram's Role Model Abstraction

A role model collaboration abstracts from the physical representation of classes in programming language terms. The collaboration can be seen as related objects serving a defined business objective. A collaboration in the example is depicted through an arc linking the interacting objects via so-called *ports* (see Fig. 8, p. 35). Ports can be regarded as an abstraction of interfaces, comprising the set of operations used within the collaboration. For the complete description of objects in roles, Reenskaug proposes ten different OOram views that even include a primitive model for the decomposition of algorithms. Within these OOram views, the *Scenario View* corresponds to UML sequence diagrams and the State Diagram View corresponds to Harrels statecharts [Har87].

Figure 8 A Sample Role Model Collaboration [Ree96]

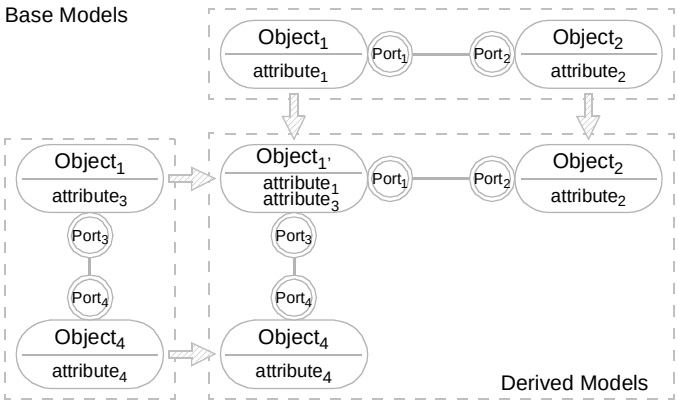
Synthesis: Superimposing Roles

The fine-grained character of initial partitions in OOram reduces the analysis complexity. An average role model collaboration should comprise between five and nine objects according to author's recommendation. Therefore OOram objects may participate in multiple roles, which might be scattered over a set of different roles models. The interesting point behind this idea is the way Reenskaug relates objects in participating in different role model collaborations. *Synthesis*, in the OOram terminology, superimposes a set of base models leading to a derived model. For the integration of structural systems aspects, interfaces (*ports*) and attributes of objects in different roles are simply superimposed. The integration of behavioural specifications is divided into two different categories: *safe* and *unsafe synthesis*. *Safe synthesis* preserves the integrity of the behaviour specified in the base models, while in *unsafe synthesis* the course of *messages* of two base roles may interfere. Figure 9, p. 36 depicts the *synthesis* of two simple base models.

Performing Role Model Synthesis

In the first step, objects from different base models carrying the same name are superimposed in the derived model. $\texttt{Object}_1$' in the example then carries the union of attributes from the base models, i.e. $\texttt{attribute}_1$ and $\texttt{attribute}_3$ in this case. The *ports* of base models are also superimposed in the derived model. As a result, $\texttt{Object}_1\texttt{'}$ also carries the union of operations defined on $\texttt{port}_1$ and $\texttt{port}_3$.

Figure 9 Role Model Synthesis



Classes, Types and Roles

According to Reenskaug, *synthesis* cannot be regarded as formal integration of behavioural specifications [Ree96, p. 85]. But even *synthesis* of the structural part of base models once again gives rise to a variety of potential conflicts, as discussed in Chapter 6. Although *synthesis* can be regarded as an informal process for the integration of base models, it still is a very elegant method for the specification of object models. Decoupling roles from classes in the programming language sense leads to the desired reduction of analysis complexity and role-model synthesis generates the required input for design activities.

The role model abstraction describes collaborating objects in terms of an OOram role. Roles specify types, which can be regarded as partial class specifications in object-oriented programming languages. The OOram type consists of all attributes and operations specified in a given role model collaboration. Synthesis finally integrates all types into a single class specification (see Figure 10).

Figure 10 Relationship Between Objects, Roles, Types and Classes in OOram [Ree96, p. 15]

OOram Summary

1 Role-models provide initial fine-grained partitions

2 Roles shape structural and behavioural properties of the resulting class specification

3 Role-model synthesis integrates a set of initial partitions – role model collaborations – towards an overall problem description modelling the entire problem statement

3.3.4.3 Use Case Driven Analysis

Use case driven analysis as proposed by Jacobson et al. [JCJ+94] combines the idea of business processes with object modelling. Plain text descriptions of business processes, the *use cases*, form the initial partitions of the overall problem statement and also define a first set of requirements. These requirements describe the interaction of stakeholders, the so-called *actors* [JCJ+94], with the envisioned system. Use cases therefore primarily stress behavioural system properties.

Categorisation of Analysis Classes

But unlike OOram and OMT, neither static associations nor operations are specified in the analysis model. These activities are completely deferred to later stages of the development cycle, the so-called *construction*. The entire behaviour specified in a given use case – the control flow in terms of imperative programming languages – is encapsulated in a distinct object, the so-called *control object*. As a result, the analysis objects are grouped into three different categories, *entity, interface* and *control objects*. Entity objects are '*model information that the system will handle over a longer period of time'* [JCJ+94, p. 184]. In other words, entity objects are persistent objects, 'surviving' the execution of a given use case. The actors in given use cases are subsequently replaced by *interface objects*, representing the future user interface. All objects derived from a given use case are related to each other through an acquaintance relationship. Generally speaking, Jacobson treats the analysis model as a rough outline of use cases. The computational interpretation of the model has to be defined in the design phase.

OOSE Summary

1. Business processes define initial partitions of the problem statement
2. Use cases provide a textual description of activities performed by actors in a given business process
3. Construction defines a computational interpretation of the requirements shaped in use cases
4. A coherent description of classes involved in several use cases is not given.

3.3.4.4 Comparing Object-Oriented Analysis Styles

Having introduced the particularities of different object-oriented analysis styles, the following issues provide a baseline for the discussion of their individual advantages in concurrent requirements engineering:

- Behavioural versus structural perspective
- Initial partitions

Behavioural versus Structural Perspective

Emphasising structure over behaviour and vice versa leads to significant differences in the resulting analysis model. OMT suits for instance the design of information systems, where the persistent management of data is the most prominent characteristic.

Environmental information systems or inventory systems are examples of this kind of applications. The main responsibilities in terms of services are related to static integrity and induced dynamic integrity of analysis objects. C^3I applications, like air traffic management, are characterised by complex behaviour, related to the system's internal state and a complex reactive environment. Thus, OMT is not likely to suit these particular requirements, since behavioural aspects fall short. Although the key abstractions of problem statements become visible, there is no adequate representation of control flow. Use cases are likely to compensate this deficiency. They focus on business processes, the central source of requirements changes. Encapsulating control flow in dedicated control objects further improves the maintenance of the future system, as entity objects are not bloated with behaviour that does not necessarily belongs to them. The idea of objects in roles further stresses the importance of behaviour. OOram depicts the required behaviour as a set of interacting objects under a defined objective. Interfaces, so-called *ports*, group all necessary operations. Decoupling objects in roles from classes in the programming language sense also suits the central objective of requirements engineering, describing what a future system should do.

Initial Partitions

Although the object metaphor has to be regarded as a powerful abstraction for real-world things, industry-scale problem statements require additional mechanisms for a reduction of analysis complexity. Firesmith, for instance, proposes UML packages as *initial partitions* [Fir93]. A hierarchic decomposition of the problem statement can also be achieved. But there are still severe limitations to this approach. Self-contained *initial partitions*, as a central objective in concurrent requirements engineering, demand loose couplings between the involved partitions. So background knowledge is necessary for the division of the overall problem statement into *initial partitions*. Classes and packages only realise a single analysis perspective with respect to the overall problem statement, that does not incorporate stakeholder perspectives.

OOram and use cases provide multiple perspectives on the problem statement. Business processes are the initial decomposition mechanism in use case driven analysis and therefore automatically incorporate the perspectives of the involved stakeholders. In comparison to use cases, OOram role model collaborations provide fine-grained descriptions of objects realising a defined objective. Due to the arbitrary character of this objective, role-models are not automatically related to individual stakeholder perspectives. Use case driven analysis and role-based object-oriented analysis do not provide an overall problem analysis model. Consequently, class specifications derived from a set of use cases have to be integrated, in order to document the entire problem statement. Although role-model synthesis in OOram claims to solve the integration of individual collaborations towards an overall problem description, its fine-grained character does not suit industry-scale projects.

3.3.5 Requirements Engineering Standards

Requirements engineering standards are gaining more and more acceptance in the industrial practice and are therefore also introduced in this work. The IEEE recommendations and the MIL-Std-490 are examples of such standards. In this work we concentrate on the 1993 revision of IEEE Std-830 [IEEE830]. This standard neither prescribes a defined process nor proposes a particular requirements technique. Its objective is clarifying the terminology in requirements engineering and providing a defined structure for the individual project results in terms of a software requirements specification (SRS). Defining the characteristics of a good SRS lays out a basis for the evaluation of software requirements. One of the central advantages of this approach is its flexibility with respect to the applied requirements engineering techniques. The standard proposes a distinct organisation for the techniques listed in Table 1:

Table 1 Organisation of Software Requirements Specifications According to the IEEE Std.830-1993

Organisation	**Description**
System Mode	Systems may behave differently depending on their major operation modes
User Class	Requirements are organised according to the perspective of involved business users
Object	Requirements are organised in terms of classes, attributes and services
Feature	Requirements are organised in terms of externally visible features.
Stimulus	Requirements are organised in terms of external stimuli a system has to react to
Response	Requirements are organised according to outputs of the future system.
Functional Hierarchy	System functionality is organised into a hierarchy of functions ordered by common inputs, common outputs or common internal access

Software requirements specification may also be organised using a combination of these styles. Use cases [JCJ+94], for instance, can be regarded as a combination of the organisational styles *feature* and *object*. Organising SRS's according to functional decomposition techniques is still quite popular. Most operational requirements in EUROCONTROL's European Air Traffic Control Harmonisation and Integration Programme (EATCHIP) are organised in terms of this technique.

Although the IEEE standard only focuses on the organisation of an SRS as the agreed result of requirements engineering projects, it still provides hints for assessment of individual techniques. Using particular requirements engineering techniques always gives rise to the question as to which aspects of a problem statement are covered and which aspects of the problem statement remain at least unclear. This question is related to the efficiency of given requirements engineering techniques. Problem analysis models shall directly be used for the creation of an SRS. Directly using problem analysis models in a SRS provides a compact outline to complex requirements. On the other hand, the analysts have to know about the shortcomings of a given formalism, i.e. which requirements are still missing.

Mixing plain-text with modelling techniques also offers varied access to the same information and therefore suits the different cultural backgrounds of project participants. Chapter 10 finally evaluates aspects covered by our proposed requirements engineering technique – *component schemas* – and proposes an SRS organisation for this specific technique.

Figure 11 The Proposed IEEE Outline for Object-Oriented Software Requirements Specifications

Table of Contents

3.4 Discussion: Stakeholders and Modelling Techniques

Requirements engineering techniques should efficiently support business users and analysts in the following objectives.

- Gaining insight to the problem statement
- Covering all kinds of requirements, i.e. structural, functional, temporal and design requirements
- Defining *initial partitions* related to the perspective of participating stakeholders
- Provide coarse-grain abstractions for initial partitions
- Enable the participation of *all* relevant stakeholders

Lack of stakeholder participation and fine-grained analysis results are the central disadvantages of OMT [RBP+91] and OOram [Ree96]. OMT also suffers from insufficient input for modelling behavioural problem aspects. Use case driven requirements analysis using the UML notation combined with the process introduced in

Section 3.3.4.3 provides a solid basis for industry-scale projects. Stakeholder participation is automatically guaranteed and skills from various disciplines may effectively be combined. Writing the initial use cases is the central contribution of domain experts. Extensive IT knowledge and experience in modelling techniques are not required. Analysts with an IT background complement business users in the provision of modelling skills, transforming plain-text use cases into object models. Models and use cases have the potential to clarify vagueness and contradiction in collaboration with business users. So business users do not necessarily have to create the models themselves, which presupposes extensive modelling experience, they only have to be 'literate' in a given notation. Comprehension to business users can be further improved by extensive plain-text descriptions of the individual models, leading to complementary information documenting the problem at hand. As a result, business user participation can be incorporated into projects on a broad range, which dramatically improves the acceptance of the future system. Even the familiarization of IT personnel without experience in a given domain may be accelerated by quality problem analysis models. However, the individual expressive power of a requirements specification based on this technique has to be evaluated. The principal criteria for this task are the coverage of requirements, e.g. structural, temporal, functional and design requirements and especially how these requirements appear in a specification based on this technique.

Chapter Summary

This chapter introduced relevant project participants and their specific contribution to concrete projects. Various kinds of requirements have been distinguished, in order to provide a basis for the evaluation of *component schemas*, as our preferred requirements engineering technique. Of the different object-oriented analysis styles, we favour use case driven analysis. Incorporating stakeholder perspectives and focusing on business processes are the central advantages of this analysis style with respect to industry-scale projects. Focusing on business processes also incorporates the most likely source for changing requirements. Finally, the IEEE-830 standard provided characteristics for quality software requirements specifications and proposed a standardised organisation for software requirements specifications.

4 Concurrent Requirements Engineering

The central challenge in requirements engineering for industry-scale systems is the size of the problem statement itself. Usually requirements engineering projects take several man years. In order to accomplish this task in a reasonable time scale, the overall problem statement has to be divided into smaller initial partitions, each of them elaborated by an assigned group of analysts. This demand gives rise to a number of interesting questions concerning the character of partitions and the overall project organisation.

This chapter mainly discusses organisational issues emerging in requirements engineering for industry-scale software systems. Several analyst teams concurrently elaborate specialised topics within the problem domain. But working concurrently always gives rise to a set of severe problems with respect to organisational, technical and procedural aspects. Industry-scale problems are simply too complex to be entirely understood by individual experts or a single analyst team. As a result, communication and coordination between individual teams become important factors for the overall project success. Tools and their notations alone do not improve the project performance. Organisational issues combined with a customised process have the potential to improve the project performance and gain higher maturity levels in requirements engineering in the sense of the *Capability Maturity Model* [PWC95].

Organisation, process and tools form a *methodology* [OKK97]. This chapter provides an outline for the methodology applied in a series of requirements engineering projects in the air traffic management domain. Here we discuss the particularities of concurrent requirements analysis in industry-scale projects. The first section of this chapter discusses organisational aspects, in terms of participating stakeholders and their specific responsibility within the overall project. Then partitions, the character of requirements within partitions and the relationship between partitions and repositories have to be analysed. Finally, we briefly introduce the concurrent requirements analysis process in the air traffic management domain.

4.1 Organisation

Project organisation can generally be divided into two different categories: the organisation of the business domain and the organisation of the requirements engineering project itself. Although the first category is always subject to careful considerations, this work concentrates on the interaction of involved stakeholders conducting the actual project. So the following subsections proposes different project organisations for industry-scale projects. In this context we identify additional stakeholders involved in industry-scale projects: *analysts*, *business users* and *project managers*.

We want to describe their individual responsibilities and especially the interaction among different stakeholders within a given project, paying special attention to working in parallel.

Partition Owners

In concurrent requirements engineering the problem statement is divided into a set of *initial partitions* and each *initial partition* is assigned to a distinct analyst team. An analyst team is directed by a dedicated *stakeholder*, the *partition leader*. The term *partition owner* then, stresses the organisational responsibility of the project participants.

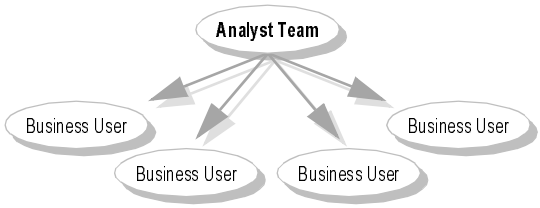
> Definition 6 (Partition Owner) – The analyst team assigned to a given *initial partition* is called *partition owner.* ◊

Analysts that are not *partition owners* for a given *initial partition* are called *external analysts*. In this context we also use the term *owned partition* when we refer to the partition an analyst has been assigned to. Consequently the *partition leader* is responsible for analysing an *initial partition* in time and in budget, and also represents the analyst team externally, i.e. with respect to senior management and other involved *partition leaders*. The *partition leader* completes the list of relevant stakeholders, described in Section 3.1.

4.1.1 Centralised Analyst Teams

In this work we primarily distinguish between the analyst team conducting the actual requirements analysis and the business users providing the necessary input. Business users fall into two groups: domain experts which are senior personnel with extensive domain knowledge and the specialists using the future system. The analyst team is composed of analysts with extensive requirements engineering experience and domain experts providing the necessary knowledge of the actual domain.

Figure 12 Centralised Analyst Team



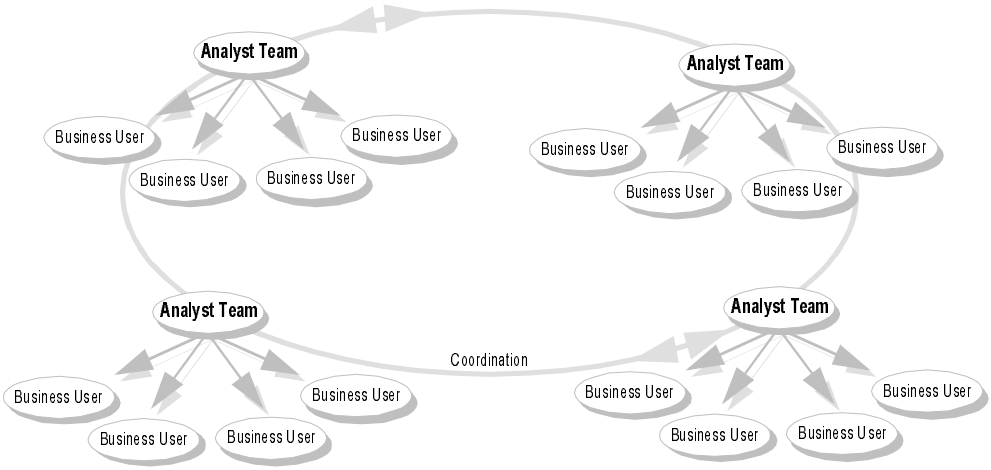
In centralised organisations, one team of analysts conducts a requirements engineering project. Depending on the size of the actual project, analyst teams consist of two to seven members. Due to this small number of participants, the communication between them does not impose special considerations. In this kind of project organisation, analysts design questionnaires and conduct interviews with business users.

Communication between individual stakeholders in the analyst team does not require further consideration, since direct contact among the analysts as well as the business users is always assured.

4.1.2 Autonomous Analyst Teams

The second method of organisation consists of teams of analysts concurrently elaborating their *initial partitions*. This form of organisation does not foresee a centralised management.

Figure 13 Autonomous Analyst Team



Advantages

Autonomous analyst teams, as we call this form of organisation, speed up requirements analysis, due to the concurrent analysis of *initial partitions*. Analyst teams may concentrate on their individual portion of the overall problem statement, resulting in a reduced analysis complexity. Creativity and autonomy, especially in the beginning of requirements elicitation, where the real particularities of a problem statement usually remain unrevealed, guarantee smooth development of key abstractions forming the particular problem partition and allow a thorough understanding of the *initial partition.*

Disadvantages

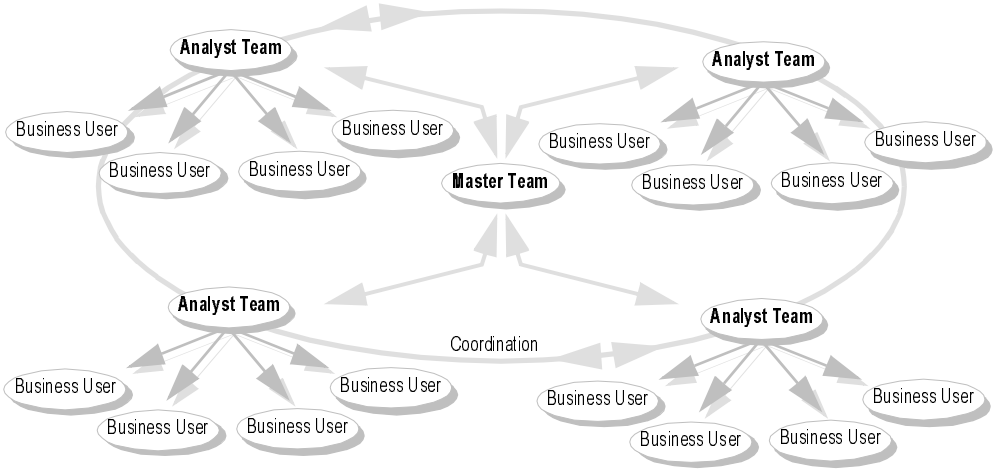
However, the absence of centralised project management, means no communication process among the involved analyst teams is prescribed. In the extreme case, this approach leads to creative and efficient but self-contained solutions. Due to a potential lack of communication and an undefined coordination among all participating analyst teams, the individual results remain incompatible with each other. Partial aspects of the problem have been solved, but the overall problem statement remains vague, since *global* and *inter-partition requirements* have not been considered.

Such a specification does not meet necessary quality standards, as formulated in the IEEE-830 standard [IEEE830]. A detailed discussion concerning the impact of this form of organisation on concurrently elaborated *component schemas* can be found in Section 8.2, pp. 167.

4.1.3 Master Multiple Team

The master multiple from of organisation can be regarded as an autonomous analyst team with an independent master team responsible for the mediation of the entire process. The project manager of the entire requirements project also directs the master team. In other words, master team and project management may be used synonymously. The responsibility of the master team incorporates two important tasks, controlling the project progress from a budgetary and disciplinary point of view as well as reviewing the individual results of the participating analyst teams. Depending on the actual project size, the master team may also be responsible for a defined partition. In this case the master team incorporates the roles of an autonomous analyst team and project management at the same time.

Figure 14 Master Multiple Analyst Team



Educational Background

In the requirements elicitation and negotiation phase, the master team provides consultancy to the involved analyst teams, without actually doing the modelling. Therefore, a master team mainly incorporates experienced IT personnel with an extensive modelling background and senior domain experts. The combination of domain and IT knowledge, with a strong focus on modelling, is the most important skill required by a master team.

Master Team Integrates the Partitions for Validation Purposes

At the beginning of requirements validation, the master team has to integrate the results of all participating autonomous analyst teams. This integrated model provides a coherent set of requirements as an agreed baseline for the production of a software requirements specification, organised according to the IEEE-830 standard [IEEE830].

Advantages

A master multiple organisation has several significant advantages. Individual analyst teams may elaborate their *initial partition* in isolation and therefore improve especially the earlier phases of requirements elicitation. The key abstractions forming a distinct *initial partition* may be developed, without external interference. Creativity and innovation are assured and the provision of crucial modelling know-how is guaranteed by the master team. Issues affecting a set of partitions can be assessed and propagated by the master team. This responsibility directly increases the awareness for issues affecting the entire project and will decrease emerging problems in requirements validation.

Freeing a master team of budgetary and disciplinary responsibility also provides an excellent basis for incorporating external consultants, thus focusing on the core competence of an organisation. On the other hand, external consultants especially those with an extensive IT background, might run into considerable acceptance problems with respect to domain experts and business users. As a result, a master-multiple organisation suits outsourcing, as long as sufficient domain expertise can be incorporated to this team.

Disadvantages

Especially when project management and modelling competence are not concentrated in the master team, its position is quite critical. Due to the lack of disciplinary responsibility, violations of defined conventions cannot be assured. The advisory character of the master team cannot assure that issues effecting the entire project will really be covered in the participating analysts teams. Individual analyst teams might insist on their results without taking other requirements into account. So the efficiency of a master-multiple organisation is in question, when disciplinary and budgetary responsibility are not delegated to this group.

4.1.4 Master-Multiple Organisation in ATM Projects

A master-multiple organisation model was chosen in our reference project in the context of the European Air Traffic Control Harmonisation and Integration Programme (EATCHIP). In this concrete project [WCW97] the master team did not include project management and therefore only provided modelling knowledge and reviewed the results of participating autonomous analyst teams. Moderating the communication process for issues affecting the entire project formed the other central responsibility of this team. Moreover, this group can be regarded as an internal team of consultants assisting the individual analyst teams, mostly staffed with domain experts, in their work. The master team had the following responsibilities:

- Coaching analyst teams in the selected modelling technique
- Mediation for issues affecting the entire project
- Maintaining a balanced level of abstraction over all partitions
- Version management of the intermediate results of individual analyst teams.

Especially decoupling disciplinary and budgetary responsibility proved a serious drawback in this project. Missing *global* and *inter-partition requirements* were detected by the master team. Proposals for the resolution of this incoherence were not applied by the affected analyst teams. So the integration of the partitions proved problematic, since no agreement could be found concerning the concrete structure or the real-world semantics for several key abstractions in the application domain.

4.1.5 Additional Stakeholders in Concurrent Requirements Engineering

Besides the stakeholders introduced in Section 3.1, concurrent requirement engineering introduces new responsibilities, which have to be assigned to involved project participants. Ownership and disciplinary responsibility for an analyst team elaborating an *initial partition* is assigned to the *partition leader*. This stakeholder represents an analyst team responsible for an *initial partition*. The partition leader is also the contact for the coordination of *inter-partition requirements* among involved partitions. Table 2 lists the completed set of stakeholders in concurrent requirements engineering.

Table 2 A Classification of Stakeholders in Concurrent Requirements Engineering

Stakeholder	**Role and Responsibility**
Business Users / Controller	Business users provide the input to requirements elicitation and are also responsible for a review of the resulting problem analysis models
Analyst	The analyst represents user requirements in terms of given problem analysis models and conducts reviews with the business users later on.
Project Manager	The project manager conducts the entire project. On the basis of the Corporate Domain Model representing past requirements specifications, s/he assigns participating Analysts and Business Users to the partitions. Finally the Project Manager uses the requirements specification as one of the inputs for estimating development costs.
Information Officers	The information officer reviews the completed requirements specification for consistency with the Corporate Domain Model to ensure that separately developed systems interoperate. New domain objects are then incorporated to the Corporate Domain Model for reuse in future projects.
Partition Leader	The partition leader is responsible for a group of analysts, elaborating a defined partition within the overall problem statement.

4.2 Partitions, the Scope of Requirements and the Analyst's Horizon

Sheer size demands for a division of the overall problem statement of industry-scale applications into manageable chunks, so-called *initial partitions*. The rationale for dividing a problem statement into *initial partitions* is usually motivated by the application domain at hand and the personal judgement of domain experts. So from a requirements engineering perspective there are no common criteria for building *initial partitions* for all possible application domains.

> Definition 7 (Initial Partition) – An *initial partition* is an arbitrary fraction of the overall problem statement, constraining the area of discourse of an analyst team. ◊

Initial partitions in this work are regarded as organisational units assigned to a given analyst team. As a matter of consequence, *initial partitions* have to be built before the actual requirements elicitation starts. *Initial partitions* as organisational units are the central prerequisite for the concurrent use of analyst teams, as they allow requirements analysis to be accomplished in a reasonable time frame. The requirements elicited within a given *initial partition* can be documented in various forms. In this work we use the term *partition* for a model documenting the requirements of an *initial partition*. For the time being we do not regard a concrete modelling technique and base the discussion in this section on an *abstract model* (see Definition 3, p. 31).

> Definition 8 (Partition) – A model documenting requirements of an *initial partition* is called partition. ◊

Suitable UML views and *constructs* for concurrent requirements engineering are identified in Chapter 5. Whenever the selected UML subset discussed in Section 5.2 has been used to document requirements, we call it a *component schema*.

The Character of Partitions

Independent of a concrete modelling technique and the applied tool set, we want to detail the principal character of *partitions* in concurrent requirements engineering. In this context we distinguish between *disjunct* and *overlapping partitions*.

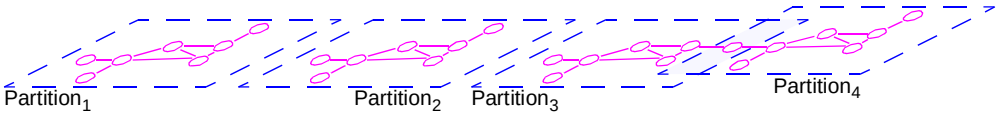
> Definition 9 (Disjunct Partitions) – Two partitions are called *disjunct*, if and only if, they do not contain common elements. ◊

> Definition10 (Overlapping Partitions) – Two partitions are called *overlapping*, if and only if, both of them contain common elements. ◊

Common in this context means, that the same *element* appears in different partitions. Since the identity of *elements* depends on the applied modelling technique, we do not further detail this consideration. Figure 15 depicts this distinction.

Cost efficiency demands that individual partitions are kept disjunct, i.e. a specific partition within the problem domain should not be analysed by two teams at the same time. However, this gives rise to the important question of, whether disjunct partitions automatically lead to disjunct analysis results.

Figure 15 Disjunct versus Overlapping Partitions



The Scope of Requirements

Overlapping partitions imply, that requirements may affect several *partitions. Non-temporal requirements* such as *performance* or *reliability* affect entire systems and therefore also affect several partitions, although this fact may not become apparent to the *partition owners*. This observation leads to the following definition:

> Definition 11 (Scope of Requirements) – The *scope of requirements* refers to the number of *partitions* affected by a given requirement. Requirements are called *local* if they only affect a single *partition*. A requirement is called *global*, if all involved *partitions* are affected and is called *inter-partition requirement*, when at least two distinct *partitions* are affected. ◊

In concurrent requirements analysis with a set of *initial partitions*, elaborated in isolation, the individual requirements statements may at first glance be considered as local. But this does not exclude interference with requirements stated in other partitions. The fundamental problem of global versus local consistency becomes obvious in this example. Central project management, or top-down strategies in general, ensure global consistency, since the decomposition process will always aim for disjunct divisions. In bottom-up strategies, however, this statement is much more complicated, since local consistency does not automatically guarantee global consistency. A requirements statement deemed locally consistent can be called globally consistent, when it does not contradict any statement in all other partitions. As we will see in Chapter 8, this distinction is quite problematic. For the time being, the distinction of *local, inter-partition* and *global requirements* is sufficient for our discussion of concurrent requirements engineering.

Environment and Contents of a Partition

A model documenting requirements of an *initial partition* also has their its own *system boundary*, which we call *partition boundary.* Therefore we also have to distinguish between the *partition contents* and the *partition environment*. In order to avoid method-specific details of a concrete modelling technique, we discuss this idea on the basis of the abstract model introduced in Definition 3, p. 31.

> Definition 12 (Partition Content) – *Partition contents* refer to the set of all *elements* that for any bi-partitioning of the set, *relationship*s exist among *elements* in the two subsets. ◊

Definition13 (Partition Environment) – *Partition environment* refers to all *elements* not in the *partition content* maintaining relationships to *elements* of the *partition content.* ◊

The distinction between the *partition* and the *system boundary* is crucial since the *partition environment* cannot automatically be considered as interface requirements in the sense of Section 3.3.3. Individual *elements* of the *partition environment* may represent *inter-partition requirements*, although this does not become apparent from the *partition owner's* perspective. As a matter of consequence the *scope* of a given requirement documented in a given *partition* has to be reviewed before *partitions* can be integrated. *Inter-partition requirements* also may lead to conflicts detailed in Section 7.3, pp. 131.

Repositories

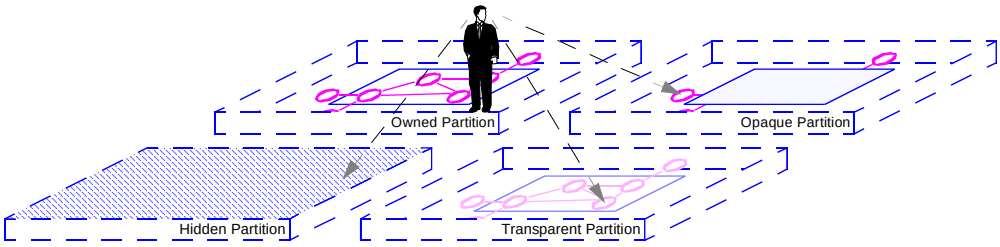
A repository provides persistent storage for *partitions* documented with a CASE tool. A *repository* is called *centralised* if *all* partitions are persistently stored in a *single* repository. A *repository* is called *distributed* when *each* partition is persistently stored in an *individual* repository.

Analyst's Horizon

Distributed repositories, give rise to another challenge, namely the visibility of *elements* and *relationships* of a *partition* to *external analysts*. In order to avoid a detailed discussion of access rights to repositories, we use the term visibility in its colloquial sense. Although we always assume that *external analysts* are not supposed to change a given *partition*, coordination among analyst teams requires at least, that they can 'see' *elements* and *relationships* of a given *partition*. Here we are especially interested in the visibility of *partition content* and *partition environment*, which we call the *analyst's horizon*.

Definition14 (Analyst Horizon) – The analyst's horizon defines the visibility of *elements* and *relationships* of a given *partition* to *external analysts.* ◊

Figure 16 Differing Analyst's Horizons



Regardless of the applied *modelling technique* we may distinguish four *analyst horizons*. A partition is called *hidden*, if *partition contents* and *partition environment* are invisible to *external analysts*.

A *partition* is called *opaque*, when only the *partition environment* is visible to *external analysts*. A partition is called *transparent*, if *partition environment* and *partition content* is visible to external analysts. This distinction is crucial for the management of *component schema relationships* detailed in Section 8.2.

4.3 Concurrent Requirements Engineering in the Air Traffic Management Domain

This section introduces our requirements engineering process, developed for a series of European air traffic management projects [Wor96, WFW97, WCK97]. As argued in Section 3.3.4.4, we prefer use case driven analysis for requirements elicitation. The requirements are documented in terms of *component schemas*, our UML subset for tailored requirements engineering. The process can be divided into the following steps:

1 Dividing the problem domain into (arbitrary) initial partitions

2 Analysing and documenting each initial partition in isolation ...
  - 2.a Selection of relevant business processes
  - 2.b Requirements elicitation based on complex use cases
  - 2.c Structuring requirements using object-oriented analysis

3 Integrating the partitions

4 Requirements validation

5 Establishing a logical architecture

6 Writing the software requirements specification

Comparing Use Case Driven Analysis with the Requirements Engineering Process

Apparently the sequence of activities in use-case driven analysis does not match the requirements engineering process, as introduced in Section 3.2. Although the analysis of business processes can clearly be related to *requirements elicitation*, writing down the activities in terms of a *complex use case* already incorporates a documentation aspect. Within *requirements analysis and negotiation* an approved negotiation process involving different stakeholders decides which elicited requirements are to be accepted. Sheer size, especially in industry-scale applications, demands a great deal of modelling, in order to understand the intricate nature of the problem statement. Problem analysis models, independent of the modelling technique used, provide a stable and rational basis for accepting requirements. Since individual UML views cannot be related to distinct phases within the requirements engineering cycle, we prefer the term *problem analysis* whenever we refer to the *activity of structuring requirements* in terms of a given modelling technique. In this work, the entire problem statement is divided into a set of *component schemas*, modelling distinct *initial partitions* from given perspectives. In this context, we want to assure that all involved analyst teams build the same system. Problem analysis models generally represent an important and valuable contribution for understanding the problem statement.

They also provide a stable basis for the creation of quality software requirements specifications, as detailed in Section 10.3. Having defined the applied requirements engineering process, the following subsections detail selected aspects of this approach.

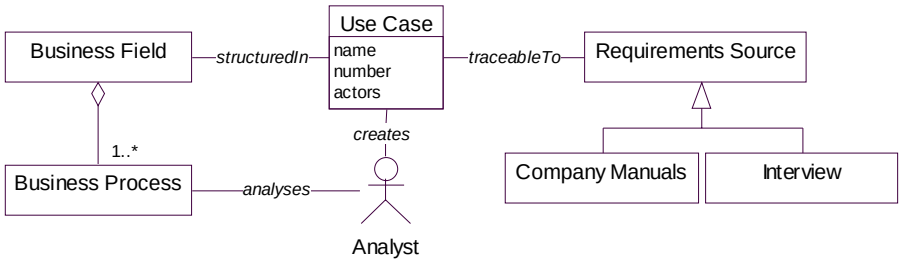
4.3.1 Building Initial Partitions

The initial incentives for the development of new systems often have a fairly abstract character. The decision may be driven by some kind of need, new business fields or simply the feeling, that information technology has the potential to improve the overall efficiency of an organisation. Having investigated general feasibility, a more concrete project organisation model has to be put into place. Process orientation, as the state-of-the-art management paradigm, can effectively support the requirements analysis process for a future system. Initial partitions are defined through the selection of relevant business processes. Affected business processes are gathered and the detailed analysis are assigned to a set of analysts. However, the activities of industry-scale organisation can be divided into:

- business fields and
- individual business processes

Business fields comprise a set of business processes, so initial partitions in this context are formed by business processes deemed to be crucial for the market success of an organisation. Then, selected business processes are assigned to analysts, who have to study them in detail. This task can already be performed in parallel. The creation of use cases as an initial set of documented requirements can be reduced to the following classic information sources, interviews with involved business users and document analysis. Business processes form the first level of *initial partitions* and interviews or consulted documents provide the first set of information, the *requirements source*. All *requirements sources* have to be identified and recorded, in order to relate documented requirements to their requirements sources, which is called *requirements source traceability* by Sommerville [SS97]. Figure 17 depicts the relationship between business fields, use cases, analysts and available requirements sources.

Figure 17 A Metamodel for Business Fields, Use Cases and Requirements Sources



4.3.2 Use Case-Driven Requirements Elicitation

Having selected relevant business processes in the larger frame of *business fields*, individual business processes have to be documented. The involved business users and the course of activities within a dedicated business process have to be analysed. The description of the detailed structure of complex use cases proposed in this work can be found in Section 5.2.4. Use cases within a given business field are then structured in terms of an object model. First of all, *use case activities* have to be structured in UML sequence diagrams, representing *functional requirements*. Adding structural properties to class specifications then shapes the *structural requirements* in terms of UML static structure diagrams. Individual business fields are concurrently analysed, resulting in *component schema*s. Each *component schema* then has to be reviewed by the incorporated business users and domain experts.

Integration of Component Schemas

By the end of requirements analysis and negotiation, the set of elaborated *component schemas* documents the overall problem statement. In order to validate the entire specification, *component schemas* have to be integrated. Now the entire specification has to be validated. *Inter-partition* and *global requirements* have to be identified and possibly regrouped. Requirements negotiation terminates with an agreed model of requirements, documenting the envisioned system. Since this subsection only introduces the process of use-case driven requirements elicitation, we do not detail the intricacies of individual activities here.

4.3.3 Defining a Logical Architecture

The integrated analysis model needs further groupings. *Initial partitions* as organisational units have to be converted into a logical architecture describing the overall problem statement in terms of UML subsystems. Subsystems have a recursive character, i.e. they may contain other subsystems and are represented in terms of UML packages. The definition of a software architecture in the sense of Shaw [SG96] is clearly a design activity and follows a completely different set of concerns.

Partitions versus Logical Architecture

Packages realise a significant change in how the problem statement is viewed. While initial partitions mostly follow a well defined perspective – top-down, business processes or stakeholders – packages follow different criteria. With respect to the system design later on, initial partitions can be regarded as arbitrary, while packages group the problem analysis model according to a set of defined rules. As a result, arbitrary groupings induced by the initial reduction of analysis complexity now have to be replaced by systematic and hierarchic structure, following engineering practice. According to Heide Balzert [Bal95] subsystems should define a logical and structured unit which

- guides the reader through a problem analysis model
- describes a given problem aspects in isolation

- groups logically related elements, e.g. interacting class specifications providing the required functionality
- enables isolated design and implementation of the packages via well defined interfaces.

The classical modularisation principle *high internal* and *loose external couplings* is the most primitive criteria for the definition of packages. As a rule of thumb, UML packages should not separate generalisation/specialisation hierarchies, nor should they separate *compositions*. The central objective of this activity is the definition of partitions in terms of logically related elements. A logical architecture excludes all design concerns such as middleware and the persistent storage technology.

4.4 Coherence of Component Schemas

This section details the consistency of industry-scale requirements specifications built on the basis of concurrently elaborated *component schemas*. Whenever something has been split into smaller fractions, we have to clarify how the entirety can be reassembled on the basis of these fractions. Referring to the old Roman divide and conquer principle, Michael Jackson puts it the following way: '*... having divided to conquer, we have to reunite to rule*' [Jack90].

Consistency

In this context we are interested in the specific relationship among *component schemas* modelling *initial partitions*. The consistency of the resulting overall specification built on this basis of *component schemas* is especially crucial for the quality of the resulting software requirements specification and all other products down-stream in the software development cycle. Consistency in this context aims for an accurate and flawless description of the entire problem statement from an external perspective. Consistency is also one of the central quality attributes of software requirements specifications in the IEEE-830 standard. First of all we want to clarify the meaning of the term consistency, in colloquial language as well as in the context of concurrent requirements engineering. WordNet proposes the following meanings for the term »consistency«:

> "1.consistency, consistence, body --
> (the property of holding together and retaining its shape;
> "when the dough has enough consistency it is ready to bake")
> 2. consistency, consistence --
> (a harmonious uniformity or agreement among things or parts)
> 3.consistency --
> (logical coherence and accordance with the facts: "a rambling argument
> that lacked any consistency")
> 4.consistency --
> ((logic) an attribute of a logical system that is so constituted that none of
> the propositions deducible from the axioms contradict one another)" [WN98]

The composition of an overall problem analysis model formed by a set of *component schemas* is best expressed in meaning 2, while a formal mathematical definition is provided in meaning 4. With respect to requirements engineering, Thayer and Dorfmann provide the following definition:

> "... a specification is consistent to the extent that its provisions do not conflict with each other, other specifications, or objectives." [TD97, p. 446]

So this definition roughly corresponds to meanings three and four in the WordNet definition. According to Thayer and Dorfmann, the absence of conflicts is the primary criteria for consistency. But what are conflicts in concurrent requirements engineering using *component schemas*? The IEEE-830 standard states:

Conflicts in Requirements Specifications

> "There are three types of likely conflicts in an SRS:
> a) The specified characteristics of real-world objects may conflict. [...]
> b) There may be logical or temporal conflict between two specified actions. [...]
> c) Two or more requirements may describe the same real-world object but use different terms for that object." [IEEE830, p. 186]

Independent of the concrete nature of conflicts, the *scope of requirements* and the evolution of requirements specifications elaborated in autonomous analyst teams automatically calls the overall consistency into question. Assuming that *component schemas* are locally consistent, the term *conflict* refers to an antagonism or irreconcilability among elements stemming from different *component schemas*.

Definition 15 (Conflict) – A *conflict* is a relation among antagonistic or irreconcilable *elements* stemming from different *component schemas*. ◊

Without describing the concrete nature of conflicts among *component schemas*, we want to discuss this problem from a rather general perspective. Due to the simultaneous elaboration of *component schemas* their contents constantly evolve. The evolution of *component schemas* may lead to conflicts, hence consistency cannot be guaranteed. Enforcing consistency among partitions on a permanent basis demands constant coordination among analyst teams and therefore consumes additional effort. Consequently, we have to tolerate conflicts, and consistency may only be assured for given moments in time. Without strict enforcement of consistency, we are still interested in an orderly relation among *component schemas*. This orderly relation is influenced by the following aspects:

- Detection of conflicts:
  A definition of conflicts between *component schemas* and means for their detection.
- Notification of conflicts:
  Means for the detection of conflicts have to be complemented by means for the notification of the affected analyst teams. Conflicts have to be reviewed and resolved in a cooperative manner, in order to avoid overriding of requirements.

- Ensuring minimality:
  Means for detecting redundant and possibly incompatible specifications of the same real-world thing in different *component schemas* ensures the minimality of the overall problem analysis model.
- Settlement of inter-partition requirements:
  The settlement of *inter-partition requirements* plays an important role for overcoming the self-contained perspective of individual *component schemas*. *Inter-partition requirements* have to be announced to the partition deemed responsible for their settlement.

Providing means for *conflict detection*, *conflict notification*, *ensuring minimality* and *settlement* of *inter-partition requirements* leads to an orderly elaboration of *component schemas*. Although we cannot exclude conflicts and consequently cannot comply to the strict definition of consistency in terms of Thayer and Dorfmann [TD97], *component schema relationships* provide means to ensure that concurrent requirements analysis at least can be performed in an orderly fashion. In the described presence of conflicts, we use the term *coherence* to indicate, that requirements analysis is performed in the quoted orderly fashion.

> Definition16 (Coherence) – *Component schemas* are called *coherent* when *conflicts* have been resolved, *redundancies* and *inter-partition requirements* are detected and coordinated among the affected analyst teams. ◊

The notion of *conflict* and *redundancy* is detailed in Section 7.3, pp. 131 while *coordination* is discussed in the context of the management of *component schema relationships* (see Section 8.3, pp. 176). Improving the *coherence* of *component schemas* affects *project organisation*, *project coordination* and *conflict management*. This subsection will only summarize practical aspects of concurrent requirements engineering, that we detail in Section 8.3.

Project Organisation

The organisation of a project is one of the central factors, with a direct impact on the entire project life-cycle. Underestimating the complexity of a project leads to cumbersome and overlapping partitions and therefore threatens the project success. The first and most important factor in this context is the assignment of staff to *initial partitions*. Master-multiple organisations in particular leave sufficient room for a dedicated team to study and assess the overall impact of inter-partition and global requirements. Concentrating on the overall problem statement also allows project progress to be monitored. Depending on the choice of organisation model and the degree of autonomy of analyst teams, coherence and project progress become assessable.

Project Coordination

The degree of autonomy of individual expert teams has a direct impact on the coordination among analyst teams. In this context the size of *initial partitions* and the desired degree of coherence defined by the project management determine the coordination efforts. Coordination in concurrent requirements analysis is related to realigning partitions boundaries, coordinating *inter-partition* as well as *global require-*

ments, establishing responsibilities for inter-partition requirements and organising efficient communication among analyst teams. A realignment of partition boundaries may become necessary, due to the logical character of initial partitions. A clearly defined justification of partitions only becomes available when the *component schema* representing it has become relatively stable. The relationship among distinct partitions is basically determined by *inter-partition* and *global requirements*. Responsibilities have to be defined for certain *inter-partition requirements*. Often it is not exactly clear to which partition given requirements belong to. Therefore, analyst teams have to establish the responsibility for the elaboration of emerging inter-partition requirements. Finally, an efficient communication process has to be defined.

Conflict Management

Depending on the chosen degree of autonomy a concrete policy for the management of conflicts has to be established. This activity incorporates the cycles for the assessment of conflicts in the involved partitions, as well as the propagation of inter-partition requirements. In this context suitable means for detecting, propagating and resolving conflicts have to be defined.

4.5 Discussion

This chapter complemented our perspective of requirements engineering towards conducting projects concurrently using several analyst teams. In this context we introduced differing team organisations, that we call *centralised*, *autonomous* and *master multiple*. Within this organisation models, *partition owners* complement the list of stakeholders involved in requirements engineering projects. Then we concentrated on the specific character of partitions. Mapping *partitions* to *repositories* provides persistent storage for *component schemas* modelling distinct fractions of the problem statement. In this context, the *analyst horizon* determines the visibility of *partition content* and *partition environment* to external analysts.

Then we introduced our methodology for *use-case driven analysis* in the frame of concurrent requirements analysis. Building *initial partitions*, describing individual activities within requirements elicitation, and introducing our perspective of a *logical architecture* are the central contributions of this section. Finally, we concentrated on the consistency of specifications, built on a set of *component schemas* describing *initial partitions* in a self-contained fashion. *Conflict notification*, *ensuring minimality* and *settlement of inter-partition requirements* have been identified as important factors for increasing the *coherence* of constantly evolving *component schemas*.

5 Concurrent Object-Oriented Problem Analysis Using UML

This chapter introduces *component schemas* as our specific approach for concurrent requirements engineering. We discuss the specific character of UML as a *model-based multi-viewpoint specification* and then we analyse which of the UML views defined in OMG standard [OMG97a] are relevant in requirements engineering. A deliberate examination of the available UML constructs assures their solution-neutral character. This finally leads to our definition of *component schemas* documenting the requirements of *initial partitions*.

5.1 Perspectives and Viewpoints

Our concurrent requirements engineering approach divides the problem statement into a set of *initial partitions*, which are represented in terms of a *component schema*. Character, notation and *constructs* of *component schemas* are detailed in this section. Since a number of different stakeholders are involved in concurrent requirements engineering projects, all of them with different educational and possibly cultural backgrounds, a set of definitions must be introduced to clarify the character of the UML in general and the select subset UML viewpoints and *constructs* used in *component schemas.*

Perspective

System users form an important group within the participating stakeholders, but only few of them ever see the complete system in all its details. They define requirements mostly related to their assigned business processes. *Information officers*, who are responsible for long-term development of an information infrastructure, are interested in fitting new systems into an existing infrastructure. A perspective in colloquial terms refers to the stakeholders personal standpoint, i.e. the real-world things a given stakeholders actually sees within a given *initial partition.*

> Definition17 (Perspective) – A *perspective* is a fraction of the problem statement, as perceived by a distinct stakeholder. ◊

Consequently each partition independently of the used modelling technique incorporates the perspective of its stakeholders.

Aspect

The personal interest of individual stakeholders forms a second dimension and is called *aspect* in this work. The aspect can be rephrased through the simple question “What is the stakeholder looking for?”

> Definition18 (Aspect) – An *aspect* is a considered characteristic of the real-world things under analysis. ◊

In order to avoid lengthy discussion concerning business administration problems we, take a rather pragmatic approach, reducing this question to three aspects, that according to DeMarco [DeM82], fully describe any software system: *data*, *functions* and *states*.

Viewpoints

Combining *aspect* and *perspective* of given stakeholders with a modelling technique describing the perceived reality leads to our definition of viewpoints. Rumbaugh et al. are using the term view in their UML reference guide for the individual modelling techniques. In order to avoid mixing up the terms *view* and *viewpoint* we will use the term UML view.

Definition19 (Viewpoint) – A mental position from which a problem statement is analysed. This position includes a defined *perspective* with a determined scope, a observed *aspect* and a distinct *modelling technique*. ◊

Multi-Viewpoint Specifications

Modelling a set of perspectives with the same modelling technique can be called *multi-perspective specification*. A *multi-viewpoint specification* describes the problem at hand from an arbitrary set of *perspectives* using different modelling techniques. *Multi-viewpoint specification techniques* can be distinguished according to the way the individual *viewpoints* are organised:

Definition20 (Model-Based Multi-Viewpoint Specification) – A m*ulti-viewpoint specification technique* based on a combination of related modelling techniques, each of them highlighting a specific *aspect* of the problem statement. ◊

OMT is a prominent example in this category, where a set of existing modelling techniques have been combined in order to provide “best-practice” to analysts [RBP+91].

Definition21 (Aspect-Based Multi-Viewpoint Specification) – Aspect-based multi-viewpoint specifications describe the problem statement from a set of determined aspects. ◊

RM-ODP is an example of this kind of *multi-viewpoint specification* [ODP P1], where business objectives, static structure, object interaction, distribution and technology form the individual OPD-viewpoints.

Definition22 (Stakeholder-Based Multi-Viewpoint Specification) – In stakeholder-based multi-viewpoint specifications, the individual *viewpoints* are determined by the specific role or post involved *stakeholders* have in an organisation. ◊

Viewpoint Relationships

Modelling techniques incorporating several viewpoints in a single specification give rise to so-called viewpoint relationships. They relate arbitrary elements specified in different viewpoints. The following basic types of viewpoint relationships can be distinguished.

- *Arbitrary relationships*
  In this category inter-viewpoint relationships are not formally specified. The relationships have an arbitrary character and may be defined in an ad-hoc manner according to the analyst's needs. RM-ODP is an example of this kind of approach [ODP P1]
- *Additive character*
  Additive formalism can be compared to puzzles, where each piece has to be fitted in in order to finish a complete specification. Individual *viewpoints* concentrate on a given aspect but remain unrelated to other *viewpoints*.
- *Projections*
  The organising principle for this kind of specification is an integrated model. Each viewpoint can be regarded as a projection of this common base under specific aspect. Like the additive formalism, there is currently no approach entirely defined on a common base.
- *Metamodel*
  In a metamodel-based formalism, inter-viewpoint relationships are specified in a metamodel. UML is an example of this approach [OMG97a]. Unlike a common base, metamodels do not imply a semantic or mathematical integration of the used modelling techniques.

Viewpoint Organisation in UML

The Unified Modelling Language (UML) provides *model-based viewpoints*, related through a *metamodel*. The individual *viewpoints* focus on the *aspects data*, *function* and *state*. Due to conceptual overlaps among statecharts and sequence diagrams, UML viewpoints cannot be considered as orthogonal. UML comprises the following modelling techniques: statecharts, use case, sequence, static structure and activity diagrams. Unlike OMT, UML does not prescribe a defined process. Analysts can individually select which diagrams are relevant for a given project and the order of creating these diagrams. Although not directly relevant in this context, we assume an order of activities as defined in the *Unified Software Development Process* [JBR99].

5.2 Component Schemas: Viewpoints, Constructs and Notations

This section selects relevant UML views and *constructs* in order to assure a solution-neutral outcome of concurrent requirements engineering projects using *component schemas*.

5.2.1 Relevant UML Views

UML provides a *model-based multi-viewpoint specification technique*, in order to support the entire software development cycle. In this section we want to analyse the different UML views with respect to their relevance in requirements specification. The central objective in this context is solution-neutral description of the problem

statement. Component and deployment diagrams, for instance, are clearly implementation-related and therefore should not be used in problem analysis. This work will concentrate on use case, sequence and static structure diagrams, as the most relevant UML views for requirements analysis. Although statecharts may be of interest for requirements analysis, we leave them out in this work. Main motivation for this omission is the conceptual overlap between statecharts and sequence diagrams. Consequently, all diagrams printed in italics in Table 3 will not be considered. This UML subset assures orthogonality of viewpoints and a minimum of ontological construct redundancy in the sense of Wand and Weber [WW93]. Using a small number of carefully chosen *constructs* often eases comprehension for domain experts and therefore may improve the overall quality of problem analysis in requirements engineering.

Table 3 UML Models and their Featured Aspect

	UML Model	Featured Aspect	Considered in Component Schemas
Analysis	Use case diagram	Business processes, actors, activities	yes
	Sequence diagram	objects, messages, order in time	yes
	Static structure diagram	Classes, association	yes
	Statechart diagrams	*States, events, state changes*	*no*
	Collaboration diagram	*objects, messages, order in time*	*no*
Design	*Component diagram*	*Components (compile-time), dependencies*	*no*
	Deployment diagram	*Components (run-time), nodes, dependencies*	*no*

Component Schemas

Component schemas, our UML subset tailored for requirements engineering, form a multi-viewpoint specification. Using *component schemas* in concurrent requirements engineering leads to the following situation. *Initial partitions* are assigned to given analyst teams. The *component schema* documents requirements in terms of the selected UML subset. Since UML comprises a set of defined viewpoints, the overall problem statement is described in terms of a *multi-multi-viewpoint specification*. In order to clarify the terminology, we use the term *component schema* for a UML model describing a distinct *initial partition* within the overall problem statement.

In the current UML standard [OMG97a], use cases are only an iconic representation of business processes, comprising their name and the involved actors. The order of activities within a given use case is documented in a related sequence diagram. We believe that for participative requirements elicitation, especially in industry-scale projects, domain experts are not able to model the business process directly in terms of a sequence diagram. Plain text descriptions are usually more comprehensive to domain experts expressing their knowledge.

The Value of Use Cases in Requirements Engineering

Although frequent claims criticize a lack of scalability of plain text descriptions, our experience in the air traffic management domain confirmed their practicability even in industry-scale projects [Wor96, WCK97]. We believe that only plain text descrip-

tions provide a stable and comprehensive requirements basis for both domain experts and analysts with a computer science background. Sequence and static structure diagrams are subsequently derived from the use cases. Improving the structure and defining a pragmatic decomposition mechanism for use cases was one the central objectives in the research project "*Data Modelling for Advanced Flight Data Management*" in collaboration with the EUROCONTROL Experimental Centre Brétigny-sur-Orge France [Wor96, WCK97]. Complex use cases, as one of the central contributions of this work, are detailed in Section 5.2.4, p. 83.

In order to guarantee a solution-neutral character of *component schemas* we still have to review the *constructs* defined in *component schemas*. In this context we are mainly looking for *constructs* that originate from programming languages, such as C++ or Smalltalk. So the individual constructs defined within use cases, sequence and static structure diagram are evaluated in this section. Carefully chosen viewpoints with selected and orthogonal *constructs* improve the comprehensiveness of specification to business users without extensive modelling experience and also provide a solution-neutral documentation of requirements gathered in a *component schema*.

5.2.2 Introducing the UML Metamodel

This subsection introduces the general structure of the UML metamodel and the mechanisms applied in order to provide sound semantics. The discussion is based on the 1997 OMG proposal [OMG97a, OMG97b, OMG97c]. Recent documentation, such as the '*Unified Modeling Language Reference Manual*' [RJB99] and the '*Unified Software Development Process*' [JBR99] are based on revisions of the metamodel, which have been summarized under the term '*UML version 1.3*'.

5.2.2.1 Terminology and Conventions Used Throughout this Section

Analysing a given modelling technique requires a concise use of terms on different levels of abstraction. Apart from the four layer architecture defined by the ISO IRDS [ISO10027], additional conventions have to be introduced. On the one hand we have the UML terminology, on the other hand we want to describe the meaning of modelling constructs in broader terms. In this work we distinguish between an ontological level and the specific UML terms. This approach may also draw correspondences and differences to other modelling languages such as the Entity-Relationship Model [Chen76].

Bunge's ontology provides the baseline for this discussion [Bun77]. The real-world, as perceived by an analyst, is composed of *things*. The intensional description of a thing is called *artefact*. Things posses *properties*. *Properties* may be observed as static or behavioural. *Things* may be related to each other, i.e. they participate in a *relationship*. *Relationships* can therefore be regarded as pairs of things. *Things* may be organised in taxonomic relationships that are *generalisation* and *specialisation*.

Things might also be constituted of other *things*, i.e. the *things* are related to each other via an *aggregation*. We also assume that the whole cannot exist without its constituting parts. A *construct* is a means of describing selected *aspects* of *things* in terms of a modelling language. In UML the construct *object* is applied to describe *things*.

Objects, Things in Sequence Diagrams

However, following the intension/extension dichotomy, we still have to detail the term *thing*. UML sequence diagrams depict the interaction of objects exchanging messages. But what is the exact character of these objects, with respect to things in the real world? Objects in a sequence diagram are only regarded as abstractions of a particular real-world things interacting under a distinct objective. Message passing in this context is an interpretation of a conveyance of information among real-world things that abstract from the encapsulated internal structure of the involved *things*. Consequently, sequence diagrams describe *prototypical objects* as derived from use cases. The messages depicted in sequence diagrams lead to the allocation of operations to classes in static structure diagrams.

Table 4 Metalanguage for the Introduction of UML Constructs

Term	Denotation
Thing	Perceivable elements constituting the real-world
Artefact	An intensional description of a collection of real-world things. An artefact consists of a name and a construct
Construct	A means of describing a selected aspect of real-world things in the model world
Property	Things possess properties
Relationship	A relationship represents a belonging between things
Generalisation	A taxonomic relationship between a more general and a more specific element
Specialisation	A taxonomic relationship between things
Aggregation	Things may be constituted of other things
Semantic Relationship	Summarizes generalisation, specialisation and aggregation
Prototypical Object	Prototypical objects describe the interaction of real-world things
Class Specification	A class specification is an element in component schemas' static structure diagrams representing the structural aspect of a distinct thing.

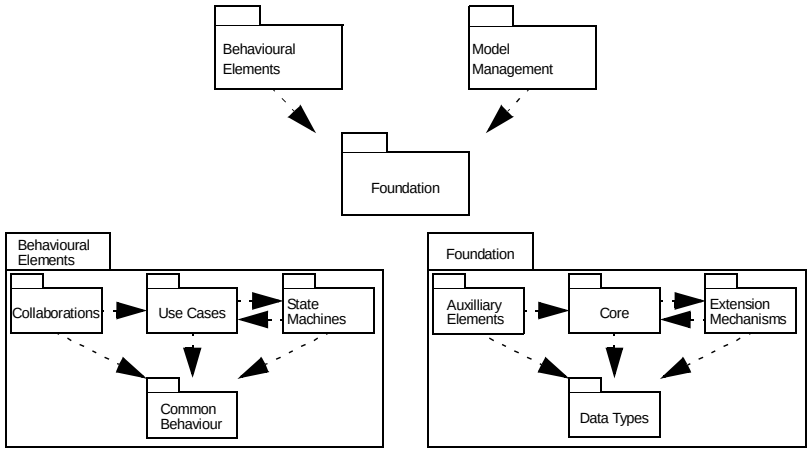
To this terminological convention we add a notational one. References to UML metamodel elements are written in the Courier font. So the term `ModelElement` therefore refers to a UML metaclass carrying this name.

5.2.2.2 The UML Metamodel Structure

The UML metamodel follows the principles of the ISO IRDS [ISO10027] four layer architecture and has been decomposed into the set of packages depicted in Figure 18. Since we concentrate on UML static structure diagrams as the backbone of compo-

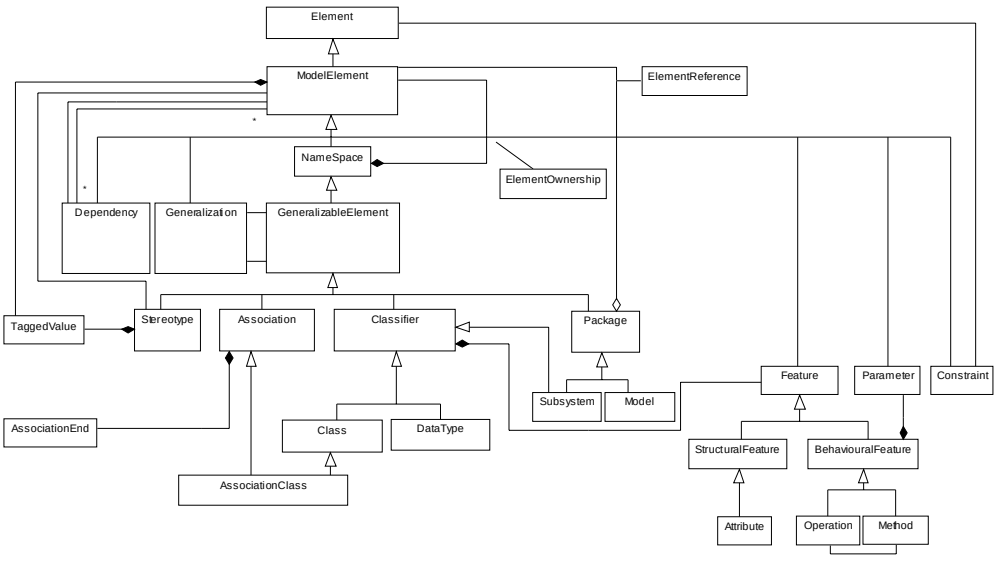
nents schemas, we only partly follow the original package structure, as described in the semantics document [OMG97a]. Identifying relevant *constructs* for *component schemas* does not require covering the entire metamodel, since design-related diagrams have already been excluded. Figure 18 depicts the metamodel structure as described in the UML Semantics.

Figure 18 UML Metamodel Organisation



The discussion in this section therefore mostly concentrates on the `Foundation` and partly the `Model Management` packages. Relevant constructs for this work have been integrated into the static structure diagram depicted in Figure 19:

Figure 19 Relevant Constructs of the UML Metamodel for Static Structure Diagrams



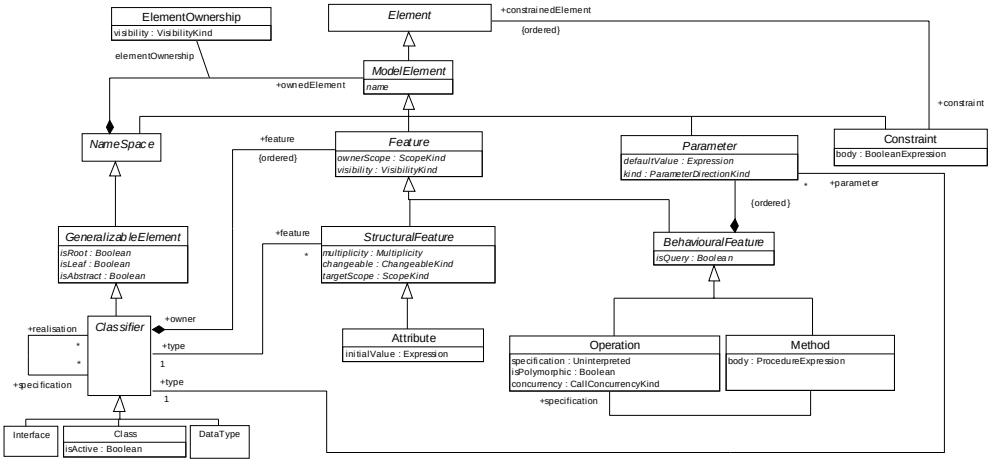
5.2.3 Considered UML Static Structure Constructs

In this section we introduce solution-neutral *constructs* for *component schemas* in the following way. First of all, we depict the relevant partition of the UML metamodel. Then we discuss all presented constructs and evaluate their adequacy for requirements engineering. Finally we conclude this discussion with a revised metamodel depicting our UML subset tailored for requirements engineering.

5.2.3.1 Classes

Classes are the central concept within most object-oriented modelling techniques. Classes are an intensional description for a collection of real-world things sharing the same properties and relationships. Figure 20 depicts the relevant part of the UML package `Core`.

Figure 20 Classes in the UML Core Package



Classes and Properties

UML introduces a uniform treatment of all constructs through the abstract metaclass `Element`. Elements may carry a name which has been realised in the property `name` in the metaclass `ModelElement`. Metaclass `Class` represents things. Properties are described through specialisations of metaclass `Feature`. Structural properties are represented in metaclass `Attribute`, while behavioural properties are modelled in metaclass `Operation`. Classes, operations and attributes carry a name since they are all specialisations of metaclass `ModelElement`. However, names are organised in a hierarchic fashion, represented through the aggregation `ElementOwnership` between the metaclasses `NameSpace` and `ModelElement`. Since the metaclasses `Feature` and `NameSpace` reside on the same specialisation level, features cannot span a name space of their own. Property names are therefore embedded in the name space of their `Classifier`. This approach reduces name clashes between properties and classes on the model level. Thus properties are encapsulated within their

`Classifier`. The property `visibility` of metaclass `Feature` may be ignored, due to its explicit reference to *access specifiers* in C++ [Str93]. Properties in C++ may be specified as *private*, *protected* and *public*. Following the object paradigm, all properties are encapsulated, i.e. they are not directly accessible to external clients. Business users, according to our experience, have significant difficulties in understanding the concrete semantics of *access specifiers* on a programming language level. Another fundamental difference between classifiers and features becomes obvious through the metaclass `GeneralizableElement`. Classifiers may participate in specialisation hierarchies, which is prohibited to features.

Classes versus Types

Metaclass `Classifier` forms a common generalisation for classes and data types. Both constructs share significant similarities, since they are both complex constructs, built from a set of atomic ones. Unlike classes, data types may not comprise behavioural properties.The *construct* data type usually denotes a name domain pair, where the (mathematical) domain represents the set of legal values. Data types may be scalar, such as numbers or characters, or complex, i.e. an arbitrary set of scalar types can be combined under a single name. UML defines data types as follows:

> "A datatype is a type whose values have no identity, i.e. they are pure values" [OMG97a, p. 22]

Therefore objects, as instances of a class, are uniquely distinguishable through their identity, while *values*, the run-time counterparts of types, do not have an identity. UML treats this distinction quite flexibly, since CASE-tools realising the language shall enable code generation for a great number of different programming languages. However, the distinction between types and classes in this work has to be clarified later on.

Classes, Attributes and Types

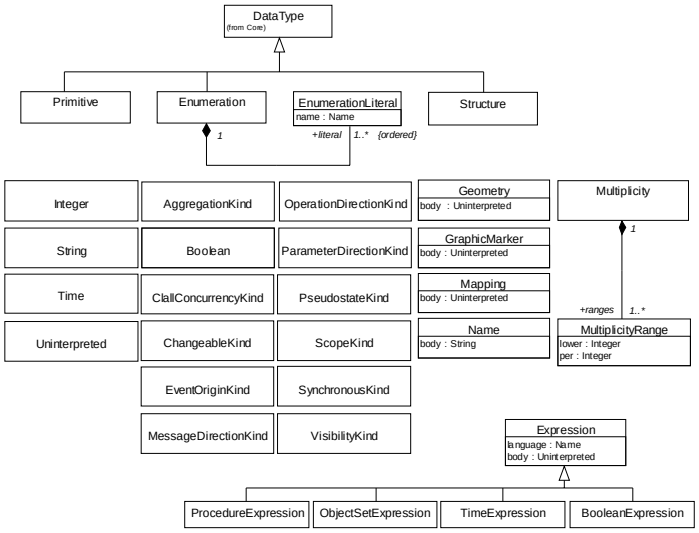
Structural properties are realised as a specialisation of the metaclass `StructuralFeature` in UML. Especially the small difference between the metaclasses `StructuralFeature` and `Attribute` seems quite arbitrary. Structural properties are characterised by a data type, which is realised through a relationship between the metaclasses `Classifier` and `StructuralFeature`. Most inherited properties of metaclass `Attribute` can be considered as implementation-related. The property `ownerScope` in metaclass `Feature`, for instance, originates from the Smalltalk [GR83] programming language, where classes are run-time entities. Via this property, operations and attributes can be bound to classes or objects. The property `multiplicity` defines the number of values stored in a given attribute. An attribute may be turned into a constant via the property `changeable`. Finally the property `targetScope` defines whether an attribute is directly contained in a class or the class only maintains a reference to the attribute. In other words, the property `targetScope` specifies whether an attribute is realised *by value* or *by reference* in the respective class specification. Objects, as instances of classes, may establish their own thread of control, expressed through the property `isActive` in the metaclass `Class`. Concurrency is a clear design concern and therefore will be discarded in *component schemas*. Although the properties `multiplicity` and `initialValue` of metaclass `Attribute` can be deemed as meaningful in problem analysis,

they will not be considered. The other interesting aspect in the metamodel is the recursive `realisation/specification` relationship on the metaclass `Classifier`. Although explicitly related to the metaclass `Interface` in the metamodel, there is an interesting alternate interpretation. A class specification in one *component schema* might be realised as a set of interacting class specifications – a class lattice – in another one. We call this situation a *construct-level correspondence*, detailed in Section 7.3.5. Nevertheless the `realisation/specification` relationship will be ignored in *component schemas*.

Simple versus Complex Types for Structural Properties

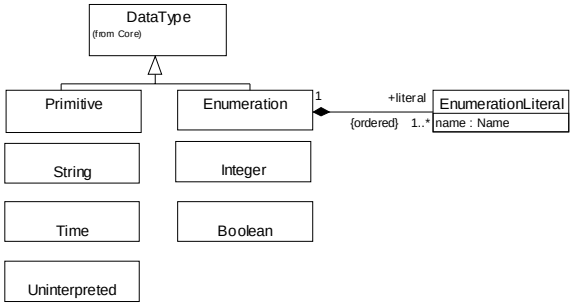
Properties of classes restrict the range of valid values via a data type, expressed in the relationship `type` between the metaclasses `StructuralFeature` and `Classifier`. Metaclass `Classifier` therefore allows uniform treatment of objects and values. According to our experience, industry-scale projects hardly require such an elaborate type system. Determining exact parameters for attributes and return values for operations is usually accomplished in design. The uniform treatment of data types and objects also interferes with the semantic relationship aggregation. In order to clarify this problem, we prohibit the use of class specifications for data types. We further restrict data types to *scalar types*. Additional well-formedness rules, formulated in the Object Constraint Language (OCL), may realise this point without actually changing the metamodel. Besides the possibility of defining complex data types, UML provides a basic type system in the package `Data Types`, depicted in Figure 21.

Figure 21 UML Package Data Types



The data type package makes intensive use of the stereotype concept, which is detailed in Section 5.2.3.4. The data types `String` and `Integer`, for instance, are categorised as primitive types. In order to increase the comprehensiveness of *component schemas*, we propose the simplified type system, as depicted in Figure 22, p. 69:

Figure 22 The revised Metamodel for Data Types in Component Schemas‘ Behavioural Features



Behavioural properties are represented through the metaclasses `Parameter`, `Operation` and `BehaviouralFeature`. As a specialisation of metaclass `ModelElement` behavioural properties have a name, which has to be unique within its classifier. Metaclass `Parameter` allows the definition of an argument list for a given behavioural property. Property `kind` determines the character of an argument, where `in`, `out`, `inout` and `return` are legal values. Each argument is also related to a datatype via the relationship `type` between the metaclasses `Parameter` and `Classifier`. As a result, arguments may be both objects as well as values. Similar to attributes, we restrict the data types for parameters, as discussed in the previous paragraph. Like their structural relatives, the metaclasses `BehaviouralFeature` and `Operation` contain a number of implementation-related aspects. The property `isQuery`, for instance, indicates whether the method implementing the operation may change the internal state of an object or not. The property `isPolymorphic` in the metaclass `Operation` restricts overriding of operations for specialisations of the classifier. The property `concurrency` is related to the property `isActive` in metaclass `Class`. When instances maintain their own thread of control, they may respond to incoming requests quite differently. Usually an object can only respond to one request at a time, but they also may establish their own request queues. As a clear implementation construct, we discard the concurrency of behavioural properties.

Methods & Interfaces

Methods, represented through the metaclass `Method`, model the body, or the implementation of an operation. The metaclass `Operation` specifies the signature that will be implemented through the method. Metaclass `Interface` represents a construct related to middleware platforms such as CORBA or Microsoft’s DCOM, where interfaces become first class-citizens and have to be independent from individual class specifications in a given programming language. The UML metamodel describes interfaces as follows:

> “A declaration of a collection of operations that may be used for defining a service offered by an instance.” [OMG97a, p. 153]

As a result, the metaclasses `Interface` and `Method` have to be discarded. The other interesting aspect in the metamodel is the recursive `realisation/specification` relationship on the metaclass `Classifier`.

"A set of ***Classifiers*** that implement the ***Operation*** of the ***Classifiers***. These may not include ***Interfaces***." [OMG97a, p. 21]

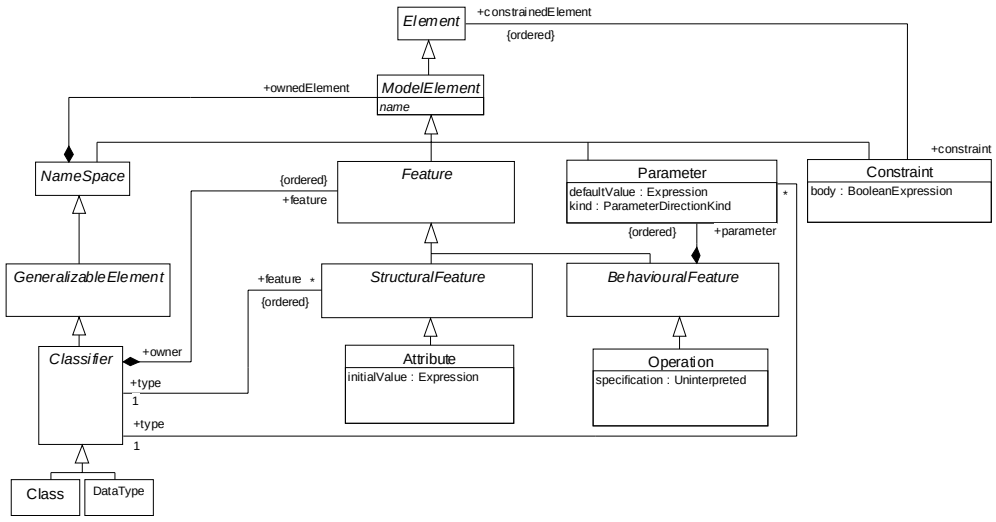
This relationship describes which classes implement the operations actually offered by an interface. An interface may also combine operations offered by a set of classes and therefore can be regarded as an abstraction for interacting classes. Nevertheless, methods and interfaces are irrelevant constructs in requirements engineering, since interfaces are design entities and methods are subject to implementation.

Constraints

The provision of constraints is one of the central advantages of UML. Realising them as a distinct metaclass facilitates the formulation of business rules, which otherwise cannot directly be expressed.

Due to the relationship between the metaclasses `Element` and `Constraint`, business rules may be applied to all UML constructs. Constraints may be formulated in Object Constraint Language (OCL) [OMG97c, WK99] and are represented through the property `body` in the metamodel.

Figure 23 The revised Metamodel for Classes in Component Schemas



Section Summary

Component schemas comprise the following UML constructs deemed relevant in requirements engineering. Classes (metaclass `Class`) comprise structural as well as behavioural properties, realised in the metaclasses `Attribute` and `Operation`. Attributes have a scalar data type and may carry an initial value. Operations are characterized by a name and a list of arguments, represented through the metaclass `Parameter`. Like attributes, parameters have a data type, but we only distinguish between arguments and return types through the property `kind`. Constraints may be applied to all elements in *component schemas*. Figure 18 depicts all *constructs* relevant in *component schemas*.

UML classes allow an integrated documentation of structural and behavioural requirements. Operations express functional requirements demanded by the business users. Attributes define the data contents which have to be remembered by the envisioned system. In other words, classes are the central means of formulating user requirements in *component schemas*.

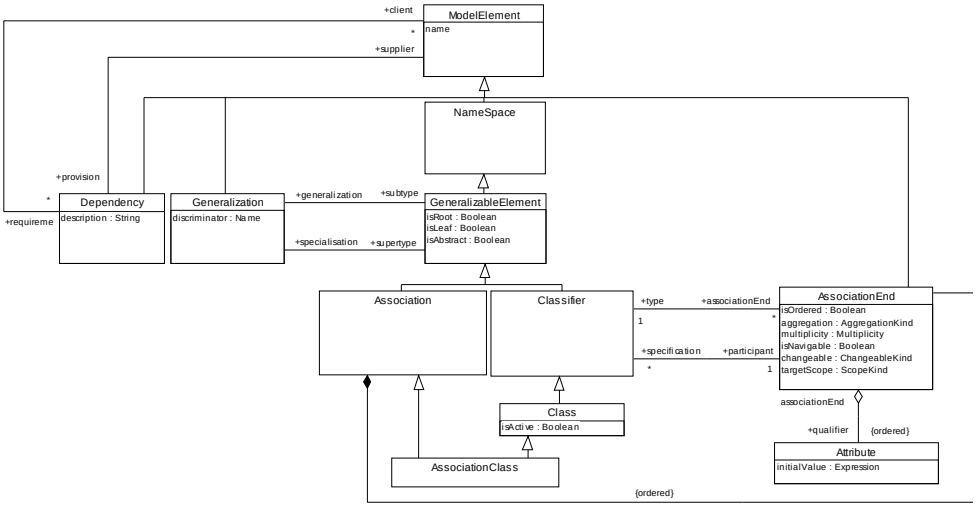
5.2.3.2 Relationships

The semantics of relationships in object-oriented programming languages, as well as in UML, give rise to a set of different realisations with individual advantages and drawbacks. The semantics document gives the following description:

> "An *association* defines a semantic relationship between *Classifiers*; the instances of an associations are a set of tuples relating instances of the Classifiers. Each tuple value may appear at once." [OMG97a, p. 17]

This definition corresponds to the idea of relationships, in the ER-Model [Chen76]. However, other object models, are often based on referential semantics, i.e. relationships have to be regarded as pointers to objects. Especially object-oriented programming languages follow referential semantics. The individual advantages are discussed below.

Figure 24 Relationships in the UML Package `Core`



Deficiencies of Referential Relationships Semantics

In CLASSIC, a structural data model for objects based on description logics [BBM+89], properties as well as relationships are represented as so-called "*roles*". That means they are two place predicates. This leads to a rather "artificial" representation of the perceived reality. Relationships among real-world things become properties of class specifications in CLASSIC. This representation clearly contradicts human perception. Relationships cannot be regarded as properties of real-world things, they

are merely things in their own right. The fact that properties cannot be bound to relationships in referential semantics is another central drawback of this representation. So referential semantics lead to a rather counter-intuitive realisation of relationships and exclude the representation of relationship attributes. The following subsections introduce the different types of relationships as defined in the UML metamodel. Figure 24, p. 71 depicts the relevant metaclass in the UML metamodel:

Associations: n-ary Relationships

The relational semantics of UML relationships bypasses the shortcomings of referential semantics through the introduction of the metaclasses `Association` and `AssociationClass`. Relationships may have a unique name, since metaclass `Association` is a specialisation of metaclass `ModelElement`. Separating `Associations` from `AssociationEnds` also guarantees a flexible handling of n-ary relationships. A relationship is therefore characterised by its name, while instances of metaclass `AssociationEnd` maintain references to the linked classes through the relationship `type`. Since `AssociationEnd` is a specialisation of metaclass `ModelElement`, all references may carry an individual name, the *association role* [OMG97b, p. 50]. Association roles especially facilitate reading recursive relationships. Cardinalities restrict the amount of participating objects in a relationship at run-time, realised through the property `multiplicity` of metaclass `AssociationEnd`. A range of integers form the upper and lower bound of participating instances.

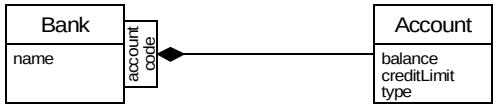
Qualifiers & AssociationEnds

Qualifiers are an interesting UML construct defined as follows:

> "An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association." [OMG97a, p. 156]

In Figure 25, the qualifier `accountCode` relates accounts to the bank maintaining the account. In this sense the qualifier in UML can be compared to keys in the relational data model. Generally speaking, qualifiers may be used in the sense of compound keys, i.e. they represent additional information for the traversal of relationships. Although object-orientation is based on the idea of an invariant identity, real-world situations like the bank example demand additional identification mechanisms for related class specifications. We therefore consider qualifiers as a suitable modelling construct for *component schemas*.

Figure 25 Example of a Qualified Association [RBP+91]



Additional Properties of Metaclass AssociationEnd

Besides qualifiers, the metaclass `AssociationEnd` comprises a variety of additional properties. The property `isNavigable` is directly related to the referential semantics of relationships in object-oriented programming languages. When the property is set to the value `true`, a traversal from the source end to target end of the relationship is prohibited, i.e. there is no inverse pointer. This means the objects may not mutually use their operations. Another design-related construct has been realised in the property `changeable`. This property determines whether links may be added or changed at run-time. The availability of metaclasses in a programming language is the central idea behind the property `targetScope`. In Smalltalk, the association between classes and objects has to be distinguished from links between individual objects. Imposing an order on a set of links, may be specified through the property `isOrdered`. We also consider this UML construct in *component schemas*. The semantics of the property aggregation are detailed in the context of semantic relationships on p. 76.

Relationship Attributes in UML

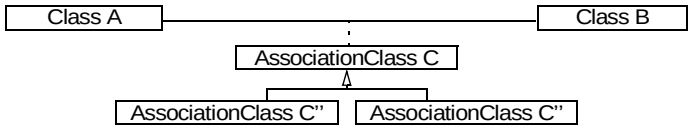
Relationship attributes are realised through the metaclass `AssociationClass`, as a specialisation of the metaclasses `Association` and `Class`. Its generalisation `Association` guarantees maintaining references to classes, while properties may be attached to the relationship via its generalisation `Class`. In other words, relationships in the UML metamodel are entities in their own rights, maintaining an arbitrary amount of links to other classes. The number of participating objects in a relationship may be constrained via cardinalities. Finally, qualifiers allow handling of complex identification schemas for participating objects and also realise compound keys.

Association & AssociationClass Revisited

However, the concrete realisation of relationships in UML is quite surprising, since metaclass `Association` is a *specialisation* of metaclass `GeneralizableElement`. This may imply that relationships can be specialised in UML. There is definitely no serious interpretation assuming regular set inclusion semantics for specialisation. Moreover only association classes carry attributes, so specialisation does not further restrict objects participating in a relationship.

Association classes, on the other hand, add properties to relationships and exactly these properties may be specialised. Sharing properties seems to be the central motivation for the specialisation of relationships. So we may assume that the authors of the metamodel had the following idea in mind.

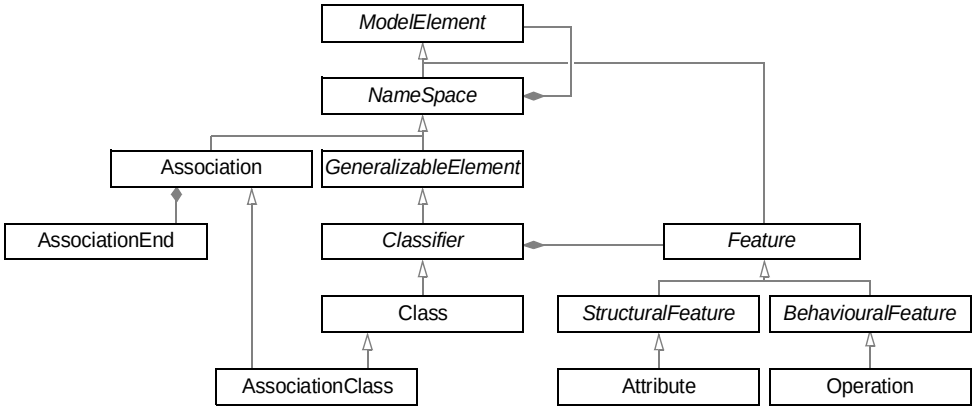
Figure 26 Specialisation Example for AssociationClasses



Only Figure 26 provides a reasonable explanation for regarding relationships as generalisable elements. Association classes may carry behavioural as well as structural properties.

So differing behaviour or additional attributes may distinguish specialisations of an association class. But this possibility is already guaranteed in the metamodel through the generalisation hierarchy of metaclass `Class`. Placing metaclass `Association` on the same specialisation level with metaclass `GeneralisableElement` might avoid this anomaly. A potential improvement of the metamodel is depicted in Figure 27.

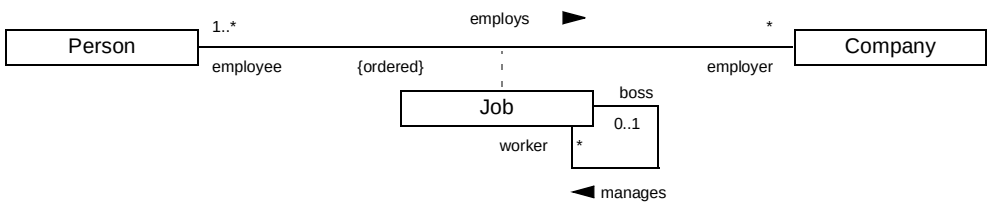
Figure 27 Improved Metamodel for Associations and AssociationClasses



AssociationClass Implementation

Rational Rose 98 [Rat98], as the reference CASE tool for UML, does not follow the metamodel in the implementation of association classes. It creates an individual instance of metaclass `Class` carrying the properties. Then this class specification has to be linked to an instance of metaclass `Association` via a distinct relationship. Rational Rose also maintains the individual names of the association and the class specification carrying the properties. This representation clearly violates the definition in the metamodel document [OMG97a]. Figure 28 depicts the complete relationship notation used in *component schemas*.

Figure 28 Selected Notation for Associations in Component Schemas



Section Summary

Associations in UML have tuple semantics and allow flexible and uniform modelling of relationships with an arbitrary arity. However modelling associations as generalisable elements is a rather counter-intuitive construction. Restricting navigation or changes to connections are design elements which will not be supported in this work. Associations at runtime never link classes and always connect objects. Association

ends indicate the cardinality and possibly impose an ordering constraint on the relationships. The reading direction of associations may be supported by the so-called *name-direction arrow*, as notational element not yet covered in the metamodel. Associations may link more than two classes, representing so-called n-ary relationships. Finally, associations may be qualified.

Generalization and Composition: Semantic Relationships

Semantic relationships are the most important contribution to the expressive power of object models. From a naive perspective, semantic relationships can be regarded as binary relationships with extended integrity constraints. N-ary and semantic relationships have not been treated in a uniform way in the metamodel, i.e. they are not specialisations of metaclass `Association`. Unlike `Generalization`, the metamodel does not contain a direct counterpart for aggregation, i.e. there is no particular metaclass called aggregation. The semantics document defines generalisation as follows:

> "A *generalization* is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information." [OMG97a, p. 24]

Generalisation Criteria

A `Generalization` leads to an instance of metaclass `GeneralizableElement`, maintaining references to instances of metaclass `Class`. Generalisations in UML are not bound to a given name space, since the metaclasses `NameSpace` and `GeneralizableElement` share the same specialisation level. However, the well-formedness rules require that sub- and supertype must reside in the same name space [OMG97a, p. 31]. Another interesting aspect of metaclass `Generalization` can be found in the property `discriminator`. According to the semantics document, the `discriminator` builds partitions on the set of generalisation links. This corresponds to the idea of generalisation criteria proposed by Schienmann [Sch97]. A set of real-world entities can be categorised according to a variety of criteria, depending on the analyst's perspective. The `discriminator` captures the individual criteria chosen by the analysts. Unlike Schienmann's proposal, UML allows an assignment several discriminators on a given specialisation level.

Restricting Generalisation Hierarchies

Metaclass `GeneralizableElement` carries a set of properties dealing with generalisations hierarchies. When the properties `isRoot` and `isLeaf` are set, given classes may not be further refined [OMG97a]. Furthermore, classes must not participate in generalisation hierarchies at all, when property `isRoot` is set. These restrictions do not suit requirements engineering, since problem analysis is likely to identify structural and/or behavioural similarities between real-world things, that may be represented in terms of a generalisation. As a consequence, these properties will not be regarded for *component schemas*.

Abstract Classes

The property `isAbstract` indicates that a given class specification cannot be instantiated directly. This construct mainly introduces a uniform treatment of related real-world things in problem analysis. According to our modelling experience, this adornment is often used when similar structural as well as behavioural features of classes are combined into a generalisation. The specialised classes directly represent real-world things in the application domain, while the generalisation expresses the similarity between them. It does not lead to instances of their own right, since there is no direct counterpart in reality. However, building abstract classes can be considered as design-related, since code sharing is the central concern for this activity. On the other hand, introducing abstract classes is one of the important assets of problem analysis. From an analysis perspective, concepts previously considered as independent will then be related to each other. Although this construct has a considerable significance in design, it is equally valid in problem analysis and therefore will be incorporated in *component schemas*. According to the Notation Guide [OMG97b], class names written in italics distinguish abstract classes from concrete ones.

Aggregation versus Composition

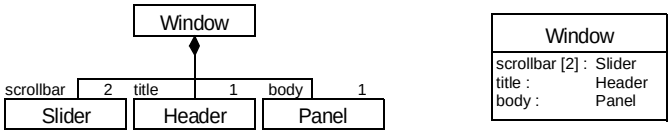
Unlike `Generalization`, the metamodel does not contain a direct counterpart for aggregation, i.e. a metaclass called aggregation. Furthermore, UML distinguishes in between two different forms of aggregation: *aggregation* and *composition*. While aggregation does not impose constraints on the life-time between whole and part, composition relationships express a coinciding life-time of both whole *and* part. In other words, parts in a composition have to be destroyed when the life-time of the whole ends. Furthermore, parts in a composition must not be shared, i.e. an instance of a part may not participate in several aggregates. Concrete life-time considerations with respect to whole and parts have to be ignored in requirements engineering.

Composition in the Metamodel

As mentioned above UML does not feature compositions as individual metaclasses. Furthermore, compositions are defined as adornments of metaclass `AssociationEnd` in terms of the property `aggregation`. The semantic document states:

> "When placed on a target end, specifies whether the target end is an aggregation with respect to the source end. Only one end can be an aggregation." [OMG97a, p. 18]

Figure 29 Different Notations for Compositions in UML



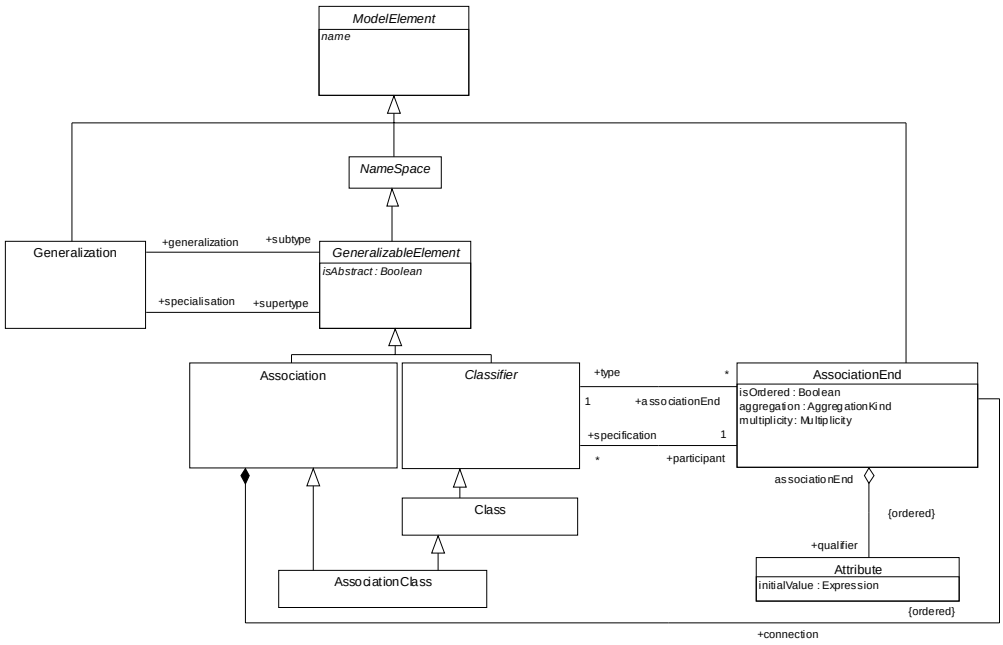
However, this essential consistency constraint has not been covered in the well-formedness rules! This and other deficiencies have been discussed in detail in Ute Schirmer's diploma thesis [Schi98]. Generally speaking, n-ary relationships and compositions are only distinguished by adornments on the `AssociationEnds`. Composition, like associations, may carry a unique name. The Notation Guide presents the following example for alternate visualisation of compositions with role

names [OMG97b, p. 64]. Obviously the second notation on the right hand side of Figure 29, p. 76 corresponds to array declarations in C++. The notation once again points to the close similarity between class and type declarations, as discussed on p. 67.

Section Summary

The restrictions with respect to the participation of classes in generalisation hierarchies will not be considered for *component schemas*. Composition has been selected as the relevant construct for the representation of aggregation *component schemas*. Discriminators for capturing the generalisation criteria and role names for compositions will be regarded as comments. Finally, abstract classes are deemed relevant for requirements engineering.

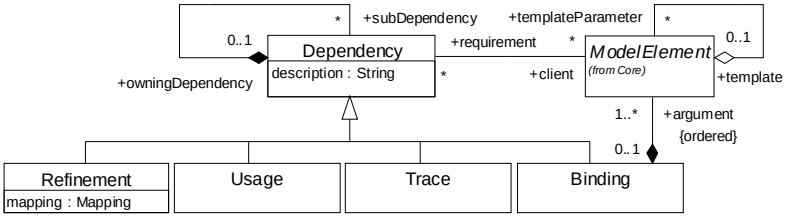
Figure 30 Revised Metamodel for Associations in Component Schemas



Dependencies and Traces

Besides n-ary and semantic relationships, UML offers a set of additional relationships not covered in traditional semantic data models. Based on the metaclass `Dependency` defined in the package `Auxiliary Elements` UML introduces a set of specialisations detailing particular constructs for object-oriented analysis and design. Figure 31 depicts the package `Auxiliary Elements`, including additional relationships from the package `Core`.

Figure 31 Dependencies and Traces in the UML Package `Auxiliary Elements` [UML1, p. 43]



The starting point for this discussion is metaclass `Dependency`. As a specialisation of metaclass `ModelElement`, dependencies have a name. Dependencies are not bound to a specific name space, since its metaclass resides on the same specialisation level as metaclass `NameSpace`. The semantics document states:

> "A *dependency* states that the implementation or functioning of one or more elements require the presence of one or more other elements. All of the elements must exist at the same level of meaning, i.e. they do not involve a shift in the level of abstraction or realization.
>
> In the metamodel a ***Dependency*** is a directed relationship from a client (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier, i.e. the client element requires the presence and knowledge of the supplier element." [OMG97a, p. 22]

The property `description` explains the nature of the dependency in a plain-text fashion. The relationship `client` depends on the presence of an arbitrary number of `ModelElement` instances and the `supplier` relationship denotes the inverse direction [OMG97a, p. 22]. Dependencies in UML originate from uses associations in the Booch method. A dependency between two classes indicates that one class, the *client*, relies on operations provided by a *supplier* [Boo94]. But dependencies may also be used to indicate a dependency among packages. Since *component schemas* exclusively model behavioural aspects in sequence diagrams, dependencies are not applicable to classes. This restriction improves the orthogonality of the selected diagrams in *component schemas* and rigorously separates invariant from dynamic aspects.

Categorising Dependencies

The recursive aggregation `subDependencies` allows further categorisation of dependencies. The semantic document states:

> "... a dependency can serve as a container for a group of Dependencies. This is useful, because often dependencies are between of groups elements (such as Packages, Models, Classifiers, etc.). For example, the dependency of one package on another can be expanded into a set of dependencies among elements within the two packages." [OMG97a, p. 45]

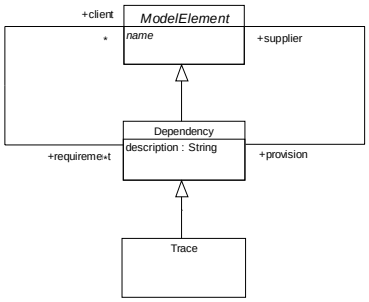
The relationships `templateParameter` and `argument` combined with metaclass `Binding` are related to the C++ templates and can therefore be discarded in *component schemas*. Metaclass `Refinement` represents a dependency between model elements on different levels of detail. Analysis classes may be distinguished from design entities. So this construct will also be discarded in *component schemas*.

Since Metaclass `Usage` guarantees the compatibility to the Booch Method, we also leave out this construct. Finally, metaclass `Trace` introduces a rather interesting construct. The metamodel states:

> "A *trace* is a conceptual connection between two elements or sets of elements that represent a single concept at different semantic levels or from different points of view. However, there is no specific mapping between the elements. [...]
> The construct is mainly a tool for tracing requirements." [OMG97a, p. 47]

As a specialisation of metaclass `Dependency`, the trace relationship may link arbitrary UML `ModelElements`. As detailed in Section 7.4, the UML trace relationship provides an ideal basis for incorporating *component schema relationships* into the metamodel.

Figure 32 Revised Metamodel for Dependencies in Component Schemas



5.2.3.3 Packages & Models

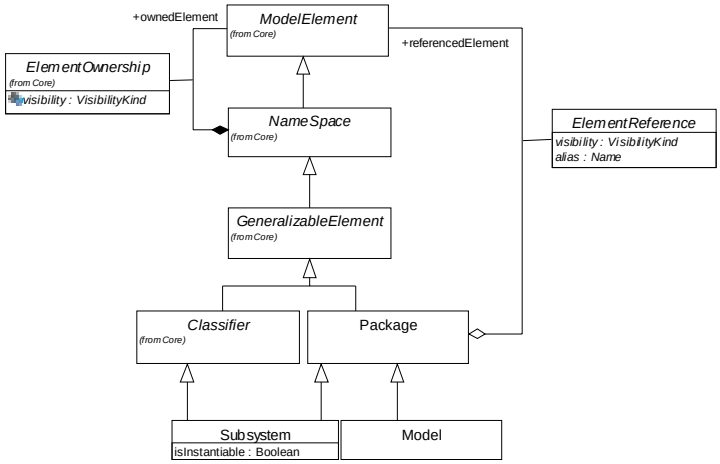
Large-scale specifications in UML can be organised into packages. This mechanism has also been applied for dividing the entire UML metamodel into comprehensive partitions. The Semantics Document states:

> "A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages. A system may be thought of as a single high-level package, with everything else in the system contained in it." [OMG97a, p. 155]

Packages, as specialisations of metaclass `ModelElement`, have a unique name and logically group its contained elements. Although this grouping entirely depends on the analyst's perception, packages may represent a coherent portion of a given application domain. Despite the fact that metaclass `Package` does not carry properties of its own, the intended grouping is realised through inherited relationships. Metaclass `NameSpace`, as a generalisation of metaclass `Package`, relates the contained classes through the relationship `ElementOwnership`. Its property `visibility` indicates whether an element contained in package may or may not be visible to other

packages. Hiding elements of a specification restrains the detection of correspondences, as a central prerequisite for requirements validation. Thus *component schemas* do not restrict the visibility of package contents.

Figure 33 Packages in the UML Model Management Package [OMG97a, p. 129]



All model elements comprised in a package have a unique name within the entire specification. As a specialisation of `NameSpace`, all the names of all contents of a given package are prefixed with the package name. Package contents may maintain relationships to model elements in other packages, realised through the relationship `ElementReference`. The property `visibility` will be discarded in *component schemas*, in order to keep all package contents visible. The property `alias` defines a local name within a given package for referenced external elements.

Metaclass Subsystem

Metaclass `Subsystem` is another interesting specialisation of metaclass `Package` and contains model elements such as classes, associations or even packages. Rumbaugh states in the UML Reference Manual:

> "A subsystem is a grouping of model elements, of which some constitute a specification of the behaviour offered by the contained model elements." [OMG97a, p. 158]

But unlike packages, subsystems stress behavioural abstractions of its contents. As a specialisation of metaclass `Classifier`, subsystems principally carry behavioural as well as structural features. Well-formedness rules, however, exclude structural features. This means subsystems can be regarded as modules in modular programming languages such as Modula-2. This hypothesis will further be underpinned through the property `isInstantiable` in metaclass `Subsystem`. UML classifies the subsystem contents into *specification* and *realisation* elements.

> "The specification of a subsystem consists of the specification subset of the contents together with subsystem's *features* (operations). It specifies the behavior performed jointly by instances of the classifiers in the realization subset, without revealing anything about the contents of this subset." [OMG97a, p. 135]

Subsystems as Facade Classes

This definition corresponds to the idea of *facade classes*, in the sense of Gamma [GHJ+95]. Facade classes provide a uniform interface to a set of related classes jointly performing a defined functionality. They introduce layers to otherwise "flat" object models and hide a potentially complex interaction of the underlying classes. Subsystem clients may rely on a narrow and loosely coupled interface, facilitating reuse and customization. The semantic document continues the definition as follows:

> "The specification is made in terms of use cases and/or operations, where use cases are used to specify complete sequences performed by the subsystem (i.e. instances of its contents) interacting with its surroundings, while operations only specify fragments. Furthermore, the specification part of a subsystem also includes constraints, relationships between the use cases, etc." [OMG97a, p. 135]

Subsystems as Control Objects

Relating use cases to subsystems corresponds to Jacobson's idea of *control objects* [JCJ+94], which comprise behaviour that cannot naturally be related to individual domain classes. While *facade classes* stress layering of classes, *control objects* incorporate control flow of an object-oriented application, derived from use cases and represented in terms of sequence diagrams. Gamma's definition can be regarded as design-related, since domain classes are hierarchically structured. Opposed to this objective, Jacobson deems control objects as analysis constructs incorporating an abstraction of the control flow among the set of interacting objects of a use case.

Although we follow Jacobson's argument and interpret subsystems as control objects in this work, we still have to exclude subsystems from further consideration. The major reason for this decision is the lack of subsystem support in off-the-shelf CASE tools like Rational Rose.

Metaclass Model

While packages logically group its contents, a complete specification in UML is organised in the metaclass `Model`. The semantics document states:

> "A *model* is an abstraction of a modeled system, specifying the modeled system from a certain viewpoint and a certain level of abstraction. A model in the sense that it fully describes the whole modeled system at the chosen level of abstraction and viewpoint." [OMG97a, p. 130]

Packages, Models & Component Schemas

Unlike its generalisation `Package`, metaclass `Model` does not have a visual representation in the notation. In other words, a model is the top-level package, describing the entire system including its environment. This representation is quite advantageous for *component schemas*. In concurrent requirements engineering, a *component schema* corresponds to an instance of metaclass `Model`. For the integration of the entire problem description, all *component schemas* have to be integrated into a new instance of metaclass `Model`. The *component schemas* then have to be converted into `Package` instances. As a positive side effect, the logical grouping of *component*

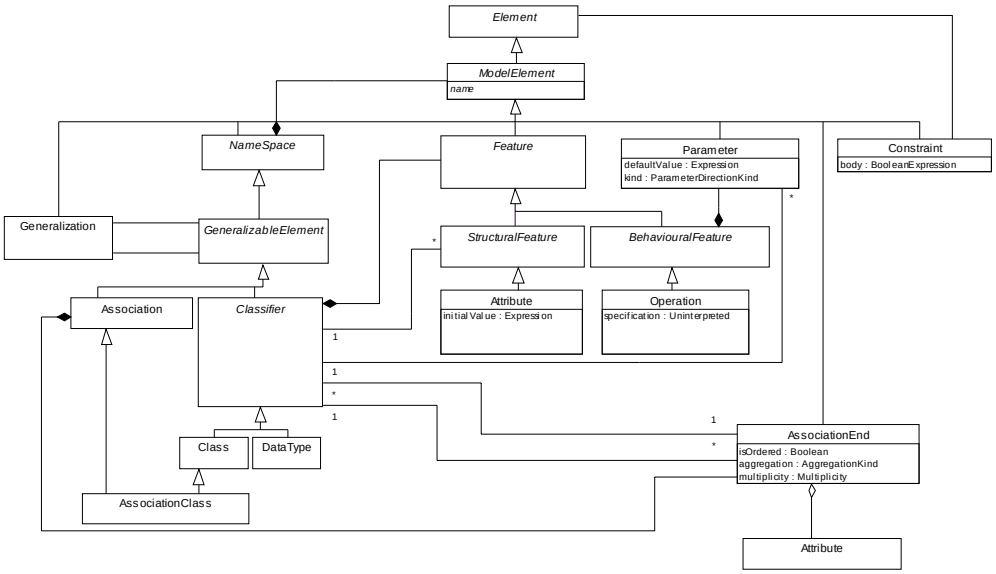
schemas can be preserved in the complete problem description, increasing the traceability of the problem description. The name space concept in UML also avoids name clashes for package contents originating from different *component schemas*.

Dependency and Packages

As mentioned in Section 5.2.3.2, dependencies in UML originate from *uses associations* in the Booch method and can be classified into two groups with slightly differing semantics [Boo94]. According to Booch, a dependency between two classes denotes that one class – the client – relies on operations provided by a supplier. This matches the definition in the UML semantics. Dependencies are also applied to denote provision of services among classes in different packages. Since *component schemas* exclusively model behavioural aspects in UML sequence diagrams, dependencies are not applicable to classes.

There are basically two different possibilities to restrict the uniform treatment of classes and packages for dependencies in *component schemas*. The first solution may formulate a new OCL constraint. This idea corresponds to the well-formedness rules in the semantics document. Another way is enhancing the metamodel itself, i.e moving the relationships `client` and `supplier` from metaclass `ModelElement` to metaclass `Packages` (see Fig. 34). Both solutions have the potential to clarify *component schema* semantics. Nevertheless, we prefer additional well-formedness rules. Maintaining structural compatibility compared to the UML metamodel will in the long term guarantee the compatibility of *component schemas* to CASE tool implementations. Avoiding dependencies among classes and concentrating on a few comprehensive and well-chosen modelling constructs was the central motivation for deferring the discussion of the dependency association.

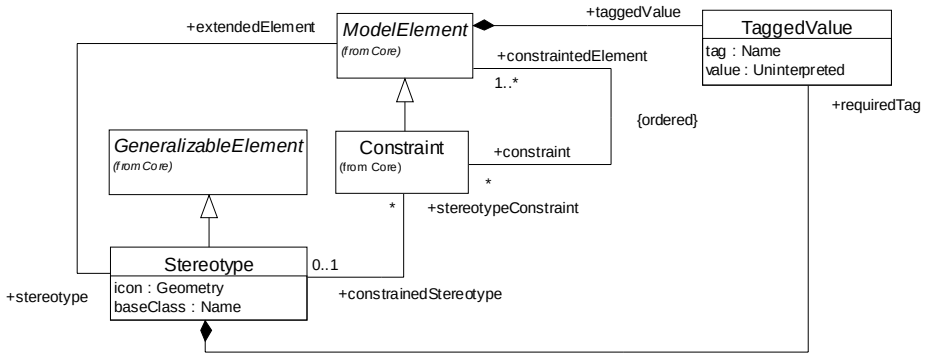
Figure 34 Revised Metamodel for Classes and Associations in Component Schemas



5.2.3.4 Stereotypes

As introduced in Section 5.2.3.1, stereotypes allow a categorization of metamodel elements at instance level. This approach enables a value-based categorisation of metamodel elements, which cannot be achieved by the rather inflexible specialisation. Moreover, metamodel elements can be further classified by CASE tool users at runtime. So analysts may introduce their individual classification scheme for *component schema* contents. Metaclass `Stereotype` is bound to a name space and can be applied to all metamodel elements. The classification itself, however, is realised through the metaclass `TaggedValue`, carrying the classification criteria. Figure 35 depicts the relevance of the UML metamodel:

Figure 35 UML Package Extension Mechanism



Originally stemming from Jacobson's [JCJ+94] distinction of *entity*, *control* and *interface* classes, UML extends this user-defined classification mechanism for model elements significantly. Stereotypes also provide a suitable basis for the classification of different *component schema relationships*, as detailed in Chapter 7. Therefore stereotypes will be incorporated in *component schemas*.

5.2.4 Use Cases

Use cases are the central means for the elicitation of requirements in this work. Relating constructs of the regarded UML views to use case's plain-text descriptions provides multiple access to complex requirements. Based on [RAB96] and [Rum94] we proposed the following template for use cases. The fields `Author`, `References` and `Open Questions` are intended to improve document source traceability and clarify open questions, when use cases have been written by analysts instead of domain experts.

The field `Description` contains the course of activities in the analysed business process. Enumerating the individual activities in the `Description` field improved clarity and readability. The enumeration also increases the traceability between a use

case and its derived sequence diagram. A complete reference for this structure including the results of an analysis project in the air traffic management domain can be found in [Wor96].

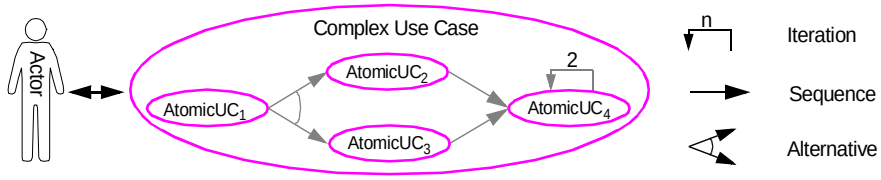
Figure 36 Template for Complex Use Cases

`Use Case <No>: <Name>`	A sequential number and a unique name for use cases
`Actors:`	Involved Actors
`Summary:`	Summarising the objective / purpose of the use case
`Precondition:`	Conditions for the invocation of this use case
`Description:`	List of activities performed in the business process
`Error Condition:`	Potential error conditions
`Postcondition:`	Constraints that have to be met at the end of a use case
`Author:`	For traceability reasons
`References:`	Traceability of information sources - referenced documents
`Open Questions:`	Identifying vagueness, ambiguity, omissions in the use case

5.2.4.1 Complex Use Cases: A Pragmatic Composition Mechanism for Business Processes

Complex business processes, especially in application domains where control [Dav93] is the most prominent external behaviour, are often difficult to describe with use cases in their original form [JCJ+94]. *Extends* and *uses* associations among use cases provide a very low degree of structure for complex business processes. In our air traffic management domain, the analysed business process contained small groups of activities that appeared in a variety of use cases, always in different orders. Consequently we created a separate use case in its own right for these recurring groups of activities. Nevertheless, the lack of a clear composition mechanism prevented the combination of these newly created use cases into complex use cases.

Figure 37 Notation for the Composition of Complex Use Cases Based on Path Expressions



Path expressions [CH74] provide a pragmatic formalism for the construction of complex use cases and therefore solve the described deficiency. Sequences, alternatives and iterations now form complex business processes on the basis of so-called *atomic use cases*. The same mechanism can also be applied for the decomposition of complex business processes, improving the scalability of plain-text descriptions dramatically. Incorporating path expression to use cases leads to the new field `Assembly` in the use case template (see Figure 36) and an extended notation for use case diagrams, depicted in Figure 37. The `Assembly` field is only specified in complex use cases

and defines the order of atomic use cases forming the entire business process. Due to their plain text character, we did not foster the idea of a proper formalisation of complex use cases. Further details can be found in [Wor96].

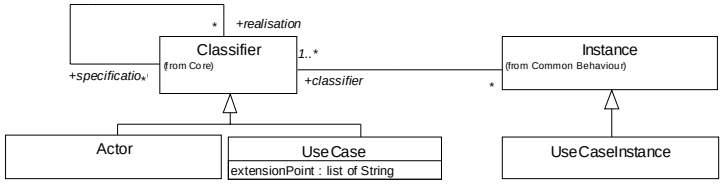
Component Schema Contents

Component schemas are partial UML specifications, modelling an *initial partition*. They comprise a set of plain text use cases describing all relevant business processes within the analysed partition. Use cases follow a defined structure and can be further decomposed. Complex use cases can be constructed from a set of atomic use cases, where a path expression specifies their order. Each atomic use case will be structured in a particular sequence diagram, modelling the business process in terms of objects exchanging messages. Structural aspects mentioned in a use case will be modelled in a static structure diagram, defining classes, attributes and associations. A detailed discussion of relevant *constructs* for *component schemas* including their notation is introduced in the next section.

5.2.4.2 A Metamodel for Complex Use Cases

The UML metamodel [OMG97a] classifies use cases as behavioural elements and for convenience sake defines an individual package named `Use Cases`. Figure 38 depicts the relevant metaclasses:

Figure 38 UML Package Use Cases



According to the metamodel, use cases define system behaviour without revealing its internal structure. They specify a sequence of activities the system has to perform in interacting with system users. In order to clarify semantics of use cases, Figure 39 depicts the amalgamation of the UML packages `Core` and `Use Cases`.

Use Cases & Actors

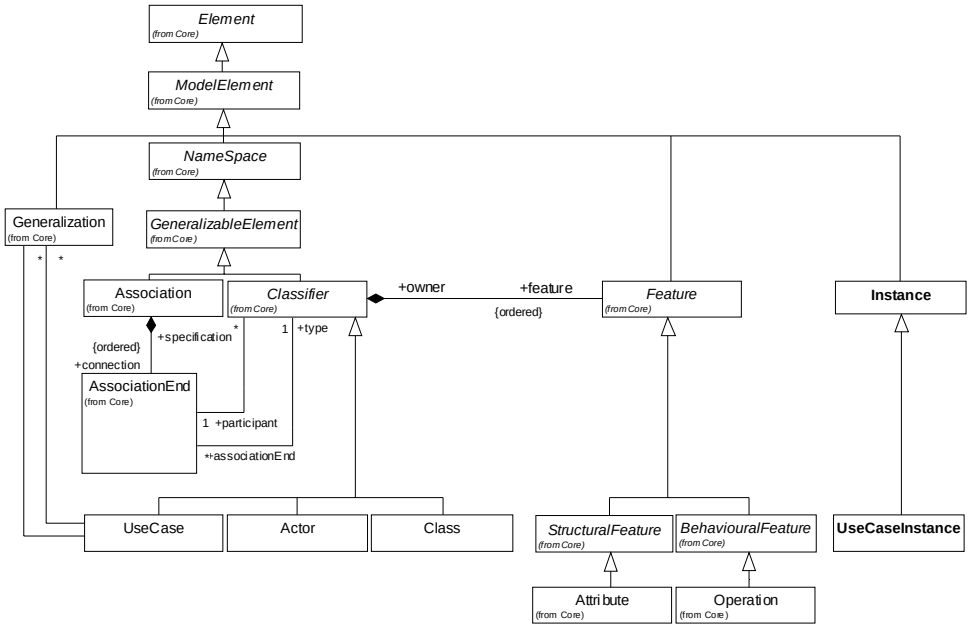
Use cases as well as actors are specialisations of metaclass `Classifier` and therefore carry a name and may be characterised by a set of behavioural and structural features. This approach leads to a counterintuitive situation, since use cases as well as actors may be characterized by properties. The metamodel document states:

> "An *actor* defines a coherent set of roles that users of an entity[1] can play when interacting with the entity. An actor has one role for each use case with which it communicates"
> [OMG97a, p. 90]

Use Cases, Classes and their Runtime Counterparts

Describing system users with properties thus contradicts the analyst's perception. The external character of system users who are not an integral part of the system at hand has not been considered. A similar argument holds for use cases themselves. Relating properties to use cases leads to a conceptual overload in the sense of Wand & Weber [WW93], since only the metaclasses `Class` and `DataType` (see Fig. 34, p. 82) may comprise properties. From this perspective use cases can be regarded as the same construct. Use cases in the UML metamodel are not only characterised as specification time entities, they also have runtime counterparts, represented in the metaclasses `Instance` and its specialisation `UseCaseInstance`.

Figure 39 UML Package `Use Cases` and Related Metamodel Elements from the Package `Core`



The motivation behind this representation can once again be found in Jacobson's idea of control objects [JCJ+94]. Behaviour not naturally bound to individual objects should be bound to control objects, which can be regarded as an object-oriented representation of control flow in the sense of imperative programming languages. Use cases specify a sequence of activities the system has to perform during an interaction with business users. These activities are structured in a sequence diagram and are represented in control objects in design. Unlike Jacobson, UML does not distinguish between control and entity objects on a metamodel level. The metaclass `Instance` realises both constructs at runtime, entity classes as well as control objects. In order to assure uniform treatment of control and entity objects at runtime, the metamodel

1 Entity in the sense of system at hand

represents metaclass `UseCase` as a specialisation of metaclass `Classifier`. Thus use cases like classes may be characterised by behavioural, as well as structural properties. This discussion is detailed below after the introduction of use case relationships.

Use Case Relationships

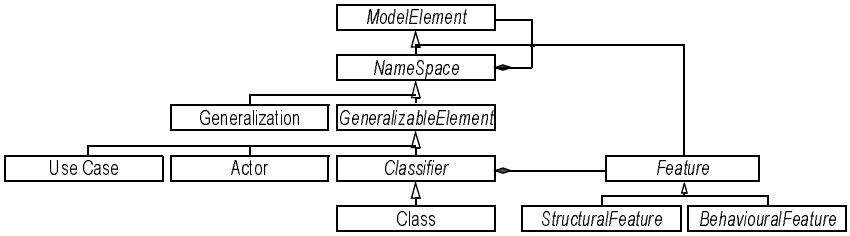
UML foresees two dedicated use case relationships, so called *uses* and *extends* relationships. The semantics document states:

> "**extends**
> A relationship from a use case to another, specifying how the behaviour defined for the first use case can be inserted into the behavior defined for the second use case" [OMG97a, p. 152]

> "**uses**
> A relationship from a use case to another use case in which the behavior defined for the former use case employs the behavior defined for the latter." [OMG97a, p. 160]

The uses relationship can be regarded as a specialisation of use cases, although due to the textual character of use cases there are no precise semantics. The metamodel realises both relationships as stereotyped instances of metaclass `Generalization`. The property `extensionPoint` of metaclass `UseCase` marks the location within a use case where the additional behaviour of an *extended* use case comes in.

Figure 40 Decoupling Classes, Use Cases and Actors in the Metamodel

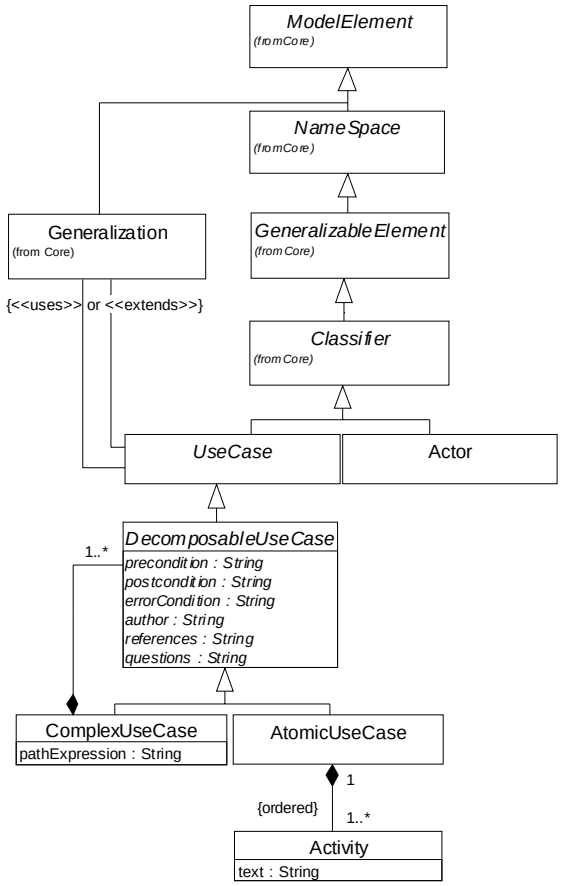


Deficiencies of the UML Use Case Representation

The UML metamodel stresses two aspects of use cases which do not suit requirements engineering. Use cases are basically treated as diagrams in UML, which are detailed as a sequence diagram. In other words, use cases are only an iconic representation of the underlying sequence diagram. They carry a unique name for the modelled business process, depicting the involved actors as well as related use cases. The textual description of the modelled business processes will not be incorporated to the repository of the CASE tool implementation. As a result, the most comprehensive part of the specification with respect to non-IT stakeholders remains completely outside the scope of UML. Treating actors and use cases similar to class specifications leads to the second problematic aspect with respect to requirements engineering. Both constructs, actors as well as use cases, are represented as discrete elements in a given model, i.e. they can be mapped to class specifications in the code generation of CASE tools. This kind of design-related representation suits code generation, but also

leads to a concept overload in the sense of Wand & Weber [WW93], which reduces clarity of the modelling technique and hinders understanding of the semantics by stakeholders without extensive IT background.

Figure 41 Revised Metamodel for Complex Use Cases in Component Schemas



In order to overcome the analysed deficiencies and incorporate the pragmatic decomposition mechanism, as described in Section 5.2.4.1, the metamodel has to be enhanced. Decoupling use cases at specification time from control objects at runtime can be realised through changes to the specialisation hierarchy in the UML package `Use Cases`. Placing the metaclasses `Actor` and `UseCase` on the same specialisation level as metaclass `GeneralizableElement` prevents relating properties to these constructs (see Figure 40, p. 87). A redefinition of the metamodel, however, threatens the compatibility to CASE tool implementations and therefore will be discarded. On the other hand, the proposed decomposition mechanism in Section 5.2.4.1, enforces those changes. Nevertheless, changes to the metamodel always have an impact both on the structure of models *and* on instances in terms of the ISO IRDS [ISO10027].

Use Case Representation

In order to keep the impact of metamodel changes minimal we define a new metaclass `DecomposableUseCase`, which specialises the existing UML metaclass `UseCase`. It introduces the additional structural properties to the metamodel, with the attributes `precondition`, `postcondition`, `errorCondition`, `author`, `references` and `openQuestions`. The UML composition between the metaclasses `ComplexUseCase` and `DecomposableUseCase` realises the decomposition mechanism. Instances of metaclass `ComplexUseCase` may contain arbitrary collections of decomposed use cases, represented through metaclass `AtomicUseCase`. The order of use cases within an instance of metaclass `ComplexUseCase` is specified in its property `pathExpression`. A formal interpretation of this path expression is not provided. Finally, metaclass `Activity` comprises the textual descriptions of the described business process in the field description of the use case template.

Incorporating Use Case Activities to the Metamodel

Realising individual statements in the use case's plain-text description as a metaclass in its own right leads to improved traceability among *component schemas* and use cases. Unstructured textual information represented through instances of the metaclass `Activity`[1] can now be related to derived representations in sequence and static structure diagrams. First of all, the use case activities become part of the problem analysis model itself after which, they can be related to all derived representations in *component schemas*. The enhanced metamodel for *component schemas* is depicted in Figure 41.

5.2.5 Sequence Diagrams

UML proposes two different notations for object interaction, documenting behavioural requirements in *component schemas*. Both diagrams depict examples interactions of the system with its users and may be used to structure the textual representation of use cases in terms of objects exchanging messages. Consequently, each use case will be structured in terms of either a *sequence* or a collaboration *diagram* in *component schemas*. Sequence diagrams depict inter-object behaviour in a time sequence, while collaboration diagrams depict the relationship among objects without incorporating time as separate dimension. In order to clarify the distinction we are going to present an example using both notations.

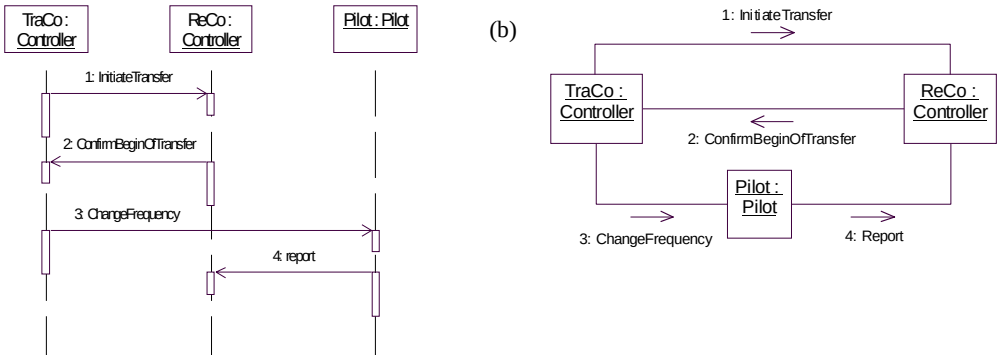
Sequence Diagrams Suit Interpretation of Use Cases

We prefer sequence diagrams for the purposes of object-oriented interpretation of activities in a use case. Textual activities in a given use case will be represented as objects exchanging messages and the course of use case activities directly corresponds to the course of messages in the sequence diagram. Especially the direct correspondence between the course of activities in a use case and the course of messages exchanged among objects enhances the comprehensibility of the documented

1 Whenever we refer to this new metamodel element, we are going to write the term *activities* in italics

requirements to business users. The combination of use case and sequence diagrams also provides a comprehensive and traceable justification for operations specified for a given class specification in the derived static structure diagram. Although the course of messages is indicated through a number, collaboration diagrams are regarded as less comprehensible in practice. In the remainder of this work we concentrate on UML sequence diagrams, specified in the UML metamodel package `Collaborations`.

Figure 42 Examples of UML Sequence (a) and Collaboration (b) Diagrams



Sequence Diagrams Semantics

The semantics for sequence and collaboration diagrams are both defined in the UML package `Collaborations`, documenting once again the close relationship between the two modelling techniques. The concrete mapping of the semantics to the two alternate modelling techniques, however, is described in the UML Notation Guide [OMG97b], so the discussion will incorporate both the semantics and the notations document.

Sequence Diagrams form the final building block of *component schemas* in this work. They structure the textual descriptions of individual *activities* described in a given use case in terms of objects exchanging messages. As a specialisation of metaclass `NameSpace` collaborations have a unique name within a name space. Metaclass `ClassifierRole` represents prototypical objects participating in a given collaboration. The objects are called prototypical in order to avoid the runtime character of the term object. This runtime character is formed by the intension / extension dichotomy, where objects are generally regarded as instances of a given class. The semantics document states:

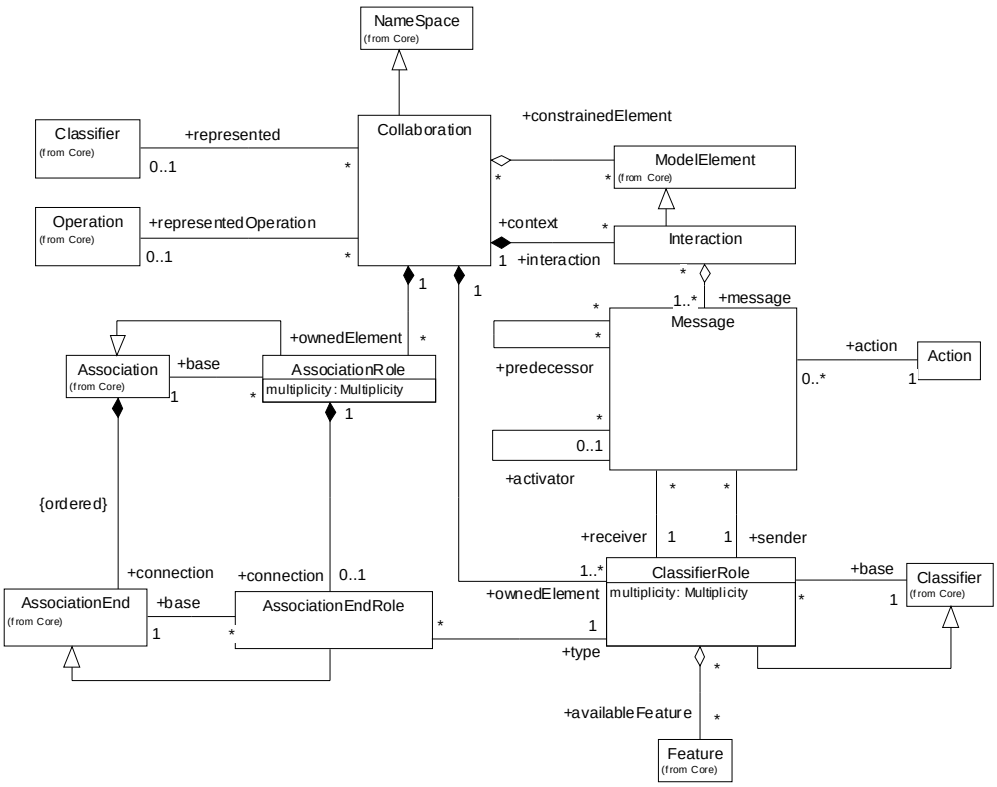
> "**object**
> An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class. ..." [OMG97a, p. 155]

ClassifierRoles vs. Prototypical Objects

As a specialisation of metaclass `Classifier`, metaclass `ClassifierRole` may contain behavioural as well as structural features and is based on class specifications, which is expressed through the relationship `base`. As a result, instances of the meta-

class `ClassifierRole` may comprise properties of their own, i.e. which are currently not defined for the generalised instance of metaclass `Classifier`, on which this instance is based. So each instance of metaclass `ClassifierRole` depicts a restricted view of a classifier in a given sequence diagram. This restriction is expressed through the aggregation `availableFeature` between the metaclasses `ClassifierRole` and `Feature`. In principle, assigning individual features to `ClassifierRoles` also allows the specification of OOram role models [Ree96], as described in Section 3.3.4.2.

Figure 43 UML Package Collaborations [OMG97a, p. 81]



The current version of the metamodel, however, does not promote this possibility through the following well-formedness rules:

"**ClassifierRole**

[2] The ***Features*** of the ***ClassifierRole*** must be a subset of the base ***Classifier***

```
self.base.allFeatures -> includesAll (self.availableFeature)
```

[3] A ***ClassifierRole*** does not have any ***Features*** of its own.

```
self.allFeatures -> isEmpty
```
" [OMG97a, p. 84]

Only existing `Features` of a `Classifier` can be used in a `ClassifierRole`, just the opposite way round to OOram role model collaboration, where only those properties relevant in a given collaboration are specified and lead to a definition of an OOram type. Role model synthesis finally maps types to class specifications, i.e. defining all properties for a class specification in terms of a given programming language. As we will see, this approach in the UML metamodel is basically related to structural features, which are irrelevant in sequence diagrams, where the primary concern is modelling behavioural requirements.

The property `multiplicity` of metaclass `ClassifierRole` depicts the number of instances playing this role in a given sequence diagram. Several prototypical objects (metaclass `ClassifierRole`) based on the same class specification (metaclass `Classifier`) may participate in a sequence diagram. The usage of the property `multiplicity` is not clearly defined in the semantics document and we assume it is intended for accounting purposes in CASE tools. This property may prevent CASE tool users from permanent recreation of `ClassifierRoles` sharing the same features. As a specialisation of `Classifier` each instance of `ClassifierRole` carries its own name, guaranteeing the identity of the prototypical objects. Therefore we suppress the property `multiplicity` in *component schemas*. The metaclasses `AssociationRole` and `AssociationEndRole` are only relevant in Collaboration diagrams and can therefore be ignored for the remainder of this section.

Realisation of Inter-Object Behaviour

The inter-object behaviour described in a given sequence diagram is represented through the metaclass `Interaction`, which aggregates instances of metaclass `Message`. The order of messages is represented through the metarelationships `predecessor` and `activator`. The objects involved in a sequence diagram are realised through the metarelationships `receiver` and `sender` between the metaclasses `Message` and `ClassifierRole`.

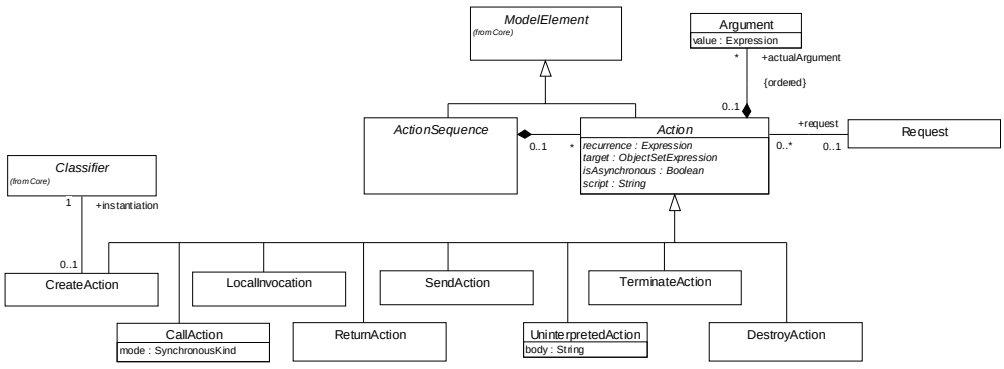
Messages and Actions

Relevant messages within a sequence diagram are represented through the metaclass `Message`. In order to allow the definition of new behavioural properties for prototypical objects, they are only related to `Requests` via the metarelationship `request`. The context of messages is defined through the metaclass `Interaction` which relates individual instances of metaclass `Message` to an instance of metaclass `Collaboration`. Through this indirection the same messages may be used in several sequence diagrams and decouple messages from operations defined for a given `Classifier` in the static structure diagram. Metaclass `Message` needs further clarification, since in the current version of the metamodel it is completely unrelated to the metaclass `ModelElement`, which contains the name of the message at hand. Therefore we introduce the specialisation between metaclass `ModelElement` and metaclass `message` (see Fig. 45, p. 94). The order of messages within sequence diagrams is represented through the relationships `predecessor` and `activator`. Arguments may also be bound to `Actions` represented through the aggregation `actualArgument`.

The individual character of a message is represented through the metarelationship `action` among the metaclasses `Message` and `Action`. According to the semantics document, `Actions` specify the computational statement, i.e. they distinguish between signals or events and various other forms of operation invocation that can be found in object-oriented programming languages. Metaclass `CallAction` represents the invocation of an operation on the receiving prototypical object. Metaclass `LocalInvocation` is used when sending and receiving prototypical objects are the same. This corresponds to the invocation of local operations via the pseudovariable `self` in Smalltalk. A `SendAction` is used for events or signals in the OPD terminology [ODP P1]. Signals do not directly lead to the invocation of an operation. Thus the reaction of an object is non-deterministic. `ReturnActions` return results to the caller. The life-cycle of prototypical objects is affected by `CreateAction`, `TerminateAction` and `DestroyActions`. Creation and destruction of prototypical objects is represented through `CreateAction` and `DestroyActions`. This approach corresponds to the keywords `new` and `delete` in C++. Otherwise the initialisation of potentially complex objects has to be revealed, which leads to a violation of encapsulation. Metaclass `TerminateAction` corresponds to the `finalize() operation` in Java. This operation masks code executed when an object has to be destroyed, for instance freeing file handles. Metaclass `UninterpretedAction` is used when no specific character of the message has been defined. This default action is a consequence of the abstract character of metaclass `Action`, that does not lead to instances.

The properties of metaclass `Action` further detail the behaviour of actions. Concurrency is defined through the property `isAsychronous` and property `recurrence` defines how many times a given `Action` is to be performed. Property `target` allows the definition of several receivers for `Actions` corresponding to a multicast in network protocolls. Finally, the semantics of property `script` remain undefined. There are two potential interpretations for this property. On the one hand it may be used for a detailed textual specification of the behaviour of an `Action` or it may be used for code generation purposes.

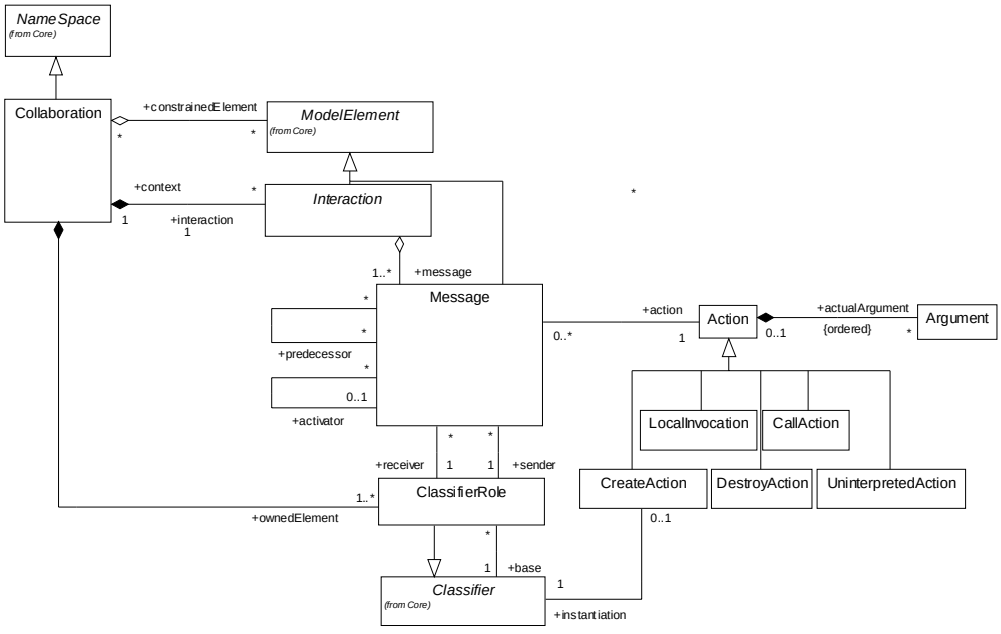
Figure 44 Actions in the UML Package Common Behaviour



Selected Actions in Component Schemas

From a problem analysis point of view, all properties of metaclass `Action` can be neglected, since they detail design and implementation aspects and potentially lead to premature design. A similar argument holds for metaclass `SendAction`, since we do not assume an event-based architectural style in the sense of Shaw and Garlan [SG96]. Sequence diagrams document behavioural requirements, i.e. the functionality that has to be realised by distinct class specification. Therefore we prefer `CallActions` to `SendActions` and `TerminateActions` can also be neglected, since they refer to the Java programming language. Due to the high-level character of problem analysis models, a detailed specification of return values in Metaclass `ReturnAction` can also be disregarded. *Component schema's* primitive data type system (see p. 68) provides another important argument against a detailed specification of return values. The properties of the metaclasses `CallAction` and `UninterpretedAction` can also be disregarded, since they aim for a detailed specification of action behaviour in design. The metamodel constructs valid in *component schema*'s subset of sequence diagrams are depicted in Figure 45.

Figure 45 Revised Metamodel for Sequence Diagrams in Component Schemas



Control Flow in Sequence Diagrams

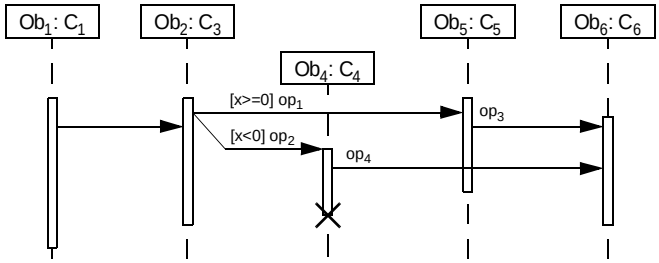
A representation of control flow in sequence diagrams is described in the UML notations guide [OMG97b]. So-called *guard conditions* [OMG97b] can be attached to individual arrows representing the `Action`, leading to branches in the thread of control (see Fig. 46, p. 95). Although this construct is principally desirable for structuring use cases, its value for problem analysis is in question. Sequence diagrams in this context provide means for determining behavioural requirements, as derived from activities in a given use case. These behavioural requirements then will be allocated to class specification in static structure diagrams in the form of operations. Control

flow, however, already reveals parts of the internal realisation and can therefore be considered as design-related. As a result we may suppress control flow in *component schemas*, leading to straightforward mapping of activities in a use case and the derived sequence diagram.

Traceability among Use Cases and Sequence Diagrams

Integrating complex use cases to the metamodel leads to a number of advantages that cannot be realised in any other way. As metamodel elements, UML trace relationships may link use case activities to messages in sequence diagrams. In fact, all information derived from complex use cases can now be linked back to the respective activity. Metaclass `Activity` in combination with the metarelationship `Trace` increases the overall traceability of *component schemas* compared to the current UML metamodel edition.

Figure 46 Control Flow in Sequence Diagrams



Section Summary

The selected subset of UML sequence diagrams depicts interacting prototypical objects as a computational interpretation of activities described in a complex use case. Sequence diagrams as specialisations of metaclass `ModelElement` have a unique name. Each use case is structured in a sequence diagram in a *component schema*. Prototypical objects are depicted as rectangles containing the name of the actual instance and the related class specification defining the offered behavioural properties. Named arrows represent messages exchanged among the prototypical objects. Messages as specialisations of metaclass `ModelElement` have a name but are not related to a given name space. As a result, messages are visible within the entire *component schema*. The individual character of messages is characterised through specialisation of metaclass `Action`. `CallActions` represent the invocation of operations offered by a given class specification, while `LocalInvocation` represents the invocation of an operation defined by the respective prototypical object. The object life-cycle is specified through `CreateAction` and `Destroy-Actions`, while `UninterpretedActions` are used as the default for messages. A dashed line under each prototypical object depicts the dimension time. Time consumed for the execution of a method, i.e. the invoked operation, is depicted by rectangles on the dashed line.

5.2.6 Component Schema Summary

The beginning of this section introduced all syntactical elements used in *component schemas*. The semantics of these syntactical elements is based on the *intuitive semantics* of UML, as stated in the UML semantics document [OMG97a]. So in this subsection we will further detail the definition of *component schemas*. *Initial partitions* are documented in terms of our selected UML subset described in the previous subsection. A *component schema* is a model documenting an *initial partition*. *Component schemas* comprise three different viewpoints, namely complex use cases, sequence and static structure diagrams. All design-related *constructs* have been eliminated, in order to guarantee a solution-neutral character of problem analysis models.

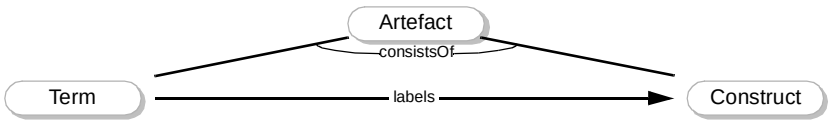
Definition23 (Component Schema) – A *component schema* is a *model-based multi-viewpoint specification* documenting requirements of an *initial partition*. It consists of the UML viewpoints use cases, sequence and static structure diagram and is populated with *artefacts.* ◊

Artefacts belong to the realm of the model world and denote those collections of real-world things of an *initial partition* deemed relevant by the analysts. In order to abstract from given syntactical elements in a *component schema*, such as class or association, we introduce the term *artefact.*

Definition24 (Artefact) – An *artefact* consists of a *term* and a *construct.* ◊

The *term* of an *artefact* expresses a *concept* and the *construct* models the regarded *aspect* of this *concept*. The relationship between *artefacts*, *terms* and *constructs* is depicted in Figure 47. The main reason for introducing the new term *artefact* is to provide a generic term for all elements of *component schemas* and to analyse the specific impact of names in concurrent requirements engineering (see Chapter 6). We leave out a lengthy summary of the selected UML constructs valid in *component schemas* (see Appendix A, p. 231).

Figure 47 Artefacts



5.2.6.1 Relating Component Schemas to the ISO Information Resource Dictionary Standard

Unlike schema integration for federated databases, requirements engineering does not deal with extensions, in the sense of tuples in a relational database management system. Therefore *artefacts* have to be regarded as intensional elements and *component schema relationships* deal with the similarity of *artefacts* in given partitions.

The ISO IRDS Architecture

The ISO Information Resource Dictionary Standard [ISO10027] identifies a four layer architecture for modelling techniques. The lowest layer only contains pure instances, i.e. records in a database. The second layer describes concepts in terms of classes and associations. In other words, a *component schema* describing an *initial partition* is located on layer two. The UML metamodel itself has to be regarded as an instantiation of layer three, where valid UML constructs are defined. This holds for the structured part of *component schemas*, namely complex use cases, sequence and static structure diagrams. Plain-text requirements statements, as encountered in operational requirements documents, have to be excluded. They do not comprise a formalised model and their unrestricted use of natural language gives rise to ambiguities. Layer four is only of the interest when the metamodel has to be adaptable. ConceptBase [CB97] designed at the computer science department of RWTH Aachen is an example of a CASE-tool realising all four levels.

Considered ISO-IRDS Levels in this Work

Relevant levels for this work are level 2 and 3, since requirements engineering does not lead to concrete instances, persistently stored in a given database management system. Level four can also be neglected, since we do not want to manipulate the metamodel at run-time. *Component schemas* are layer two elements, i.e. instances of the UML metamodel. Introducing *component schema relationships* as first-class modelling constructs leads to possible enhancements of the current version of the UML metamodel, located on layer 3 of the IRDS architecture (see Section 7.4).

5.2.6.2 Advantages of Component Schemas

Component schemas are based on a subset of UML *constructs* tailored for requirements engineering. All design-related *constructs* have been eliminated, in order to guarantee a solution-neutral character of problem analysis models. With exception of use cases, no changes have been applied to the metamodel, in order to ensure the compatibility to tool implementations based on the OMG standard [OMG97a, OMG97b]. Therefore complex use cases lead to new specialisations of the existing metaclass `UseCase` in the metamodel. While UML only regards use cases as an iconic representation for sequence diagrams, *component schemas* stress the value of textual information. Incorporating plain-text information into the repository has a number of significant advantages, especially with respect to stakeholders with limited modelling background. Incorporating the activities into the repository increases the traceability of source information, since structured and unstructured information reside in a single repository. Moreover, formalised and unstructured information can be related, increasing the comprehensibility of analysis models to business users. The intellectual process behind modelling, i.e. interpreting a problem statement in terms of a model, also becomes more transparent to business users and remains traceable later on.

Component Schemas and Stakeholders

The varying degree of formality within *component schemas* suits the different skills of project participants. Use cases can be elaborated by business users and domain experts with little coaching from modelling experts. Structuring textual requirements

in terms of object-oriented analysis models remains the prime intellectual activity for modelling experts. Documenting requirements in a textual as well as in graphical form improves the comprehensibility to all project participants. In the long term this strategy increases the success of modelling activities in industry-scale projects. Reviewing models by domain experts can be improved and the mutual suspicion and rivalry among IT and non-IT stakeholders may be improved.

Component Schema Process

The elaboration of *component schemas* can be divided into the following phases. Analysts and business users select relevant business processes in the *initial partition* at hand. Then business users and domain experts elaborate the use cases, i.e. they describe the individual activities and actors in a concise form. Analysts then review the description in order to understand the terminology and possibly improve the description of business activities. Use cases mark the first set of requirements, without any computational interpretation, documented in a problem analysis model. Afterwards analysts structure use case activities in terms of sequence diagrams. Structural requirements will be modelled in a static structure diagram. Relating plain-text to model elements increases the comprehensibility and traceability of problem analysis models. Finally, the entire *component schema* has to be reviewed with the participating business users. The reviewed *component schema* comprises the complete understanding of the problem statement in terms of structural and behavioural requirements. Only a prioritisation of requirements, i.e. an identification of mandatory and optional requirements, is still missing. This activity may be accomplished in requirements validation and therefore remains outside the scope of this work.

5.3 Using Component Schemas in Concurrent Requirements Engineering

This subsection discusses the mapping of logical partitions to *component schema constructs* as well as the different representation of *inter-partition* and *global requirements*.

Partition Environment

Class specifications provide the sole representation for structural and behavioural requirements in *component schemas*. Packages group class specifications and possibly further divide the partition into coherent logical parts, but they do not represent any kind of requirement. Unlike Wand's & Weber's [WW93] ontology for information systems, UML does not provide a distinct construct for the representation of the system boundary. Thus there it is no clear distinction between *partition contents* and *partition environment*. Consequently, a distinct representation for *inter-partition requirements* is also missing. However, as we argue in Section 8.2, this distinction is vital for the management of *component schema relationships*. Stereotypes as user-defined classifications of UML metamodel elements bypass the missing *construct* partition boundary. Jacobson [JCJ+94] for instance distinguishes between *entity*, *control* and *interface objects*. The latter category falls into two separate groups. The first group of interface objects deals with the human-machine interface, while the second one is related to interfacing with external systems. A clear distinction between these completely different kinds of requirements seems desirable in industry-scale projects.

Interfacing with External Systems

Class specifications dealing with external systems are vital for system integration and carry important requirements for the functionality of the entire system. Structural as well as behavioural requirements have to be specified, in order to specify proper interfacing for the future system. The requirements for user interface and external systems also diverge. Statements like "*The user interface shall be implemented using the Microsoft Foundation Classes"* are related to non-behavioural requirements, while statements like "*Messages received from system xyz shall be processed within .5 seconds"* comprise behavioural and temporal aspects. Therefore we prefer to use individual stereotypes to separate these different kinds of class specification. Class specifications representing the future user interface will be marked with the stereotype «`HMI`», while we reserve the stereotype «`interface`» for class specifications depicting the partition environment.

Partitions and Component Schemas

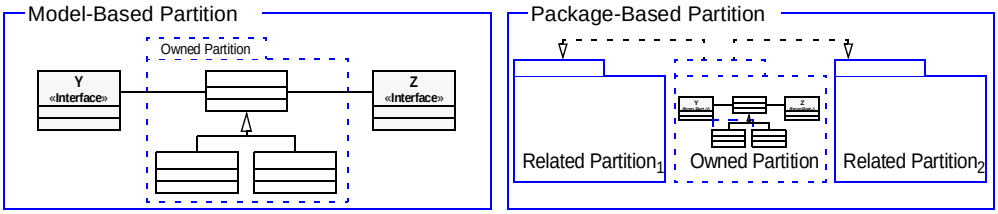
Having distinguished between the class specifications inside the area of discourse and class specifications forming the *partition environment*, we now want to focus on mapping partitions to *component schemas*. Partitions can be mapped to *component schemas* in two different ways, a *model-based* and a *package-based* approach. Since models are realised as specialisations of the UML metaclass `Package`, there is only little difference between the two approaches. Nevertheless, these different representation, have a considerable impact on the management of *component schema relationships*. *Model-based partitions* take a straightforward approach to mapping *initial partitions* to *component schemas*. An *initial partition* is mapped to a *component schema* carrying the same name. Class specifications with the stereotype «`interface`» mark the *partition environment*, while place holders for the user interface are marked with the stereotype «`HMI`». All other class specification in a static structure diagram form the actual problem statement. The second approach maps a partition to a UML package. In this context the *component schema* and the package representing the *initial partition* carry the same name.

Model- versus Package-Based Partitions

Model-based partitions take an isolated perspective to the *component schema* representing a given *initial partition.* All class specification marked with the stereotype «`interface`» are outside the area of discourse. Within industry-scale projects however, *inter-partition requirements* mark dependencies among partitions. Especially class specifications representing this category of requirements are subject to coordination among affected analyst teams. Unresolved *inter-partition requirements* threaten the completeness of a specification and in the worst case may obstruct system integration. *Package-based partitions* allow a fine grained representation of *inter-partition requirements*. As organisational units, they will be indicated using the stereotype «`partition`». Packages representing *initial partitions* can clearly be distinguished from those packages logically grouping model elements. The area of discourse is represented as a set of UML packages representing the initial partitions. Class specifications representing *known inter-partition requirements* can now be assigned to the particular partition deemed responsible for the settlement of the requirement. Class specifications representing *inter-partition requirements* are subject to the *management of component schema relationships* and have to be notified to the affected analyst teams. In this approach only class specifications deemed external

to the entire problem statement carry the stereotype «`interface`». This approach also increases the awareness of analysts assigned to other partitions for *inter-partition requirements*.

Figure 48 Model- versus Package-Based Partitions



Inter-Partition Requirements

Having introduced possible mappings of *initial partitions* to *component schemas*, we still have to clarify another highly relevant aspect, namely *inter-partition requirements*. Class specifications within the area of discourse of a given partition may depend on services provided by other partitions. The *inter-partition requirements* are crucial with respect to minimality and consistency of the overall problem analysis model. Consequently we are looking for different representations of *inter-partition requirements* in *component schemas*. Since *inter-partition requirements* reside outside the actual scope of the analyst team elaborating its assigned partition, these *inter-partition requirements* usually result in less detailed *artefacts* with respect to the *partition contents*. This gives rise to the question of suitable representations of *inter-partition requirements*.

Facade-Based Inter-Partition Requirements

Class specifications as the central abstractions for real-world things in object-oriented problem analysis therefore provide the only valid construct for a representation of structural as well as behavioural requirements in *component schemas*. Generally speaking, class specifications provide two different representations of *inter-partition requirements* on differing levels of detail. In the first possibility, *initial partitions* and their resulting *component schemas* are regarded as objects in their own right, comparable to subsystems in traditional modular design. A distinct class specification labelled with the name of the prespective *initial partition* represents it in the *owned partition*. Due to encapsulation, as one of the central principles of the object paradigm, *inter-partition requirements* have to represented as operations. We call this approach *facade-based inter-partition requirements*. Following the idea of Gamma [GHJ+95] a single class specification provides a uniform interface to a set of collaborating class specifications realising the required functionality in terms of a *component schema*. Using *facade-based inter-partition requirements* realises given requirements as an operation of the facade class and does not prescribe their concrete structure in terms of fine-grained class specifications. As detailed in Section 7.3.5, *facade-based inter-partition requirements* lead to so-called *inter-construct level relationships.*

Class-Based Inter-Partition Requirements

The other possible representation may realise structural and behavioural *inter-partition requirements* in terms of individual class specifications on a rather detailed level, compared to the *facade-based* approach. Unlike the previous representation, this approach explicitly represents structure and behaviour of the required class specification, provided from another *initial partition*. In order to assure the completeness of a requirements specification, *inter-partition requirements* are subject to careful consideration. Missing *inter-partition requirements* automatically result in an inconsistent overall problem analysis model. The central difference between a *class-* and *facade-based* representation lies in varying categories of *correspondence assumptions* and consequently different ways of managing the detected *correspondences assumptions*. Different categories of *component schema* relationships are introduced in Section 7, while their assessment and management is detailed in Chapter 8.

6 Terms in Component Schemas

The classic schema integration literature divides conflicts in structural and naming conflicts [BLN86]. This chapter analyses the use of language in *component schemas* and its individual viewpoints, in order to detail naming conflicts in the sense of [BLN86]. Each application domain has its own use of *terms* and therefore the meaning of these *terms* varies significantly. But even within a specific application domain we cannot count with a standardised use of *terms*, since stakeholders have their particular use of *terms*. Concurrent requirements engineering therefore gives rise to the problem, that two *artefacts* stemming from different *component schemas* model the same *concept*. Analysing the relationship between *names*, *concepts*, *constructs* and real-world things we discuss how linguistic support in form of lexical databases and ontologies contribute to a detection of *component schema relationships*.

6.1 The Use of Language in Component Schemas

Each application domain brings in its specific use of words, which colloquially is called an *expert language*. In this context we do not imply formal or mathematical languages, since we only refer to the use of words within a given application domain. Individual words have a specific meaning within the application domain, why we call them *expert terms* or short *terms*. *Expert vocabulary* then can be regarded as a set of *terms* with a specific meaning used within a given application domain. From this colloquial use of *expert language* we want to develop concrete definitions, that help us to analyse the character of *component schemas*.

Definition25 (Term) – A term is a word that has a specific meaning in a given application domain. ◊

Going into the details of *component schemas*, the difference between use cases on one hand and static structure as well as sequence diagrams on the other hand is obvious. Individual *activities* within use cases contain plain text statements, giving rise to the inherent ambiguity of natural language. In all other *viewpoints* just a single *term* is used to label an *artefact*. However, without prior knowledge of the application domain at hand, the meaning of individual *terms* cannot be regarded as clear. The term »board« in the sentence 'The board has left the factory' may mean a piece of lumber or the board of management of a company. After this more general reflection we want to take a closer look at the individual use of *terms* in *component schemas*.

Examples

Component schemas reflect the understanding of the analyst team assigned to a given *initial partition*. This understanding has been documented through their specific use of *terms* and their specific interpretation of the application domain in a *component*

schema. Although *terms* are consistently used within distinct *component schemas*, this does not imply that the same *term* is consistently used throughout the entire application domain. To illustrate this we present two simple examples from EUROCONTROL ORDs. The flight data processing ORD for instance states:

> "State Vector. This is an elementary observation of an aircraft position ..." [Eur97, p. 3-3]

The surveillance reference model however states:

> "Plots contain samples of the aircraft position at a certain moment in time ..." [Eur96b, p. 24]

Obviously the definitions of the *terms* »state vector« and »plot« correspond to each other, so we may conclude that they have the same meaning and therefore denote the same real-world things. These examples clearly demonstrate that the meaning of *terms* is not only determined by the application domain, but also influenced by the personal preferences of stakeholders assigned to an *initial partition*. So *terms* have to be further harmonised, in order to provide consistent naming throughout all involved *component schemas*. From a linguistic point of view, individual *component schemas* may still contain synonyms as well as homonyms, which have to be detected for the purpose of integrating *component schemas*. The individual perspective of analysts and stakeholders incorporated in each *component schema* leads to another linguistic anomaly which has not yet been mentioned, i.e. *equipollence*. Two *artefacts* denote the same collection of real-world things, but express different *concepts*. This states the observation formulated by Miller et al. [MBF+93] that a listener or reader who recognizes a word must cope with its polysemy, while a speaker or writer who hopes to express meaning must decide between synonyms.

6.1.1 Linguistic Aspects of Terms

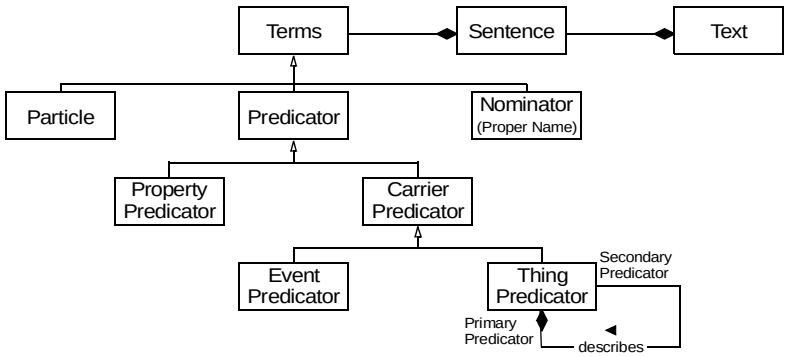
Having analysed the use of *terms* in *component schemas*, we want to take a closer look at *terms* themselves, in particular syntactic categories of *terms* in *component schemas*. Following classic naming conventions for information models, nouns are used to name class specifications while verbs are used to name relationships. This rather coarse-grain distinction only provides general rules and does not take any specific application domain into account. So beginning with an analysis of syntactic categories we want to formulate naming conventions, in order to facilitate the definition of *component schema relationships*.

6.1.1.1 Syntactic Categories in Component Schemas

In this subsection we want to analyse the use and the character of *terms* in *component schemas*. Figure 49, p. 105 depicts Schienmann's [Sch97] categorisation of *terms*, in the form of a UML static structure diagram.

This categorisation facilitates for instance the management of *terms* within a lexicon, going beyond the classification of *terms* according to simple syntactic categories, i.e. noun, verb or adverb. Individual words in a sentence can be classified into *particles*, *predicators* and *nominators*. *Particles* are syntactical elements, defining the structure of a sentence. Nominators, also called *proper names*, name real-world things in a given domain. The flight number »LH4140«, for instance, denotes a flight offered by Deutsche Lufthansa from Berlin to Paris. While *nominators* name individuals, *predicators* characterise *collections of real-world things*. *Predicators* can further be categorised into *property predicators* which generally relate to adjectives and adverbs in colloquial language. Verbs are called *event predicator* and nouns are called *thing predicator. Thing* and *event predicators* form the generalised category *carrier predicator. Thing predicators* may be further described by other *thing predicators*, leading to the recursive relationship `describes` in Figure 49. Schienmann therefore distinguishes between *primary* and *secondary predicators*, which has been represented through role names of the composition in Figure 49. Schienmann details how this category of terms can be mapped to given object-oriented modelling techniques [Sch97]. *Event predicators*, for instance, can be mapped to operations and *thing predicators* to class names and attributes in UML.

Figure 49 Categories of Terms



Morphology

Having classified *terms*, we now want to have a closer look on other syntactical aspects of *terms* labelling *artefacts*. From a linguistic perspective, *morphology* is the definition of the composition of words as a function of the meaning, syntactic function, and orthographic form of its parts. What we are really looking for is the stem of *terms*, independent of its actual flection. WordNet [WN98] for instance incorporates a morphological processing function in order to provide flexible access to its linguistic database. Since morphology as part of natural language processing is a vast research area, we leave the discussion on a rather superficial level, in order to identify further requirements for linguistic support for the integration of *component schemas*.

Composed Terms

The other severe problem within *terms* labelling *artefacts* is the use of composed terms, frequently used in engineering disciplines such as air traffic management. Composed terms can seldom be found in linguistic databases such as WordNet.

In fact due to the specialised meaning of composed terms within the application domain at hand, standard or broad coverage dictionaries only provide definitions for individual terms, but not for the new meaning formed by the composition of terms.

Subsection Summary: Predicators and Artefacts

The *terms* labelling *artefacts* have to be considered as *predicators* and therefore denote collections of things. The use *terms* within *component schemas* therefore mirrors the observation made with respect to the general character of *component schemas* introduced in Section 5.2.6.1, pp. 96. Here we stated that *component schemas* do not deal with extensions, in the sense of tuples in a relational database management system. Generally speaking a *term* labelling an *artefact* in a given *component schema* always denotes a collection of real-world things and therefore expresses a *concept*.

6.1.1.2 Naming Conventions

Taking the classic naming conventions as our starting point, i.e. nouns labelling class specifications and verbs labelling relationships, we still find significant ambiguity in naming of individual *artefacts* in given *component schemas*. Nouns labelling class specifications can be used in singular or plural. Verbs labelling relationships may also be used in active or passive voice. In order to support the reading direction for relationships, verbs may be combined with prepositions. So the relationships »pilot« »flies« »aircraft« might also be named »aircraft« »flown by« »pilot«. Although UML provides a specialised *name-direction arrow* for relationships, analysts might not make use of this *construct*. With respect to nouns labelling class specifications, some analysts may use the singular, while others use the plural. These slight differences with respect to *terms* labelling *artefacts* can easily be matched by human users, but provide serious obstacles for automatic detection of *component schema relationships*.

Naming conventions for *component schemas* involve three additional aspects, i.e. naming of *initial partitions*, operations and class specifications representing *inter-partition requirements*. Due to the linguistic complexity of *terms* labelling *artefacts*, naming conventions may facilitate assessing the *correspondence* of *terms* on an automated basis. Especially the use of existing lexical databases such as WordNet [WN98] can be improved through naming conventions. However, naming conventions are always voluntary and cannot be enforced in concurrent requirements engineering. Growing insight with respect to the problem statement leads to the definition of new conventions within the analyst teams.

Naming Partitions and Component Schemas

Initial partitions as organisational units are the only entity in concurrent requirements engineering which is known at the start of given projects. Consequently an agreement among the involved analyst teams has to be found, in order to assign a distinct name denoting the *initial partition* and its resulting *component schema*. Through the UML name space concept, all *artefacts* specified in *component schemas* are unique, since the *terms* labelling *artefacts* are prefixed with the partition name. This holds for *model* as well as *package-based partitions*. In *package-based partitions*, however, the *term* labelling the partition appears on two different levels. First it is assigned to

the *initial partition* represented as a *component schema* and secondly for the package encapsulating the *artefacts* relevant in the *initial partition* at hand. So a unique partition name already guarantees the autonomy of the analyst teams, but does not ensure minimality and consistency of the overall problem analysis model.

Stereotypes versus Class Names

A classification of *artefacts* within *component schemas* representing various kinds of requirements can be achieved by following two different strategies: prefixes to artefact names and stereotypes. A major subject within this context is a distinction between the *partition contents* and the *partition environment*. *Artefacts* in the *partition environment* represent two different forms of structural and behavioural requirements. The first category is related to *interface requirements*, i.e. systems outside the actual area of discourse maintaining relationships to *artefacts* within the area of discourse. The second category represents so-called *inter-partition requirements*, i.e. requirements that have to be fulfilled from related partitions and that will reside within the authority of the future system. Especially *inter-partition requirements* affect the minimality and consistency of the entire requirements specification.

Principally two different possibilities can be identified in order to distinguish the system environment from *inter-partition requirements*. Prefixing *artefacts names* is the first possibility and stereotypes as a UML *construct* is the second. Since our projects in the air traffic management domain used the OMT notation, four different prefixes have been developed in order to classify *artefacts* [WF97] (see Table 5). Basically class specifications denoting actors, the human machine interface, class specifications encapsulating entire (sub-) systems and class specification outside the authority of the future system have been distinguished this way.

Table 5 Naming Conventions for the Categorisation of Class Specification

Prefix	Type	Explanation
AC_	Actors	Actors represent humans interacting with the system
IO_	Interface Classes	*Interface object classes* encapsulate the human machine interface
CO_	Control Classes	Control objects encapsulate behaviour which is not naturally related to one distinct class
FA_	Facade Classes	Facade Classes describe information related to subsystems. Unlike control objects, *state* and *behaviour* related to a set of class specification in different *initial partitions* can be attached to them
EX_	External Classes	External Classes represent *artefacts* outside the domain of discourse

Although naming conventions provide a suitable distinction for the analysts, they hinder automatic means for detecting *correspondence assumptions*. Before a morphologic function can identify the stem of given *terms*, the prefix has to be removed. Relying entirely on prefixes for a categorisation is inflexible, since newly defined conventions in individual partitions may not be regarded and prefixes might be mixed up with abbreviations. As a result, prefixing *artefacts* is inflexible and also complicates a linguistic analysis of *terms*. UML stereotypes provide an elegant mechanism

for the categorisation of *artefacts* in specification, since they provide a flexible categorisation on the metamodel level and therefore do not interfere with the linguistic meaning of *terms* labelling *artefacts*. UML stereotypes also facilitate the use of linguistic databases such as WordNet in order to detect synonymous in the application domain at hand.

6.2 Terms, Constructs, Concepts and Collections of Things

As detailed in Section 5.2.6, *component schemas* have an intensional character and *are* populated with *artefacts*, stressing the man-made character of elements in the model world. *Artefacts* consist of a *name* and a *construct*. Having defined the elements of our model world, we still have to describe the elementary units of the real world, the *things*.

Definition26 (Thing) – A *thing* is an elementary unit in the real world. ◊

In other words, the real world is made up of *things* and a *term* labelling an *artefact* denotes a *collection of real-world things*. A *term* labelling an *artefact* also expresses a specific perception of this *collection of real-world things*, which is called *concept* [Web82]. The used *construct* indicates that the analysts have considered this *collections of real-world things* as a structural requirement, for example the *constructs* class and association. So at first glance, *concept* and *construct* seem to be synonyms, since both deal with perception. However, there is a significant difference between them. The *concept* can be regarded as the abstract idea that analysts generalise from the observation of real-world things in the application domain at hand, while the *construct* can be regarded as the modelled *aspect* of this *concept*. This leads to the following definition of *concepts*:

Definition27 (Concept) – A *concept* refers to an abstract idea generalised from a collection of real-world things. ◊

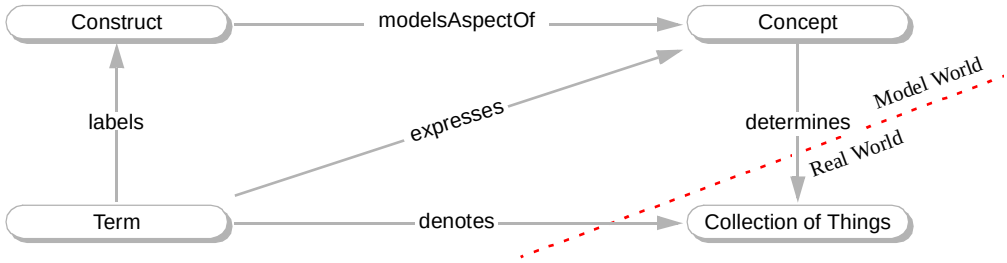
Correspondence Assumptions: a Preliminary Definition

Term, *construct* and *concept* belong to the model world, while the *collection of things* belongs to the real-world. In this light we can formulate a preliminary definition for *correspondence assumptions* between *artefacts*: A *correspondence assumption* between two *artefacts* holds, when the *terms* of the *artefacts* express the same *concept*, label the same *construct* and *determine* the same *collection of things*. Putting *term*, *construct*, *concept* and *collection of things* together leads to Figure 50, p. 109.

This preliminary definition has two central restrictions that we are going to weaken in the remainder of this work. First of all this definition does not foresee the correspondence between *primary* and *secondary predicators*. Although the *terms* express the same *concept* and denote the same *collection of things* different *constructs* have been used for the *artefacts*. But modelling *collections of real-world things* in one *component schema* as a class specification and an attribute in the other always runs the risk that the minimality of the model is in question, when such a *correspondence assump-*

tion cannot be detected. Second of all this definition does not resolve the problem of synonyms, as detailed in the examples of the EUROCONTROL ORDs (see p. 104). While the influence of different *constructs* will be discussed in Chapter 7, we detail the problem of synonyms in the remainder of this chapter.

Figure 50 Term, Construct, Concept and Collections of Things



6.3 Terms and Concepts in Lexical Databases as well as Ontologies

The *concept lattices* realised in lexical databases and ontologies code semantic and lexical relations between *terms*. In this section we are going to discuss the contribution of linguistic support for the detection of *component schema relationships*.

6.3.1 Lexical Databases

The WordNet lexical database designed by Miller et al. [MBF+93] at Princeton University is the most important reference for this work. WordNet distinguishes between an utterance of a word and its lexicalised meaning. Miller et al. call this *word form* and *word meaning* or short *form* and *meaning* [MBF+93]. There is a *n:m* relationship between *form* and *meaning*, since in the case of synonyms several *forms* have the same lexicalised *meaning*, while in the case of polysems a single *form* has several *meanings*. WordNet organises the *meaning* of *words* in a *concept lattice*, realising the linguistic relationships hyponomy and hypernomy between *terms*. As a matter of consequence the meaning of a *term*, as part of an *expert vocabulary*, corresponds to the *concept* in Figure 50, p. 109. In other words WordNet can return the lexicalised meaning of a *term* labelling an *artefact*. But there is an important restriction with respect to WordNet. As a broad coverage lexical database for english, WordNet presents *all* registered *meanings* of a given *term* to its users. In other words the user has to choose the correct meaning of a *term* in a given domain. More interesting with respect to this work is WordeNet‘s internal organisation of word *meaning* in a *concept lattice*. Synonymous *terms* labelling two *artefacts* stemming from different *component schemas* lead to the same set of possible *concepts* in WordNet‘s *concept lattice*. As a result we may weaken our preliminary definition of *correspondence assumptions* (see p. 108).

Concept Lattice and Correspondence Assumptions

WordNet's organisation of *concepts* has another interesting perspective, since its *concept lattice* also includes the linguistic relationships *synonymy, hypernymy* and *hyponomy* among *terms*. *Terms* such as »hang glider« and »airliner« have a common hypernym in the *term* »aircraft«. Although in the case of »hang glider« and »aircraft« both *terms* do not express the same *concept*, it may very well be the case that the analysts intended to model the same *collection of real-world things*. However, such kind of *correspondence assumption* also has to be verified on the structural level, i.e. attributes and relationships of a class specification in question have to be considered as well. So we can further enhance our preliminary definition in the sense that a *correspondence assumption* between two *artefacts* holds, when the *concepts* expressed by the *names* of the *artefact* have a common hypernym or synonym in the *concept lattice*. Although this assumption is correct in the mentioned example of »hang glider« and »airliner« it needs a detailed review, since WordNet does not automatically retrieve common hypernyms within a *concept hierarchy* for a given *term*. We will detail this question in Section 8.1.1, pp. 157.

Subsection Summary

Lexical databases, such as WordNet [WN98], organise the meaning of english words in a *concept lattice*, that allow to detect *correspondence assumptions* in the case that two *terms* labelling *artefacts* express the same *concept*. But WordNet simply retrieves all registered meanings of a given *term*, since WordNet has not been designed for a given application domain. Within WordNet's internal database 40% of the words have one or more synonyms, while at about 17% of the words are polysemous, i.e. have different meanings [Mil95]. Lexical databases such as WordNet are not able to detect *the „correct"* meaning of a given *term* and therefore this choice is left to the analysts. In a *master-multiple organisational model* lexical databases may support the analyst team who is driving the integration of the individual *component schemas*.

6.3.2 Ontologies

Ontologies in the philosophical sense deal with the nature and the organisation of reality. So if we were able to formalise this knowledge, we would receive a basis for reasoning about *concepts*. These ideas have been adopted in computer science, namely in artificial intelligence. Although there is still significant disagreement concerning the definition of the term ontology, we follow the definitions proposed by Guarino [GG95].

> "A *conceptualization* is an intensional semantic structure which encodes the implicit rules constraining the structure of a piece of reality" [GG95].

A *conceptualisation* organises the vocabulary of a given piece of reality and therefore provides the basis for the construction of an ontology.

> "An ontology is a logical theory which gives an explicit, partial account of a *conceptualization*" [GG95].

Categories of Ontologies

According to Guarino, ontologies can be regarded as engineering artefacts, constituted by a vocabulary used in a given domain and a set of assumptions clarifying its intended meaning [Gua98]. Guarini categorises ontologies and their underlying *conceptualisation* according to their coverage [Gua98]. *Common-sense ontologies*, such as CYC [GL93], try to provide reasoning for a variety of everyday life situations. *Domain ontologies*, on the other hand, can be regarded as the set of relevant *terms* in a given application domain. Depending on the scope of an ontology, Guarino proposes the following categorisation [Gua98]:

Table 6 Categories of Ontologies according to Guarino [Gua98]

Category	Coverage
Top-level ontology	describes the vocabulary of general things such as objects, matter, time, space, etc.
Domain ontology	describes the vocabulary related to a given domain
Task ontology	describes the vocabulary of activities or tasks
Application ontology	describes the vocabulary of things depending on particular task and domain ontologies

A *top-level ontology* provides a generic description of domain-independent real-world phenomena, such as time, space, objects and events. Based on *top-level ontology, domain ontologies* specialise the existing *terms* and introduce the vocabulary of a distinct application domain. In other words, the domain ontology defines the meaning of given *terms* in the application domain at hand, i.e. the *term* »bank« in the finance domain denotes a financial institution. Finally the subset of *terms* of a given *domain ontology* realised in a software system is called *application ontology.* A comparatively small amount of well-defined *terms* represent key abstractions of a given reality from a described perspective. While the former categories focus on structural aspects, *task ontologies* formalise behavioural aspects such as »billing« or »accounting«. Behavioural and structural aspects are usually incorporated in an *application ontology.*

Component Schemas versus Ontologies

In this light, *component schemas* can be regarded as *partial application conceptualisations*. It is clearly not an ontology, because of its lack of logical theory. Semantic relationships between *terms* labelling the *artefacts* fulfil Guarino's definition of *conceptualisation*. The availability of a domain ontology implies a suitable *conceptualisation*, which is exactly the central challenge in problem analysis. What are relevant real-world things in an application domain and which of these real-world things are relevant for the application at hand? Obviously problem analysis and the creation of domain ontologies share similar challenges. Furthermore, both disciplines require extensive prior knowledge, which is usually not available beforehand.

Elaborated *component schemas*, on the other hand, provide an excellent starting point for deriving an application ontology. UML relationships, such as compositions and generalisations, may be transformed into linguistic relationships. Due to the individual and arbitrary perspective of an analyst team elaborating a *component schema*, the

derived ontology might not follow common sense concerning the application domain at hand. *Component schemas* nevertheless provide an excellent starting point for the definition of a domain *conceptualisation*. The *terms* labelling *artefacts* describe real-world phenomena relevant in a given problem statement, which are to be realised in the future system.

Terms, Concepts and Ontologies

Going back to the relationship between *terms*, *concepts*, *constructs* and *collections of real-world things*, we may conclude that a well-defined domain ontology supports the clarification of the meaning of *terms* in *component schemas*. The ontology may provide the *concept* expressed by a given *term*. Especially encoding linguistic relationships in the ontology, such as synonymy, antonymy, homonymy, polysemy, hypernyms, hyponym, meronym and holonymy further increases the relevance of an ontology for the detection of *correspondence assumptions* among *artefacts* stemming from different *component schemas*.

6.3.3 Section Summary

Lexical databases such as WordNet [WN98] or ontologies such as OntoLingua [FFR96] may provide the meaning of *terms* and therefore help us in the detection of *correspondence assumptions*. Both technologies, ontologies as well as lexical databases, basically resolve synonymous *terms*. Their *concept lattices* also allow incorporating the linguistic relationship hypernym and hyponym between *terms* for the formulation of *correspondence assumptions*, refining our rather strict preliminary definition (see p. 108). The central difference between ontologies and lexical databases can be seen in their coverage. While WordNet covers a broad range of words in natural language, a domain ontology only reflects relevant *terms* in a given application domain. WordNet usually presents a set of meanings stored in its *concept lattice*, while a domain ontology contains a single concept relevant in the domain at hand.

Context and Application Domains

Especially domain ontologies suffer an import restriction, which is directly bound to its purpose. *Domain ontologies* only incorporate the meaning of those *terms* relevant in the domain at hand. Taking the *term* »sensor« as our example, its meaning may be a surveillance sensor in the air traffic management domain, while it might be a sensor for measuring the cabin temperature in a hypothetical aircraft spare parts ontology. So whenever the *concept* expressed by a *term* stems from another application domain, an erroneous *correspondence assumption* may be indicated. This problem leads to the definition of *concept context conflict* later on in this work (see Definition 57, p. 150).

Equipollence

Like synonyms and homonyms, equipollence is another linguistic deficiency [Ort97] in the use of *terms*. Equipollence means, that two *names* express two different *concepts* but these *concepts* determine the same *collection of things*. For instance flights can be regarded from a flight data processing as well as from a air-ground communications perspective, leading to completely different *concepts*. A *correspondence assumption* in this context does not hold for the *concepts* but is interesting from a

pragmatic point of view. Central concern is in this context is the minimality of the resulting *problem analysis model*. As a rule of thumb equipollent class specifications should not share two much common attributes, since otherwise it may the same *concept* and the *terms* have not been properly used. Linguistic support in the form of ontologies and lexical databases is not able to provide indications for this relationship, since the common criteria among the equipollent *concepts* is the *collection of things* that belongs to the real-world and therefore remains outside our scope.

Summarising may be said that lexical databases as well as domain ontologies suffer important restrictions. While lexical databases are too general and therefore fall short in *composed terms*, domain ontologies only cover the application domain at hand. Although domain ontologies compensate the lack of composed terms in WordNet, they still may indicate erroneous *correspondence assumptions*. This happens when the *concepts* expressed by the *terms* labelling two *artefatcs* stem from different application domains. The CYC project [LGP+90] showed the enormous complexity of treating common sense in ontologies. For time being lexical databases and ontologies support the detection of *corrspondence assumptions*, although the users have to be aware of the mentioned restrictions.

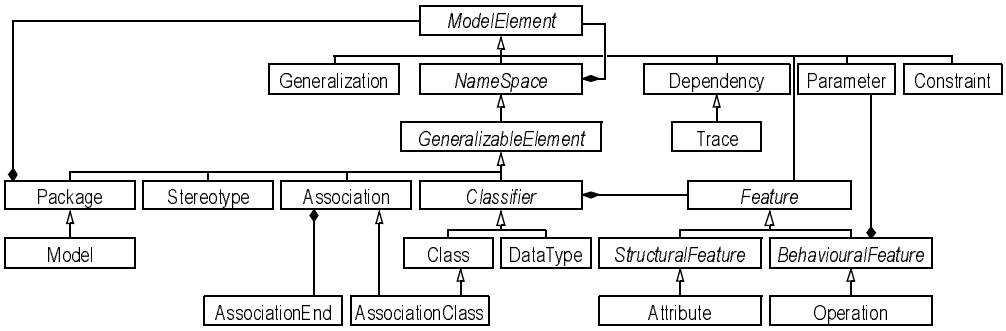
6.4 Terms and Artefacts in Component Schemas

The UML metamodel describes ‘legal’ *constructs* in *component schemas* and therefore defines how *terms* may label *constructs* in the model world. This distinction is crucial for the formulation of *component schema relationships* in the next chapter, where we especially detail the influence of *constructs* on the *correspondence* of *artefacts*. We are using the UML metamodel to analyse how *constructs* are *labelled* by *terms*. As we will see later on, not all *constructs* are named and therefore require additional considertaions.

6.4.1 Terms and Artefacts in Component Schemas’ Static Structure Diagrams

In order to clarify the issue we have amalgamated the different figures from Section 5.2 into a single diagram depicting all valid *constructs* of the UML static structure diagrams. For clarity reasons, we concentrate exclusively on the metaclass hierarchy and leave out the relationships of the relevant metaclasses. All relevant static structure constructs can be found in Figure 51, p. 114. Following the UML metamodel convention, abstract metaclasses, i.e. classes that do not lead to instances, are depicted in italics. Generally speaking, all leaf classes in the UML metamodel are instantiable and therefore lead to *artefacts* in *component schemas*. Except for the metaclasses `Dependency`, `Generalization` and `Trace` all other *constructs* are labelled by a *term*, represented through the property `name` in their generalised metaclass `ModelElement`. So the *term* labelling an *artefact* is realised as the property `name` in the UML metaclass `ModelElement`.

Figure 51 Relevant UML Metamodel Elements for Component Schemas' Static Structure Diagrams



UML relationships demand further consideration. Due to their relational character, their existence is coupled to the existence of their *constituent classes*. In other words, a relationship can only exist when its constituent classes exist. So we use the term *link artefacts* when we refer to relationships from a linguistic perspective. Analysts often leave out relationship names and therefore those relationships do not express a *concept*. *Composition* and *generalization* are anonymous relationships, i.e. they are not *labelled* at all. Hence, anonymous relationships require further consideration, since without a distinct name they do not have a linguistic meaning of their own and therefore do not express a *concept.* Table 7 lists the selected *constructs* relevant in *component schemas*' static structure diagrams.

Table 7 Categorisation of Constructs in Component Schemas' Static Structure Diagrams

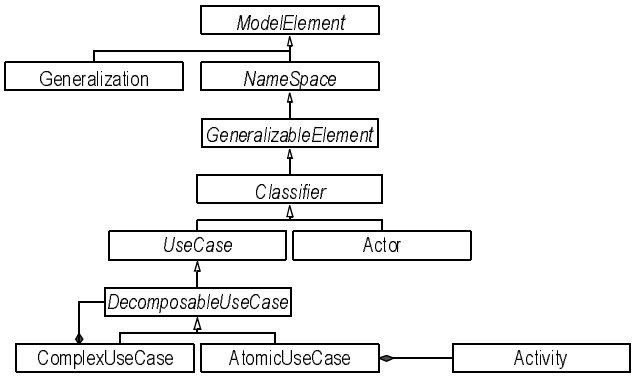
	Construct	**Related Construct**	
Artefact	Class	Package	named
	Package	Model	named
	Attribute	Class & Association Class	named
	Operation	Class & Association Class	named
Link Artefact	(Binary) Association	Class	named
	Specialisation	Class	anonymous
	Composition	Class	anonymous

This distinction is necessary for an analysis of the linguistic meaning of *artefacts* within a *component schema*. Notably the treatment of *link artefacts* requires further consideration, detailed in Section 8.1.2. Semantic relationships are in particular the central challenge with respect to *component schemas*. Unlike binary associations (metaclass Association), their linguistic meaning depends on the linguistic relationship between names of their constituent classes.

6.4.2 Terms and Artefacts in Component Schemas' Use Case Diagrams

Having categorised the use of *terms* in *component schemas*' static structure diagrams, we now want to apply the developed categorisation to complex use cases, depicted in Figure 52.

Figure 52 Relevant UML Metamodel Elements for Component Schemas' Use Case Diagrams



For reasons of clarity, we suppress properties and relationships in this figure. Once again only leaf classes in the specialisation hierarchy lead to instances. All leaf classes with the exception of metaclass `Generalization` are *artefacts*. Since *uses* and *extends* relationships have been replaced by the property `pathExpression` in the metaclass `ComplexUseCase` metaclass `Generalization` will never be instatiated in *component schemas*. Metaclass `Activity` can also be neglected due to their unrestricted plain-text description of individual activities within the described use case. So the only relevant *artefacts* within *component schemas*' use case diagrams are complex and atomic use cases, as well as the participating actors.

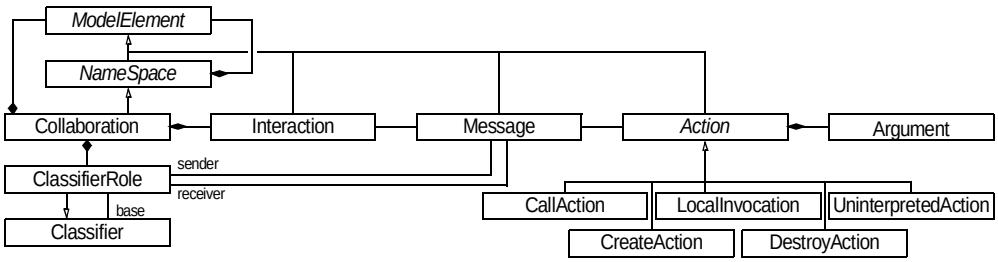
Table 8 Categorisation of Constructs in Component Schemas' Use Case Diagrams

	Construct	Related Construct	
Artefacts	Atomic Use Case	ComplexUseCase, Collaboration	named
	Complex Use case	ComplexUseCase	named
	Actor	Class	named

6.4.3 Terms and Artefacts in Component Schemas' Sequence Diagrams

Sequence diagrams form the last UML view within *component schemas*. Each instance of a sequence diagram carries a unique name. The name should be identical to that of the atomic use case the sequence diagram is based on. Metaclass `Action` and its specialisation can be neglected, since they further specify the character of individual messages.

Figure 53 Relevant UML Metamodel Elements for Component Schemas' Sequence Diagrams



Instances of metaclass `ClassifierRole` represent *prototypical objects* exchanging messages. The message exchange between the *prototypical objects* is realised through the metarelationships `sender` and `receiver`. With respect to problem analysis the *receiver* is more interesting, since the *class specification* the instance of `ClassifierRole` is based on has to fulfil the request by calling an appropriate operation. As a result, an operation carrying the same name has to be defined for the `Classifier`. With respect to the use of terms in sequence diagrams, the receiving `ClassifierRole` has to be regarded as the more important one, since it has to realise the behavioural requirement stated through the interaction between the `ClassifierRoles` at hand. However, the relationships between instances of the metaclasses `ClassifierRole` and `Class` are more complicated. Whenever a class specification realising the prototypical object has been assigned to the instance of `ClassifierRole`, the name of the `ClassifierRole` has to be regarded as a *proper name* uniquely identifying the instance. Otherwise the name of the `ClassifierRole` itself is used for this purpose in the context of a sequence diagram. This leads to the following linguistic classification of terms in sequence diagrams in Table 9.

Table 9 Categorisation of Constructs in Component Schemas' Sequence Diagrams

	Construct	Related Construct	
Artefacts	Classifier Role	Class	named
	Collaboration	Use Case	named
	Argument	Message	named

	Construct	Related Construct	
Link Artefacts	Message	ClassifierRole	named
	Message	Class	named

Terms and Artefacts: Subsection Summary

The property `name` of the UML metaclass `ModelElement` represents the *term* labelling an *artefact* in the UML metamodel. The particular properties of UML relationships lead to the distinction of *link artefacts*, caused by they dependence of relationships on their *constituent classes* and the fact that analysts often leave out relationships names. Therefore the meaning of *link artefacts* from a linguistic perspective depends on the meaning of its *constituent classes*. *Anonymous artefacts* are those *artefacts*, that do not have an individual name, such as the UML relationships composition and generalisation.

6.5 Section Summary

Component schemas are characterised by a differing use of natural language. While use case *activities* comprise whole sentences, all other UML viewpoints only use a single *term* to label *artefacts*. Due to the concurrent analysis of *initial partitions terms* are consistently used within the individual *component schemas*, but this does not imply that the same *term* is also consistently used throughout the entire application domain. Therefore concurrent requirements engineering especially demands for a standardisation of *terms*.

Terms label *artefacts* and *terms* express a *concept*. The *concept* expressed by a *term* determines the *collection of real-world things* that are the denotation of the *term*. Combining the relationship between *term*, *concept* and a *collection of things* with the *construct* we may propose a preliminary definition of *correspondence assumptions*.

Lexical databases and domain ontologies may overcome the problem of synonyms in *component schemas* and therefore contribute to the detection of *correspondence assumptions*. Their *concept lattice* presents different meanings for a set of synonymous *terms* and therefore may indicate that the terms at hand have the same meaning. But both tools suffer important restrictions. While lexical database fall short in *composed terms* and therefore are not able to indicate a *correspondence assumption*, domain ontologies may indicate erroneous *correspondence assumptions* since the *terms* in at hand express different *concepts* stemming from different application domains.

Finally an analysis of the utterance of terms in our selected UML views of a *component schema* leads to the observation that the *concept* of associations is also influence by its *constituent classes*. Often analysts do not name associations and as a matter of

consequence, the association at hand does not express a *concept*. This also applies to the UML relationships generalisation and composition, which are anonymous in nature and therefore never express a *concept* of their own.

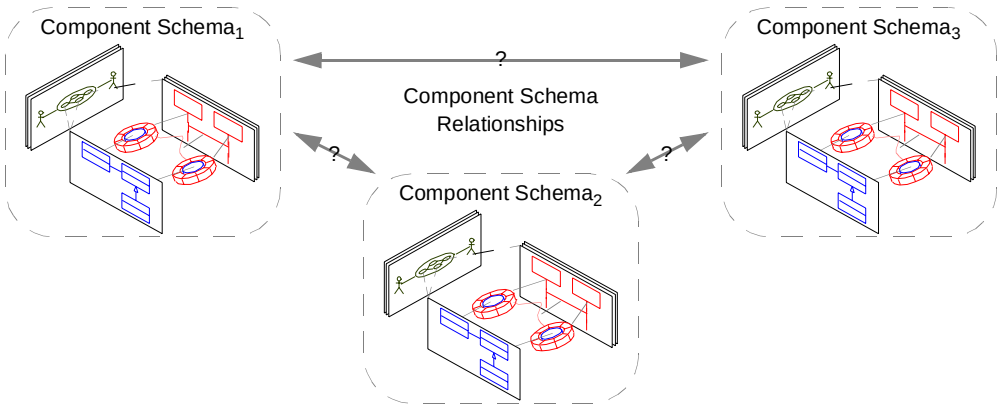
This chapter therefore has especially analysed the influence of naming with respect to the detection of *correspondence assumptions* among *artefacts* stemming from various *component schemas*. Linguistic support via lexical databases or domain ontologies significantly contributes to the detection of *correspondence assumptions*. In order to complete the analysis of *correspondence assumptions* we are going to detail the influence of different *constructs* in the next chapter.

7 Component Schema Relationships

In concurrent requirements analysis, the entire problem statement is divided into set of *initial partitions*, which are represented in terms of *component schemas*. Business processes and stakeholders, as their central analysis perspectives, are likely to introduce so-called conceptual overlaps among *component schemas*. These conceptual overlaps have the potential to introduce redundancy as well as conflicts to the resulting software requirements specification.

This chapter clarifies the nature of conceptual overlaps among *component schemas*. Taking the discussion in Section 4.4 as our starting point, conceptual overlaps are regarded as relationships among *component schemas*, so-called *component schema relationships*. The specific character of *component schemas*, as a *model-based multi-viewpoint specification technique*, call for a detailed discussion of *multi-viewpoint relationships*. Due to the special character of requirements specifications, we restrict *component schema relationships* to static structure diagrams. Batini's taxonomy of conflicts provides the baseline for a discussion of potential conflicts among component schema's static structure diagrams. The resulting categorisation of *component schema conflicts* will then be complemented by so-called *component schema correspondences*. Our definition of *component schema correspondence* extends the primarily structural character of conflicts in the federated database community towards linguistic aspects, relevant in concurrent requirements engineering. This chapter concludes with a realisation of *component schema relationships* in terms of UML metamodel constructs. Figure 54 depicts the complicated nature of relationships among *component schemas*, comprising use case, sequence and static structure diagrams.

Figure 54 Component Schemas and Component Schema Relationships



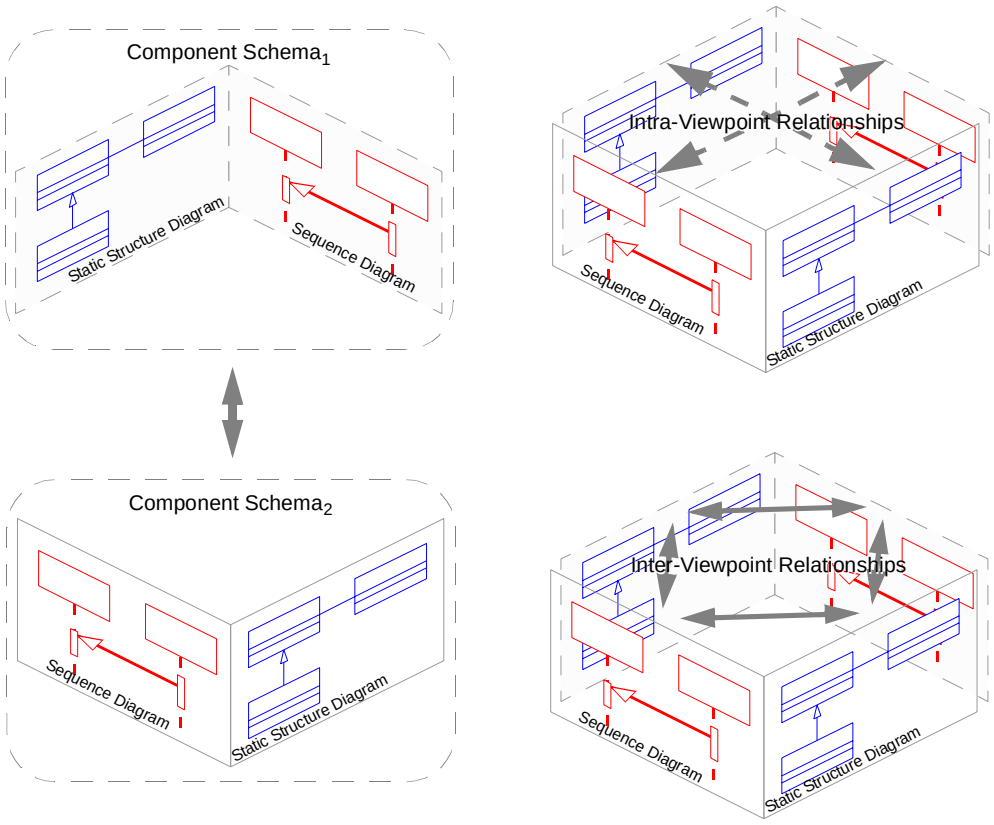
7.1 Intra- versus Inter-Viewpoint Relationships

Viewpoint relationships relate arbitrary elements specified in different viewpoints. Since *component schemas* comprise three distinct *viewpoints*, namely *complex use cases*, *sequence* and *static structure diagrams*, we further have to clarify the concrete nature of *component schema relationships* as the central contribution of this work. First of all we characterise the possible relationships among individual *viewpoints*.

Relevant Viewpoints

The plain-text character of *complex use cases* does not provide the necessary degree of formality for a detailed analysis of *inter-viewpoint relationships*. So we may concentrate on *sequence* and *static structure diagrams*. Since the remaining *viewpoints* contain a method-specific representation of use cases' plain-text descriptions, no information will be lost. Following the distinction of Nuseibeh, *viewpoint relationships* can be divided into *inter-* and *intra-viewpoint relationships* [NKF94]. Relationships among *component schemas* using the same modelling technique are called *intra-viewpoint relationships*, while relationships among viewpoints using different modelling techniques are called *inter-viewpoint relationships*. Figure 55 depicts *intra-* and *inter-viewpoint relationships* in *component schemas*.

Figure 55 Intra- and Inter-Viewpoint Relationships among Component Schemas



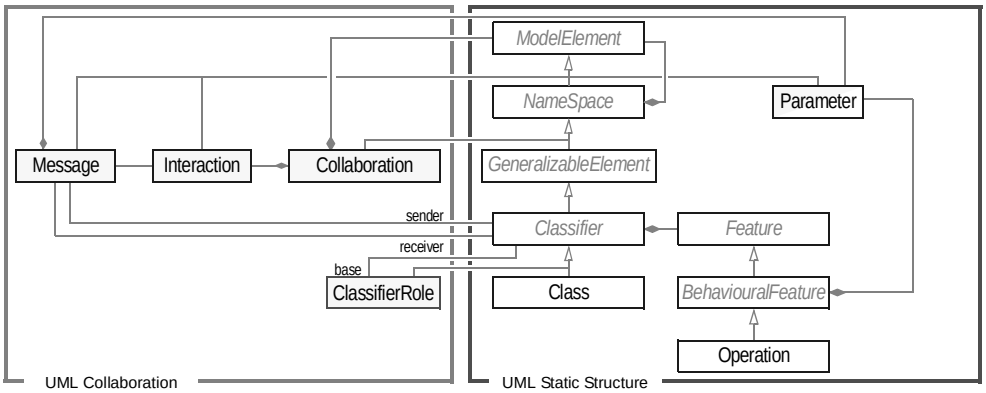
Intra-Viewpoint Relationships

Taking this distinction as our starting point we now can distinguish the potential relationships between the two viewpoints in *component schemas*. Intra-viewpoint relationships apply to relationships among sequence diagrams stemming from different *component schemas* as well as to relationships among static structure diagrams stemming from different *component schemas.* Due to the method-specific character of modelling techniques applied in the respective *viewpoints*, we may concentrate our analysis on the *terms* labelling *artefacts.* In other words, we may compare the names attached to *constructs* in the individual viewpoints. Two *artefacts* stemming from different *component schemas*, that are labelled by the same *name* and express the same *concept* have to be considered as representations of the same *collection of real-world things*.

Inter-Viewpoint Relationships

Inter-viewpoint relationships describe the relationships among viewpoints using different modelling techniques. With respect to *component schemas* we have to clarify potential relationships among static structure and sequence diagrams stemming from different *component schemas*. So we have to discuss the relationship of a *prototypical object* in a *sequence diagram* and a *class specification* in a *static structure diagram* labelled by the same name. For a number of reasons, inter-viewpoint relationships are quite problematic. First of all they demand for semantic integration of the respective modelling techniques. In order to illustrate this problem, Figure 56 depicts the superimposition of relevant constructs stemming from the UML metamodel packages `Collaborations` and `Core`.

Figure 56 Superimposition of the UML Metamodel for Collaborations and Static Structure



The UML metamodel claims to provide "*a comprehensive and precise specification of UML's semantic constructs*" [OMG97a, p. 1]. If so, we still may ask how this integration may be justified from a formal point of view? We may ask for the semantics of classes, objects, and messages, just to take a few examples of component schema constructs. UML semantics are not defined on a common semantic base, where individual viewpoints can be regarded as projections from a unifying semantic basis.

That means, at least from a mathematical point of view, that UML does not provide sufficient formal rigor for the definition of precise semantics of inter-viewpoint relationships.

Behavioural Specifications in Problem Analysis

Despite the lack of formal rigor in sequence diagrams, the genuine objective of problem analysis also puts an assessment of *inter-viewpoint relationships* into question. Within requirements engineering, problem analysis models should specify user demand and facilitate a thorough understanding of the problem statement. *Component schemas* cover this demand through a combination of *structural* and *behavioural requirements* expressed through a set of *class specifications* captured in static structure diagrams. Analysing the equivalence of behavioural specifications, stated in sequence diagrams, focuses on the specific way behaviour might be realised in the implemented system and therefore tends to violate the solution-neutral character of problem analysis models.

But there is another significant argument against the rather intricate analysis of inter-viewpoint relationships. According to our experience, *component schemas* are often built on the basis of operational requirements documents, following functional decomposition techniques. The resulting *component schemas* only provide a static representation of *functional requirements* in terms of operations in representing behavioural properties of real-world things. Concentrating on *intra-viewpoint relationships* of component schema's static structure diagrams thus bypasses the intricate clarification of inter-viewpoint semantics and also results in a dramatic reduction of efforts. Especially in industry-scale applications with a large number of class specifications, less constructs have to be considered for an assessment of viewpoint relationships.

> Definition28 (Component Schema Relationship) – Component schema relationships refer to *intra-viewpoint relationships* between *artefacts* in the UML static structure view stemming from different *component schemas*. ◊

Through the restriction of our analysis to static structure diagrams, we may use the term *component schema relationships* synonymously for the more exact explanation in Definition 28. So for the remainder of this work we stick to the term *component schema relationships*.

7.2 Intra-Viewpoint Relationships for Component Schemas' Static Structure Diagrams

Over the last 20 years, the schema integration community has published various approaches, mostly intended for the integration of federated databases. As the classic literature mostly covers the relational datamodel, we concentrate on the semantically richer set of constructs in *component schemas*' static structure diagrams. For the time being, we directly use the term *conflict*. This definition presupposes that the real-

world meaning of elements stemming from different schemas has positively been clarified. In other words, the *terms* labelling these *artefacts* determine the same *concept.*

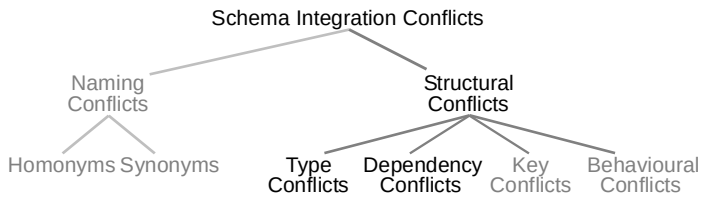
Conflicts in Schema Integration

Heterogeneity conflicts [SPD92] dealing with the integration of schemas using different data models do not appear in *component schemas*. Our definition of *component schema relationships* directly excludes this category of conflicts. Given two *artefacts* stemming from different *component schemas,* Batini et al. define the following taxonomy of *component schema relationships*. Two elements may be *identical*, *equivalent*, *compatible,* or *incompatible* [BLN86]. In the schema integration community, the equivalence of compared elements has mostly been based on the set-theoretic consideration of attribute domains.

Definition29 (Domain) – The set of values of an *attribute* for which a function is defined. ◊

However, when two attributes share the same domain, we cannot conclude that their class specifications denote the same real-world thing. The class specifications `Man` and `Tower`, for instance, might both carry a property named `height`. Assuming both properties share the same domain and the same scale, they still represent different *collections of real-world things*. Figure 57 depicts Batini's taxonomy of conflicts [BLN86].

Figure 57 Taxonomy of Schema Conflicts according to [BLN86]



Naming Conflicts in Schema Integration

The authors basically distinguish between *naming conflicts* and *structural conflicts*. Naming conflicts mostly deal with the meaning of *terms* labelling *artefacts* stemming from different *component schemas*. A systematic treatment of homonyms and synonyms is left to the database designers. In the so-called *schema conformance phase* [BLN86], naming conflicts have to be resolved manually before the actual integration starts. With respect to *component schemas* the essence of this kind of conflicts is discussed above in Section 6.2.

Structural Conflicts in Schema Integration

Structural conflicts have been classified into *type, dependency, key* and *behavioural conflicts.* Within these four categories key and behavioural conflicts are related to instances in the sense of ISO IRDS [ISO10027]. Therefore this kind of conflicts are not relevant for *component schemas*, where we primarily focus on the relationship among *artefacts.* Type conflicts summarise all conflicts where different *constructs*

have been applied to the same real-world thing and dependency conflicts concentrate on differing cardinality constraints applied to relationships. Since [BLN86] base their work on the Entity-Relationship-Model [Chen76], the classification of structural conflicts does not cover the semantically richer constructs of component schema's static structure diagrams.

Conflicts in Component Schemas

In order to prevent mixing up the terms »types« and »dependency«, we call them *construct* and *surroundings conflict*. These terms further stress the origin of these kinds of conflicts.

> Definition30 (Construct Conflict) – A *construct conflict* refers to *artefacts* with different *constructs*, that stem from different *component schemas* and express the same *concept*. ◊

A *construct conflict* corresponds to type conflicts in the sense of Batini [BLN86].

> Definition31 (Surroundings Conflict) – A *surroundings conflict* refers to *artefacts* stemming from different *component schemas* are connected using different relationships. ◊

Surroundings conflicts are called dependency conflicts in the work of Batini [BLN86]. The following subsection details *construct* and *surroundings conflicts* with respect to component schema's static structure diagrams.

7.2.1 Construct Conflicts in Component Schemas' Static Structure Diagrams

Class specifications are the central *construct* within *component schemas*' static structure diagrams and play the most important role within this category of conflicts. We further assume that the real-world meaning of regarded elements has been clarified, i.e. *artefacts* express the same *concept*. Real-world things are represented in terms of the *constructs* `Class`, `Attribute` and `Operation` in component schema's static structure diagrams. Within the permutations of these three constructs, we want to concentrate on *class-class*, *attribute-class*, *attribute-operation* and *operation-class conflicts*. *Attribute-class conflicts* directly correspond to Batini's type conflicts [BLN86]. *Attribute-class* as well as *operation-class* conflicts refer to *artefacts* expressing the same *concept* on different levels of abstraction. As detailed in Section 6.1, pp. 103, class names correspond to *predicators*, while attribute and operation names correspond to *secondary predicators* [Sch97]. As a result, conflicts among *artefacts* stemming from different *components schemas* can be categorized along the following planes: *construct-kind* and *construct-level*. Examples of these different categories of *construct-conflicts* can be found in Figure 58, p. 125.

Construct-Kind Conflict

> Definition32 (Construct-Kind Conflict) – Construct-kind conflicts refer to *artefacts* stemming from different *component schemas* with dissimilar *constructs* that express the same *concept*. ◊

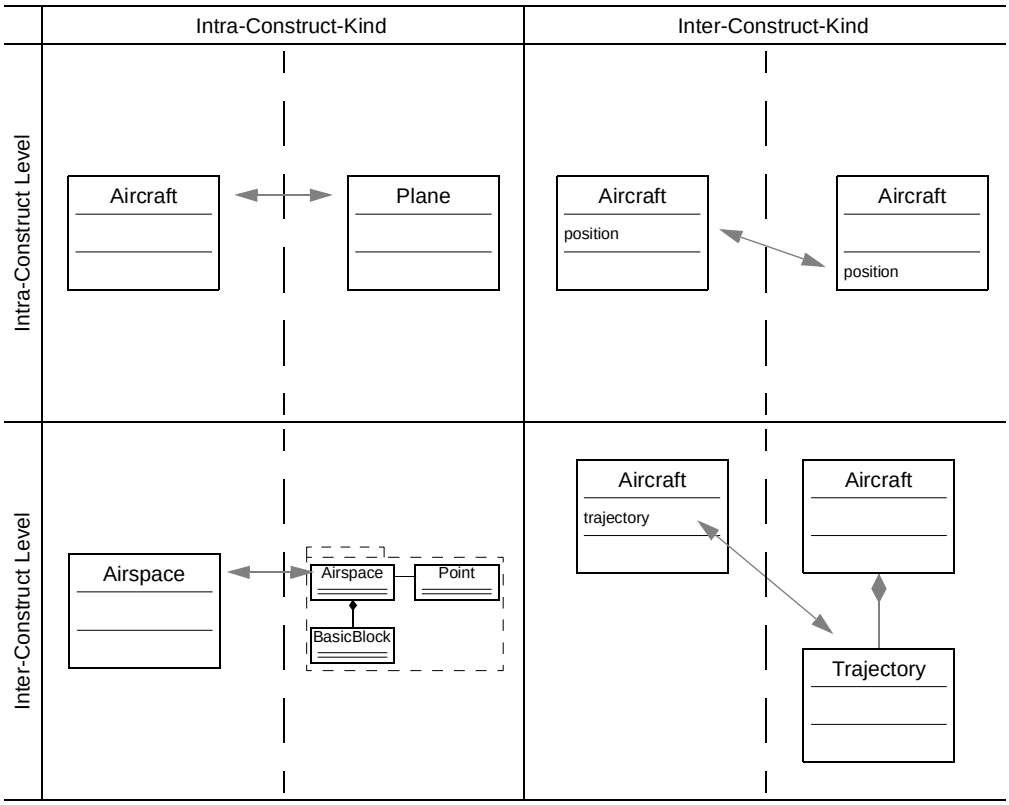
Similar to our distinction of viewpoint relationships, we may distinguish between *intra* and *inter-construct-kind conflicts*. *Class-class, attribute-attribute* and *operation-operation conflicts* refer to *intra-construct-kind conflicts*. Consequently *operation-attribute conflicts* refer to *inter-construct-kind conflicts*.

Construct-Level Conflict

All remaining permutations of the mentioned constructs refer to *artefacts* on different levels of abstraction, i.e. a primary and a secondary predicator express the same *concept*. Therefore we have to introduce a new category of conflicts:

Definition33 (Construct-Level Conflict) – Construct-level conflicts refer to *artefacts* stemming from different *component schemas* that express the same *concept* and have been represented on a dissimilar level of abstraction. ◊

Figure 58 Superimposition of Construct-Kind and Construct-Level Conflicts



Once again we may distinguish between *intra-* and *inter-construct-level conflicts*. An *operation-attribute conflict* is an example of an *intra-construct-level conflict*, since both attribute and operation names are secondary predicators from a linguistics point of view. *Attribute-class* and *operation-class conflicts* are examples of *inter-construct-*

level conflicts and in this case *primary* and *secondary predicators* express the same *concept*. *Construct-kind* and *construct-level conflicts* can also be superimposed. Except for *intra-construct-kind inter-construct-level conflicts*, we have already introduced examples of the three remaining categories within this superimposition of the conflict categories.

Intra-Construct-Kind Inter-Construct-Level Conflict

The UML name space concept provides an example of two *artefacts* with the same *construct* on different levels of abstraction. In this context, two class specifications have been modelled in different packages levels, as depicted in the lower left half of Figure 58 (see p. 125). The class `Airspace` is depicted in the top-level package in one component schema and is grouped into a package in the other. Unlike the difference between class and property names, UML packages introduce logical groupings of *artefacts* that finally result in the different levels of abstraction. The superimposition of the two dimensions *construct-kind* and *construct-level conflicts* leads to the examples depicted in Figure 58. Detailed examples of individual conflicts can be found in Appendix C and Appendix D.

7.2.2 Surroundings Conflicts in Component Schemas' Static Structure Diagrams

Class specifications in *component schemas* often participate in relationships forming their environment and therefore result in *surroundings conflicts*. Although surroundings conflicts appear as *construct conflicts* at first glance there is additional consideration required. This conflict category subsumes all kinds of UML relationships connecting class specifications in *component schemas*, i.e. associations, generalisation/specialisation and composition. First of all, we have to go into the details of relationships stemming from different *component schemas*, since class specifications depend on their *constituent classes*:

> Definition34 (Constituent Classes) – Class specifications that are connected via a relationship are called *constituent classes* of the relationship. ◊

On the basis of the *constituent class correspondence* we can detail *association type* and *association direction conflicts* within *surroundings conflicts*.

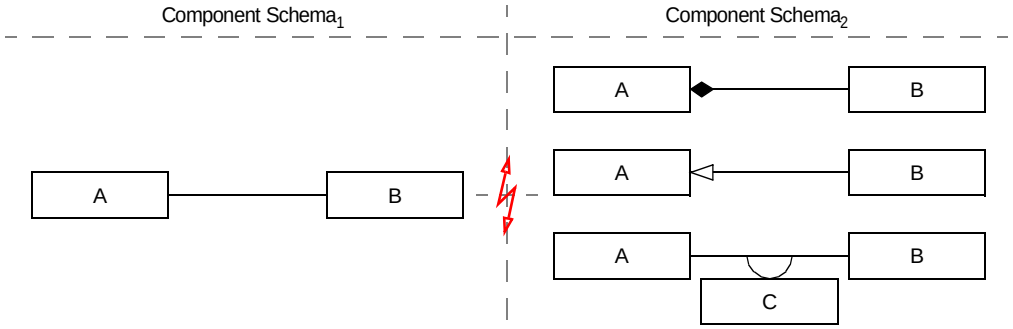
Surroundings Conflicts: Association-Type Conflict

Before we can analyse *surroundings conflicts* we have to make sure, that constituent classes of relationships stemming from different *component schemas* determine the same *concept.* Disregarding the potential impact of relationship naming for the moment, we find three distinct categories of surroundings conflicts in *component schemas*. Relevant *constructs* for relationships are `Association`, `Association Class`, `Generalization` and `Composition`. The first category within *surroundings conflicts* focuses on the association type, i.e. the constituent classes have been connected using different *constructs*.

Definition35 (Association-Type Conflict) – An association-type conflict is a *surroundings conflict* where different *construct*s have been used for the relationships. ◊

Figure 59 depicts examples of *association-type conflicts*:

Figure 59 Surroundings Conflicts - Association Type Conflicts



Incorporating the concrete semantics of individual UML relationships has been left out for the moment, since we are interested in outlining all possible combinations of conflicts from a structural perspective.

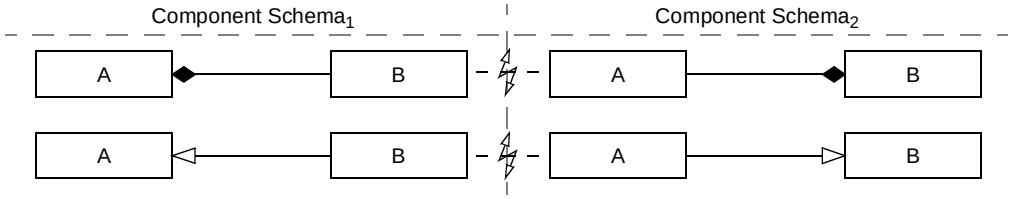
Surroundings Conflicts: Association-Direction Conflict

Besides binary associations, *composition* and *generalisation* in UML are directional associations, providing a second dimension of *surroundings conflicts*. Due to the semantic nature of these relationships, which will not be discussed for the moment, this kind of conflicts directly influences the correctness of *component schemas*.

Definition36 (Association-Direction Conflict) – An association-direction conflict is a *surroundings conflict* where *semantic relationships* have been applied in a dissimilar direction. ◊

In other words, *aggregate* and *component* are mixed up in the particular *component schemas*. Figure 60 depicts *association-direction conflicts*.

Figure 60 Surroundings Conflicts-Association Direction Conflict



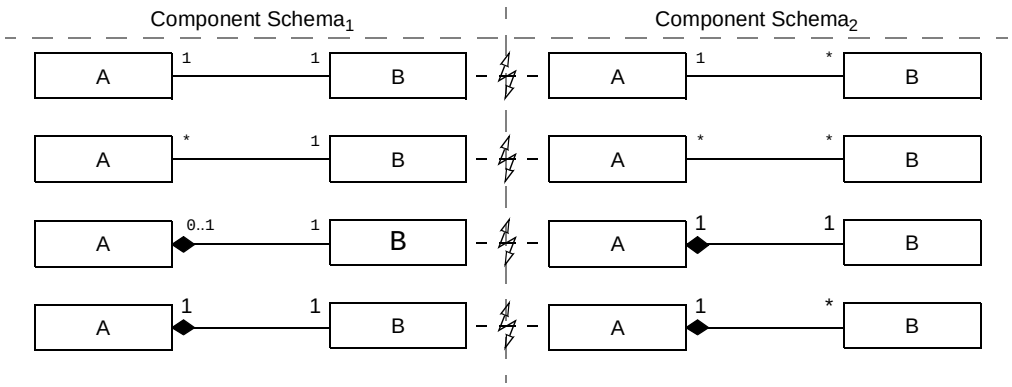
Surroundings Conflicts: Cardinality Conflict

The number of participating objects – instances – in a relationship can be restricted in UML using cardinality constraints, which are represented in the metaclass `AssociationEnd` in the UML metamodel. *Cardinality conflicts* hold for (binary) associations and compositions. *Association-direction conflicts* also directly correspond to Batini's definition of dependency conflicts [BLN86].

Definition37 (Cardinality Conflict) – A *cardinality conflict* is a *surroundings conflict*, where different *cardinality constraints* have been applied to the relationships. ◊

Figure 61 depicts selected examples of *cardinality conflicts*:

Figure 61 Surroundings Conflicts - Selection of Cardinality Conflicts



The three categories of surroundings conflicts introduced cannot completely be superimposed, since the UML generalisation has a strict one-to-one cardinality. A more detailed list of surrounding conflicts can be found in Appendix C.

7.2.3 Advanced Construct Relationships in Component Schemas

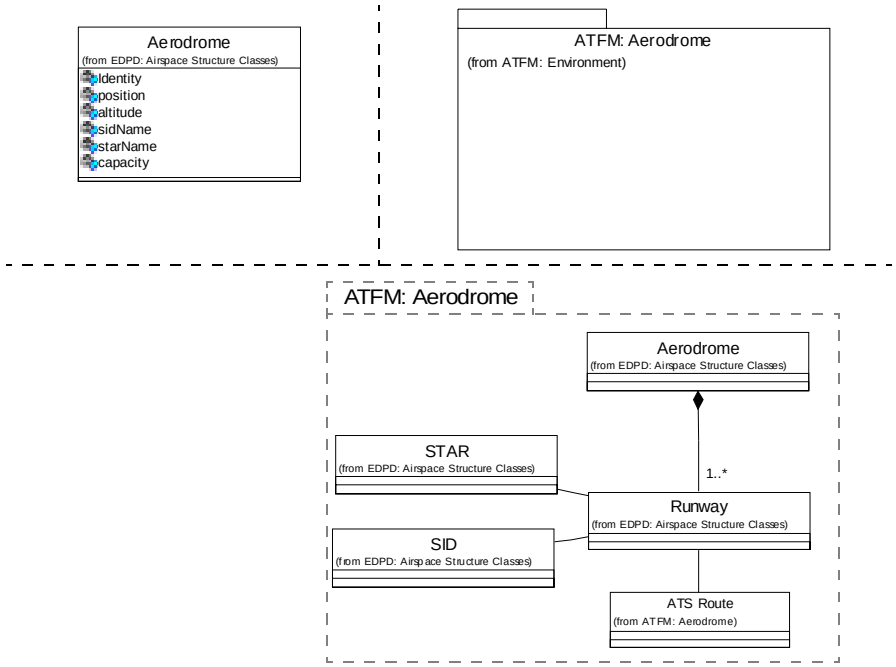
The definition of *construct conflicts* so far mostly followed the discussion in the schema integration community. *Component schemas*, however, include additional *constructs* which have not been covered yet. These *constructs* are packages, dependencies, stereotypes and constraints. Nevertheless, from a structural perspective we have to incorporate these additional *constructs* in order to cover all possible construct combinations.

Packages

Packages are irrelevant in the federated database community. They provide logical grouping of classes and therefore do not lead to instances. In *component schemas*, however, they play an important role. Packages introduce a hierarchy to industry-scale specifications. They group an arbitrary set of collaborating classes realising a coherent functionality and can be compared to subsystems in traditional modular

design. First of all, *initial partitions* are mapped to packages in *component schemas* (see Section 5.3). They can also be used to differ the modelling of given problem facets and therefore correspond to *to be resolved* requirements in the sense of Oliver et al. [OKK97]. Especially the second role is important within concurrent requirements engineering. Figure 62 depicts an example taken form the air traffic management domain:

Figure 62 Package-Class Relationship



Class `Aerodrome` in the *component schema* `EDPD` represents the *concept* aerodrome on a fairly abstract level. In the *component schema* `ATFM` a package called `ATFM:Aerodrome` details this real-world thing in terms of collaborating class specifications. These collaborating class specifications are depicted in the bottom half of Figure 62. Thinking in terms of *to be resolved* requirements and also from a cost efficiency perspective, it seems essential to track down such relationships in order to reduce the analysis efforts. Detecting class specifications and packages expressing the same *concept* also avoids the specification of redundant *artefacts* representing the same collection of real-world things in different *component schemas*.

Dependency

Dependencies in *component schemas* have been used to represent a dependency among class specifications specified in different packages. In UML this idea is often called element reference [OMG97a]. Although the UML dependency is also applicable to class specifications, we restricted their use to packages in order to exclude behavioural aspects from static structure diagrams. As an anonymous relationship, two dependency relationships are equal when they link the same packages. But the

similarity of the constituent packages can only be based on their contained class specifications. In order to avoid the creation of sophisticated metrics, we exclude dependencies from further consideration.Figure 63 depicts a simple example[1]:

Figure 63 Example of a Package Dependency

Stereotype

Stereotypes in UML provide a categorization of metamodel elements at instance level. Unlike the logical grouping of class specifications representing real-world things, the groupings introduced by stereotypes are not motivated by the problem statement. In Section 5.3, stereotypes are used to distinguish requirements related to the use interface from those related to external systems. Therefore stereotypes will be excluded from further consideration.

Constraints

Constraints allow the specification of business rules which cannot otherwise be expressed in terms of *constructs*. As a specialisation of metaclass `ModelElement` constraints have a name and comprise a logical expression in their property `body`. Constraints are logical formulas, specified in the object constraint language (OCL) [WK99]. The equivalence of two constraints is influenced by a number of factors that cannot be clarified on a linguistic level alone. We therefore exclude constraints from further consideration.

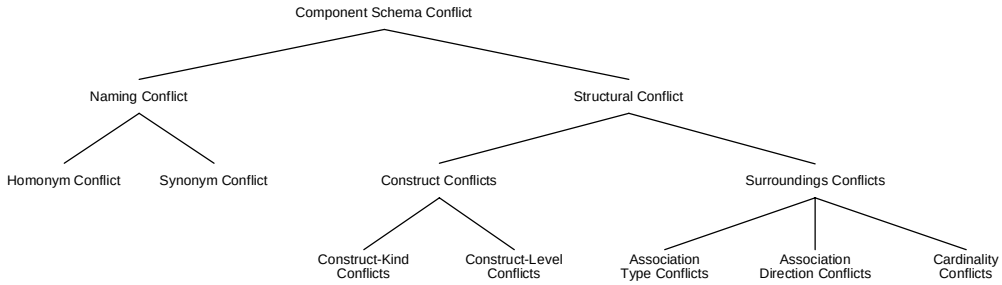
7.2.4 Subsection Summary

Traditional schema integration approaches tend to concentrate on a limited set of constructs that can be found in federated database systems, but do not sufficiently reflect concurrent requirements engineering. The selected subset of *constructs* within *component schemas*' static structure diagrams leads to the taxonomy of conflicts, depicted in Figure 64, p. 131. Most schema integration approaches follows the relational data model, with a further restricted set of constructs. So important constructs for requirements specifications remain totally uncovered, such as packages, stereotypes, constraints and dependency relationships. A comparison of OCL constraints be can excluded from this work, due to its mathematical complexity. Since stereotypes' classification of metamodel elements cannot be justified in terms of the expert language, they too are excluded. Packages and dependency relationships represent

1 Note that transparent packages – indicated by dashed lines – do not follow the UML Notation Guide [OMG97b]. UML always reserves a view in its own right, in order to present package contents to CASE tool users.

important information with respect to problem analysis. Packages represent *to be resolved requirements* and provide an abstraction for a set of related real-world things. These capabilities play an important role within problem analysis.

Figure 64 Conflict Taxonomy for Component Schemas' Static Structure Diagram



7.3 Conflict and Correspondence in Concurrent Requirements Engineering

The traditional definition of conflicts in the schema integration community relies on the assumption that the regarded *artefacts* stemming from different *component schemas* represent the same collection of real-world things. It is known to the database designers that the *artefacts* express the same *concept*. So before we can introduce a definition for *correspondence* and *conflict*, we have to reconsider the special situation as encountered in requirements analysis.

7.3.1 Particularities of Component Schema Integration in Comparison to Schema Integration

Although at first glance, schema integration and concurrent requirements engineering seem to refer to the same scientific problem, there are significant dissimilarities. While requirements engineering tries to elaborate the key abstractions forming the problem statement, schema integration targets the integration of instantiated database schemas. In schema integration the elements describing the area of discourse are known, while they are unknown in requirements engineering.

Conceptualisation has been Finished in Schema Integration

The central difference between schema integration and requirements analysis lies in the definition of the system boundary, i.e. the area of discourse. Within schema integration this task has already been finished and the available *artefacts* shall be put into a technical framework, in order to present previously disparate *artefacts* in a uniform manner to system users. Integrated metaschemas in the sense of Hammer and McLeod [HM93] or mediators in the sense of Wiederhold [Wie94b] provide technical solutions to this problem. The most important characteristic of schema integration is that the conceptualisation has already been accomplished. The area of discourse is known and agreed among the involved stakeholders.

Problem Analysis: Stabilizing the Area of Discourse

Compared to database integrators, requirements engineers still have to elaborate the *artefacts* relevant within the problem statement. The analysts do not know beforehand which real-world things are relevant within a partition and which are irrelevant. They also do not know which real-world things belong to a partition and which ones form the *partition environment*. Especially in concurrent requirements engineering, where individual analyst teams elaborate partitions simultaneously, this task becomes even more complicated. Frequent changes to the individual analysis model are likely and therefore the integration of *component schemas* become too expensive for assessing the project progress and validating the represented requirements.

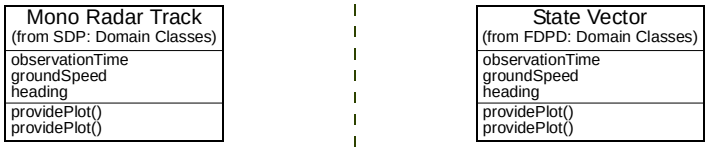
Problem Analysis: Inter-Partition Dependencies

Initial partitions and the *component schemas* representing them might depend explicitly on services provided from other partitions. This situation might not be apparent to the analysts in the affected partitions, leading to incompleteness with respect to the overall problem statement. Incompleteness due to *unsettled* requirements among partitions has to be removed, since resolving it in the implementation may become extremely expensive. Therefore *inter-partition requirements* have to tracked down as early as possible, in order to guarantee project success. Besides the constant evolution of the *component schemas* representing the individual partitions, real-world thing might be relevant in a number of partitions, leading to so-called conceptual overlaps. Unlike *inter-partition requirements*, affecting the completeness of a requirements specification, conceptual overlaps affect the minimality of the specification. Conceptual overlap means, that the same collection of real-world things has been modelled as similar *concepts* in both *component schemas*. This kind of redundancy may lead to increased design and implementation costs. In the worst case, redundancies spoil system integration and conformance testing can never be accomplished positively. The central differences between schema integration and component schema integration in concurrent requirements engineering can be summarized as follows:

- Evolving partition boundary
- Completeness: unresolved dependencies among partitions
- Minimality: redundant specification of similar real-world things in various partitions.

Concurrent requirements engineering calls for new means of detecting conceptual overlaps, currently unresolved in the schema integration community. Assertions in the sense of [LNE89, SPD92] cannot be formulated on the basis of attribute domains, due to the entirely intensional character of problem analysis models. So the potential relationships among elements stemming from different schemas require a significantly different basis compared to schema integration techniques. As described in Chapter 6, elements in a *component schema* can be regarded as a method-specific representation of an expert language and the concurrent elaboration of partitions may lead to linguistic deficiencies in the use of terms [Ort97]. *Terms* labelling different *artefacts* may be synonymous, polysemous or equipollent. Figure 65, p. 133 shows the linguistic deficiencies of concurrent requirements engineering:

Figure 65 Class Specifications Stemming from different Component Schemas



Following Batini's conflict definition [BLN86], a synonym conflict holds for the above example. Although the attributes and operations are the same, the classes carry different names and therefore a *naming conflict* holds. Taking attributes and operations as our sole criteria, the presented class specifications even seem to be identical. So the real question is: what are the criteria for detecting conflicts and hence stating or rejecting assumed conflicts? Even though misspellings of class names or operations automatically lead to conflicts in the sense of Batini, from a linguistic perspective these *artefacts* still correspond, since they express the same *concept*. On the other hand, polysemy of class names implies similarity of *artefacts*, although they represent completely disparate *concepts*. The term »bank« for instance either denotes a river bank or a financial institution. Leaving out the concrete assessment of concord or conflict of *artefacts* for a moment, this mental process still requires prior knowledge, currently not incorporated in *component schemas*. Furthermore, this knowledge is currently entirely left to the analyst's perception. This observation directly leads to the question how can analysts be supported in identifying concord or conflict of *artefacts*, even when we accept that stating or rejecting of this situation cannot be accomplished on an automatic basis alone?

Criteria for an Assessment of Artefact Similarity

Considering *structural conflicts* alone is not sufficient for assessing the correspondence of *artefacts* in concurrent requirements engineering. Naming gives rise to the inherent ambiguity of natural language. Considering structural aspects alone may lead to the situation where the *artefacts* `Man` and `Tower`, sharing the property `height`, realise the same real-world thing. Whitmire for instance proposes similarity metrics for class specifications based on a formalised algebraic object model [Whi97]. In his dissertation thesis George Spanoudakis proposes a similar approach based on TELOS [MBJK90]. Both approaches base their definition of *correspondence assumptions* on value domains and do not tackle redundantly specified real-world things.

Shortcomings of Structural Criteria

The other important question mostly deals with the pragmatics of problem analysis in industrial practice. Problem analysis models based on operational requirements documents in the air traffic management domain simply do not reach a sufficient degree of accuracy in order to precisely define the domains of attributes, as the prerequisite for Whitmire's metrics [Whi97]. Even if they would, sheer size hinders a precise formalisation of all partitions. Tool support is the other problematic aspect within this context. Popular off-the-shelf tools, such as RATIONAL ROSE™ or SELECT ENTERPRISE™, trade in formal rigor for flexibility with respect to code generation. Code-generation

should be provided for a number of different programming languages and a rigid formal model might exclude certain programming languages from this purposes and therefore threaten the tool's market success.

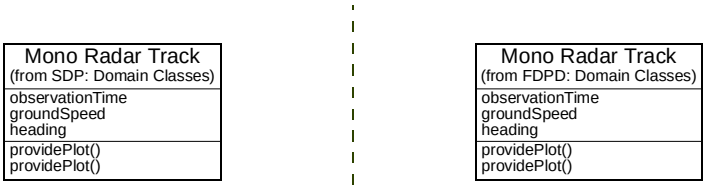
Shortcomings of Linguistic Criteria

Component schemas, however, can be considered as mostly linguistic entities, structuring relevant *terms* in a given application domain. With respect to reasoning, formalised ontologies might be an adequate support for the *conceptualisation* of the problem statement, as well as a means for assessing the similarity of *artefacts*. But once again, description logics such as CLASSIC [BBM+89] require an extensive degree of formal rigor, which once again exceeds the practicalities industrial-scale projects. At the beginning of requirements elicitation, the relevance of *terms* denoting a collection of real-world things within the application domain is not completely clear to the analysts. Synonyms, homonyms and equipollence, as deficiencies in the use of *terms*, only form one dimension within this problem. The system boundary evolves and an adequate level of detail cannot be presupposed beforehand. So the meaning of individual terms changes dramatically in particular application domains. Off-the-shelf tools mostly focus on code-generation and therefore neglect linguistic support. These tools feature neither data dictionary nor thesaurus. Especially controlling the use of *terms* is one of essential prerequisites for thorough understanding of the intricacies the problem statement. The vocabulary of thesauri is mostly difficult to expand by domain experts. Domain ontologies, even when properly build, are always restricted to a given application domain and therefore still might produce incorrect statements with respect to the similarity and dissimilarity of *artefacts*.

7.3.2 Correspondence: Relaxing the Strict Conflict Definition

The traditional conflict definition from schema integration needs a more flexible adaptation in order to suit the circumstances in concurrent requirements engineering. In this context we first concentrate on class specifications, the central abstraction for a collection of real-world things within *component schema*'s static structure diagrams.

Figure 66 Example for Identical Artefacts



According to Batini [BLN86], two *artefacts* are identical when they are labelled with the same name and have the same set of properties. Figure 66 depicts an example of identical *artefacts*. Due to the lack of a rigid formal model, we can base this definition only on the spelling of class and property names. This definition only regards

syntactical aspects and does not include the meaning of *terms* labelling *artefacts*. Using the plural instead of the singular from for a class name directly leads to a conflict. Although we have not found any example in our projects, it may also very well be the case that two class specifications with exactly the same properties represent different collections of real-world things, due to homonymous class names. From a structural perspective, we may derive other conclusions. Adding a property to a class specification automatically changes previously identical class specifications into equivalent ones. In order to properly handle this situation we need similarity metrics in the sense of Whitmire [Whi97] or Spanoudakis [SC96]. Consequently, we have to relax the strict definition of conflicts.

Correspondence

Since Batini does not further detail the exact difference between *equivalent* and *compatible* class specifications, we combine these two categories into the term *correspondence*.

> Definition38 (Correspondence) – Two *artefacts* stemming from different *component schemas* correspond, if they express the same *concept*. ◊

Their class names have to denote the same *concept* and therefore represent the same collection of real-world things. In other words, either the same class names have been used in both *component schemas* or the class names are synonyms of each other. We call the first case, where two *artefacts* share the same name, *direct correspondence*. The correspondence among names of *artefacts* being synonyms to each other is called *synonym correspondence*.

Homonyms and Correspondence

Since the meaning of individual terms varies over application domains, we also have to take into account that the *terms* are homonyms. The multiplicity of perspectives in concurrent requirements engineering in particular may lead to this situation. The *concept-context conflict* (see Figure 76, p. 149) demonstrates the impact of different perspectives on the meaning of *terms*. The problem of the varying context of *terms* cannot be solved by an *ontology*, as a logical theory describing the meaning of *terms* in a distinct domain. Consequently we cannot positively exclude that two corresponding terms are homonyms and therefore a review involving the affected analyst teams may state or reject the assumed *correspondence*.

Assertion vs. Assumption

Due to the intervention of stakeholders, we prefer the term *correspondence assumption*, since the real character of any *component schema relationship* has to be evaluated by the involved project participants. The term *correspondence assumption* also contrasts with the term *correspondence assertions*, as defined by Larson [LNE89] and Spaccapietra [SPD92]. Following the argument in the previous paragraph, a homonymy of terms cannot positively be stated by automated means. So the *term* assertion does not hold. As a side effect, the transition from an assumed to an asserted correspondence is related to the procedural perspective of the management of *component schema relationships* (see Section 8.2).

Definition 39 (Correspondence Assumption) – A *correspondence assumption* is a *correspondence* between *artefacts* that needs further review from the affected analyst groups. ◊

Definition 40 (Correspondence Assertion) – A *correspondence assertion* is a *correspondence* between *artefacts* that has been stated by review of the affected analyst groups. ◊

Although our definition of *correspondence* cannot be ensured by automatic means, additional support for analysts and project managers participating in industry-scale projects can be provided. The detection of *correspondences* has to be decoupled from their resolution and therefore avoids unintentional overriding of requirements, as a central prerequisite for analysts' autonomy.

7.3.3 Construct-Kind and Construct-Level Correspondences for Component Schema's Static Structure Diagrams

Up to this point our definition of *correspondence assumptions* relied entirely on the meaning of *terms* labelling *artefacts* and neglected the *construct*. Classifying the *correspondence* of elements into *construct-kind* and *construct-level* provides the baseline for an extensive discussion of various forms of *component schema correspondence*.

Regarded UML Constructs for a Correspondence Definition

According to Section 5.2.3, *component schema*'s static structure comprises the following constructs: `class`, `attribute`, `operation`, `package`, `association`, `association class`, `dependency`, `constraint`, `composition`, `stereotype` and `generalisation`. Within these possibilities the constructs `stereotype`, `dependency` and `constraint` have already been excluded from an assessment of *correspondence*.

Extended Correspondence Definition

UML relationships will also be included in the analysis of *correspondence assumptions*. In order to study all possibilities we take a rather straightforward approach and simply build a matrix of all relevant *constructs*. This matrix of *constructs* incorporates the complete set of possible *component schema relationships*. The taxonomy of conflicts introduced in Figure 64, provides the outline for this discussion. Then we analyse the essence of the resulting relationship taking linguistic as well as structural aspects into account. Table 10, p. 137 depicts the complete matrix of *construct*. Further examples taken from our air traffic management projects can be found in Appendix D.

The matrix of *constructs* also contains a number of illegal relationships, depicted in grey italics. Relationships between a single *class specifications* and *relationships* can mostly be excluded from this analysis due to the relational character of UML relationships. The only exception in this context is the *correspondence assumptions*

between a *named association* and a *class specification*, which may result in the creation of an association class. Another important group of excluded elements is related to dependencies, since they have been restricted to packages in this work.

Table 10 Intra-Viewpoint Construct Matrix for Component Schema's Static Structure Diagrams

	Class	Attribute	Operation	Association	Association Class	Composition	Generalization	Package	Dependency
Class	cl-cl								
Attribute	at-cl	at-at							
Operation	op-cl	op-at	op-op						
Association	as-cl	*as-at*	*as-op*	as-as					
Association Class	ac-cl	*ac-at*	*ac-op*	ac-as	ac-ac				
Composition	*co-cl*	*co-at*	*co-op*	*co-as*	co-ac	co-co			
Generalisation	*ge-cl*	*ge-at*	*ge-op*	ge-as	ge-ac	ge-co	ge-ge		
Package	pc-cl	pc-at	pc-op	pc-as	*pc-ac*	*pc-co*	*pc-ge*	pc-pc	
Dependency	*de-cl*	*de-at*	*de-op*	*de-as*	*de-ac*	*de-co*	*de-ge*	*de-pc*	de-de

7.3.4 Construct-Kind Relationships

In this section we want to analyse the relationships between class specifications, properties and packages. All constructs are labelled by a distinct name determining their real-world meaning. Following our taxonomy of conflicts – see Figure 64, p. 131 – we distinguish between *construct-kind* and *construct-level relationships*.

7.3.4.1 Intra-Construct-Kind Relationships

Although class specifications form the central real-world abstraction within object-oriented modelling techniques, we start our discussion on the rather elementary level of properties. Then we analyse the relationship among class specifications and finally detail the relationships among packages. Examples for all *intra-construct-kind relationships* can be found Appendix D.

Attributes

Attributes as structural properties of UML class specifications define their invariant structure. Attributes have a name and their potential values are defined by a data types, an instance of UML metaclass `DataType`. Data types in *component schemas* have been restricted to elementary data types such as `Integer`, `Boolean`,

`String` and `Enumeration`. Selecting only elementary data types avoids the problematic distinction between class specifications and complex data types, so called *structs* in C++, and reduces the conceptual overload of constructs in the sense of Wand and Weber [WW93]. Although enumerations cannot be regarded as elementary data types, they still are quite valuable in problem analysis, since they are often used for the definition of discriminators and for capturing state-related information. Discriminators provide a value-based distinction between individual instances and therefore may limit the depth of generalisation hierarchies. The second application of enumerations is related to the perceived state of collections of real-world things. At first glance, state-related information in UML has to be modelled in statecharts. But from a practical point of view, a simple progression of states does not justify the efforts for creating a complex statechart. Frequently, business users do not know the concrete transition of states and therefore prefer a simple enumeration representing a *to be resolved requirement*. So the simplest definition of an attribute correspondence may be formulated as follows:

Definition41 (Attribute-Attribute Correspondence) – An *attribute-attribute correspondence* holds for two attributes *expressing* the same *concept*. ◊

An *attribute-attribute correspondence* may incorporate *direct correspondence*, when the same *term* is used, or a *synonym correspondence*, when different *terms* express the same *concept*. The meaning of corresponding attribute names still incorporates significant particularities, which have to be regarded in problem analysis. The properties `airSpeed` and `groundSpeed` express the same *concept* »speed«, but their *terms of reference* differ. Due to strong headwind, air speed is usually higher compared to the ground speed, as detected by a radar system. Another example of particular differences in the interpretation of *concepts* can be found in the following example. The property `flightLevel` and `altitude` both denote similar *concepts*, however, the term »flight level« denotes the rounded hundredth fraction of the altitude. So an altitude of 33.465 ft corresponds to flight level 330. The *terms* express the same *concept*, but on a differing scale, which we call *scale divergence*. To complicate matters, the correct definition of flight level also incorporates the atmospheric pressure and therefore also has a *terms of reference* aspect. A slightly different variant can be found in the third example. The property `altitude` can be regarded as a projection of the property `currentPosition`. Despite the fact, that for this specific example both *terms* do not express the same *concept*, there is still an obvious relationship between them, which we call *value-projection correspondence.*

Attribute- Attribute Conflict

The data type assigned to an attribute restricts its potential set of values and therefore has to be incorporated into the analysis. In the most trivial case, the attributes share the same data type and further justify the correspondence. In problem analysis however, especially in its earlier phases, data types have not been defined.

Definition42 (Attribute-Attribute Conflict) – An *attribute-attribute conflict* holds for two attributes expressing the same *concept* when their assigned data types differ. ◊

Operations

Operations in object-oriented problem analysis represent behavioural requirements bound to class specifications. Depending on the analysis style, use case driven versus conventional ORDs, the granularity of operations vary significantly in the resulting *component schemas*. Following our conventions, we excluded so-called *get* and *set operations*, which are used to access the encapsulated internal representation of objects. The operation `provideTopOfClimb()` in the class `Trajectory` represents a rather high-level requirement, which in concrete implementation involves a series of rather complex calculations and consequently needs additional consideration in design. In concurrent problem analysis, however, an agreed level of detail is unlikely, due to the isolated elaboration of *component schemas*. Besides the level of detail, polymorphy results in another significant problem in *component schemas*. The central idea behind this concept is that the same operation (name) may realise a completely different behaviour, depending on its class. Since the concrete semantics of polymorphy depend on the programming language used, problem analysis using *component schemas* does not provide a sufficient formal basis to discuss its exact properties. This problem is aggravated by the fact that operation parameters often cannot be specified on a sufficient level of detail. This problem depends once again on the analysis approach applied, i.e. use case driven versus class-based, and on the individual level of detail in given *component schemas*. So we take a pragmatic approach, in order to side-step this problem.

> Definition43 (Operation-Operation Correspondence) – An o*peration-operation correspondence* holds for two operations expressing the same *concept*. ◊

Similar to attributes, this observation includes a *direct* and a synonym *correspondence* and also incorporates differing *scales* and *terms of reference*. Due to the missing parameters and return values, the definition of *conflicts* cannot be enforced in the sense of *attribute-attribute conflicts*. Projections can also be neglected, since they are related to attributes. Operations may realise this kind of projections, but always related to attributes. Projections therefore are due to requirements validation, in order to ensure the minimality of a specification and will be reconsidered in Section 8.2.

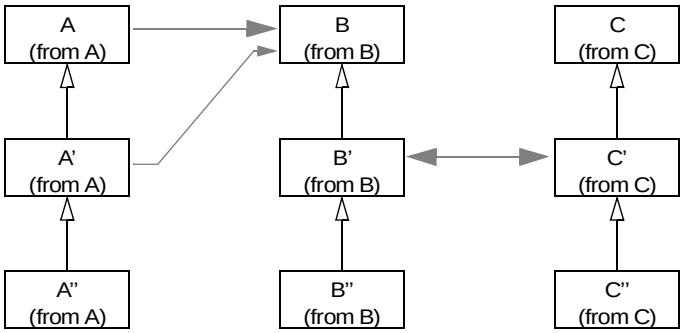
Classes

Class specifications are the central representation of collections of real-world things in object-orientation. Pointing out conceptual correspondences to the analysts provides significant support for the integration of *component schemas*, without defining complex metrics. Unlike attributes and operations, where *direct* and *synonym correspondence* are the central correspondence categories, we have a third category which stems from a deficiency in the use of terms. The same *collection of real-world things* might lead to different *artefacts* expressing different *concepts*, so-called equipollence.

The class specification SFPL_FlightPlan in the *component schema* FDPD and in the *component schema* SFPL-AirGround-Communication in the *component schema* AirGround-Communication provide an example for this situation (see Figure 123, p. 242). Both *artefacts* realise a specialised *artefact* for the concept »System Flight Plan« with respect to the various analysts' perspectives while elaborating the assigned *initial partitions*. Conflicts on the other hand can only be determined by the analysts, since the context with respect to a polysemous meaning of artefact names cannot be determined on an automated basis. So conflicts can only result from a previously detected correspondence and therefore relate to procedural aspects of correspondence management and resolution (see Section 8.2).

Definition44 (Class-Class Correspondence) – A *class-class correspondence* holds for two class specifications stemming from different *component schemas* that express the same *concept*. Two classes synonymously correspond when their *labels* are synonyms of each other. Two classes share an *equipollence correspondence* when the *artefacts* model different *aspects* of the expressed *concept*. ◊

Figure 67 Specialisation Correspondence



Class-class correspondences should also trigger an analysis of *attribute-attribute* as well as *operation-operation correspondence* for its properties. Class specifications may participate in generalisation hierarchies, which leads to additional concerns with respect to this correspondence. Specialisations normally realise specialised properties of *concepts*. From a linguistic perspective we find a hypernym/hyponym relationship. Due to its specialised perspective, this situation resembles equipollence at first glance. Unlike equipollence the hyponymy has already been expressed in the *component schema*.

Packages

Packages within *component schemas* serve two different purposes: they represent initial partitions and they logically group model elements. All top-level packages, as well as the ones labelled with the stereotype «partition» can be excluded from a

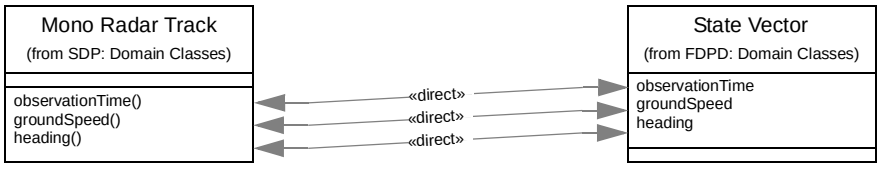
detailed analysis. These elements relate to organisational aspects within concurrent requirements engineering, such as the area of discourse of a given analyst team represented by an *initial partition*. Since the class specifications incorporated in a package realise a coherent purpose, the *correspondence assumption* can be built on their names. Due to the arbitrary decomposition of the package contents, realised through their comprised classes, a *correspondence assumption* entirely based on their *terms* has to be carefully analysed. As an additional hint, the class specifications in a given package should be meronyms, i.e. parts, with respect to the package name. Due to the mentioned lack of formal rigor of *component schema*'s static structure diagrams, we leave out the definition of similarity metrics for package contents and entirely concentrate on the meaning of package names.

Definition45 (Package-Package Correspondence) – A package-package correspondence holds for two packages stemming from different *component schemas* that express the same *concept*. ◊

7.3.4.2 Inter-Construct-Kind Relationships

In this section we analyse the *inter-construct-kind relationships* for attributes and operations, since all other *inter-construct-kind relationships* also incorporate a differing level of detail, which is analysed in Section 7.3.5. Although the purpose of attributes and operations within object-oriented problem analysis clearly distinguishes between the two constructs, the relationship between attributes and operations stemming from different *component schemas* may affect the minimality of the entire specification. We further assume that the compared class specifications do not contain simple *get* and *set* operations.

Figure 68 Example for an Operation-Attribute Correspondence



Due to the differing perspectives incorporated in distinct *component schemas*, operations in one *component schema* might correspond to attributes in another one (see Figure 68). In this context the same distinctions hold with respect to *attribute-attribute conflicts*, i.e. *direct*, *synonym*, *terms of reference*, *scale* and *projection correspondences* also apply to *operation-attribute correspondences*.

Definition46 (Operation-Attribute Correspondence) – An *operation-Attribute correspondence* holds for an operation and an attribute stemming from different *component schemas* that express the same *concept*. ◊

Since operation names must at least contain verbs, this correspondence can only be detected on an automated basis when the infinitive of the verb has the same spelling as the noun. Due to this problem an *operation-attribute correspondence* is only assessed, when the class specifications described by the respective attribute and the operation have already been deemed corresponding. Due to the potential missing data types specified for the attribute as well as the return value of the operation, conflicts cannot be found within this correspondence.

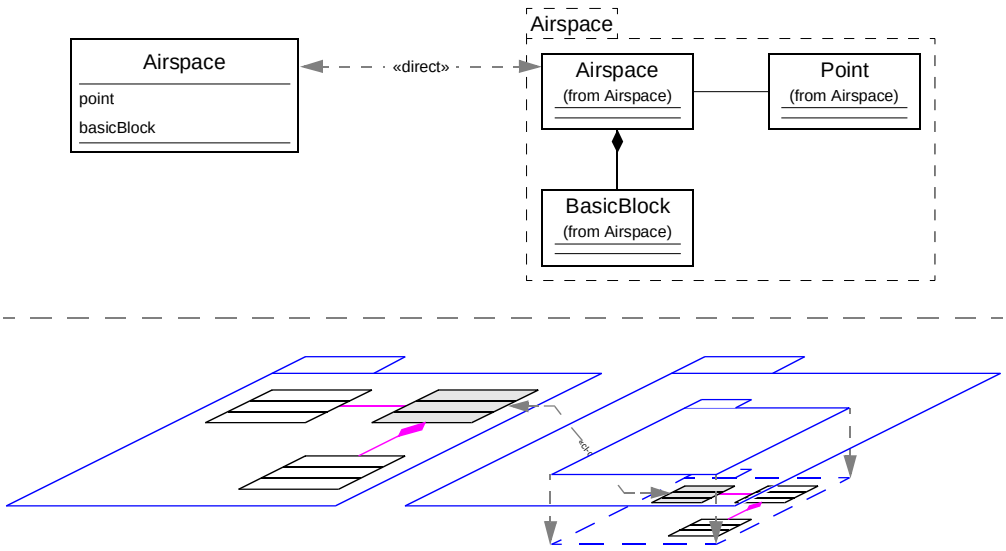
7.3.5 Construct-Level Relationships

Packages, class specifications and properties form three different levels of abstraction within *component schemas*. Packages logically group sets of classes and properties describe the structure of class specifications. Although the *terms* labelling these *artefacts* differ in their *constructs*, we want to analyse the specific impact of detected *correspondences* with respect to the integration of *component schemas*.

7.3.5.1 Intra-Construct-Level Relationships

This correspondence category only applies to class specifications residing on different package levels. Within this *correspondence* a UML construct currently disregarded plays an important role, the name space. Most *constructs* are bound to a name space, which is basically formed by packages and classes. Name spaces separate different parts of *component schemas* from each other and ensure the identity of *artefacts* comprised in the respective package. Figure 69 depicts this situation.

Figure 69 Example for a Inter-Level Intra Construct Correspondence



Although the example depicts a direct *class-class correspondence*, there is significant information not yet covered in our analysis of *conflicts* and *correspondences*. Detecting leaps in the level of abstraction, as outlined in the bottom half of Figure 69, provides interesting information for applied level of abstraction and potential imbalances in the entire problem analysis model. Unfortunately there is no clear advice for how to deal with this kind of *correspondence* in general terms. It is more like a hint for a systematic review of the corresponding class specifications with respect to their internal structure, i.e. the properties, and their surroundings. The surroundings provide interesting information with respect to the participating relationships and their applied cardinality constraints.

7.3.5.2 Inter-Construct-Level Relationships

Packages, class specifications, attributes and operations are relevant *constructs*, that may share *correspondences* over differing levels of abstraction. This section outlines the potential correspondences belonging to this category.

Attribute-Class Correspondence

Attributes and class specifications may express the same *concept* on varying levels of detail. Although from a formal perspective this claim may be criticised, for the purposes of problem analysis, especially in large-scale application domains, this situation is common. This observation also entirely relies on the meaning of *terms* and completely disregards the differing formal nature of objects and values, the runtime counterparts of class specifications and attributes. The first problem in a set-theoretic comparison lies in the difference between values and objects at instance level. Objects, as instances of classes, have an invariant identity, which is not the case for values as instances of data types. So an *attribute-class relationship* automatically leads to the problem of comparing values to objects. Due to the conceptual character of *component schemas*, this relationship is elementary for their integration.

Figure 70 Example for an Attribute Class Correspondence

Definition47 (Attribute-Class Correspondence) – An *attribute-class correspondence* holds for an attribute and a class specification stemming from different *component schemas* that express the same *concept*. ◊

Unlike the hybrid character of UML with its clear distinction between objects and values we do not distinguish between *predicators* and *secondary predicators* (see Section 6.1, pp. 103). In other words, we assume recursive relationships between *terms*. Since the class specification in an *attribute-class corresponding* offers a more detailed representation of the respective real-world thing, we remove the attribute and link the two classes via a composition.

Operation-Class Correspondence

Like attributes, the *terms* labelling operations may also correspond to class names. This kind of correspondence is rare in *component schemas*, since operations are named with verbs and class specifications with nouns. This kind of correspondence provides additional information concerning the use of class specifications in operations and therefore might identify dynamic relationships between *component schemas* from a structural perspective.

Definition48 (Operation-Class Correspondence) – An *operation-class correspondence* holds for an operation and a class specification stemming from different *component schemas* that express the same *concept.* ◊

Package-Class Correspondence

Package-class correspondences between *artefacts* stemming from different *component schemas* may indicate that real-world things have been modelled on different abstraction levels and therefore provide important information for balancing the overall problem analysis model.

Definition49 (Package-Class Correspondence) – A p*ackage-class correspondence* holds for a package and a class specification stemming from different *component schemas* that express the same *concept*. ◊

Package-Attribute Correspondence

A *package-attribute correspondence* bridges two levels of abstraction and therefore should only be used for assessing the correspondence of properties describing the *partition environment.* The stereotype «`interface`» indicates the class specification forming the *partition environment* of a given *component schema*.

Definition50 (Package-Attribute Correspondence) – A *package-attribute correspondence* holds for a package and an attribute stemming from different *component schemas* that express the same *concept*. ◊

Package-Operation Correspondence

Similar to *package-attribute correspondence*, this correspondence bridges two levels of abstraction. Therefore the operations attached to class specification representing *inter-partitions requirements* should be compared to the partition deemed responsible for its settlement.

Definition51 (Package-Operation Correspondence) – A *package-operation correspondence* holds for a package and an operation stemming from different *component schemas*, that express the same *concept*. ◊

7.3.6 Surroundings Correspondences

The central difficulty with respect to the correspondence of surroundings, is their relational character. UML relationships are based on the correspondence of its *constituent classes*, which has to be assessed beforehand. Although UML relationships are based on relational semantics, the UML metamodel realises them in terms of objects in their own right [OMG97a]. Especially the latest revision of the UML metamodel, published March 1999, introduces a distinct metaclass `Relationship` generalising the previously independent metaclasses `Association`, `Dependency` and `Generalisation` [Rat99]. This representation dramatically facilitates the discussion of *constituent class correspondence.*

Definition52 (Constituent Class Correspondence) – A *constituent class correspondence* holds for two relationships stemming from different *component schemas*, when the names of their *constituent classes* express the same *concepts*. ◊

We also have to distinguish between binary and semantic relationships, where especially the anonymous character of semantic relationships leads to our definition of *component schema conflict*. Besides cardinalities, which are not considered, we also neglect *role names*, represented in the UML metaclass `AssociationEnd` and the so-called *name-direction arrow,* only defined in the notation guide [OMG97b]. Role names usually facilitate reading recursive relationships, where both `AssociationEnds` connect the same class specification and therefore increase the comprehensibility of modelled reality for business users and domain experts.

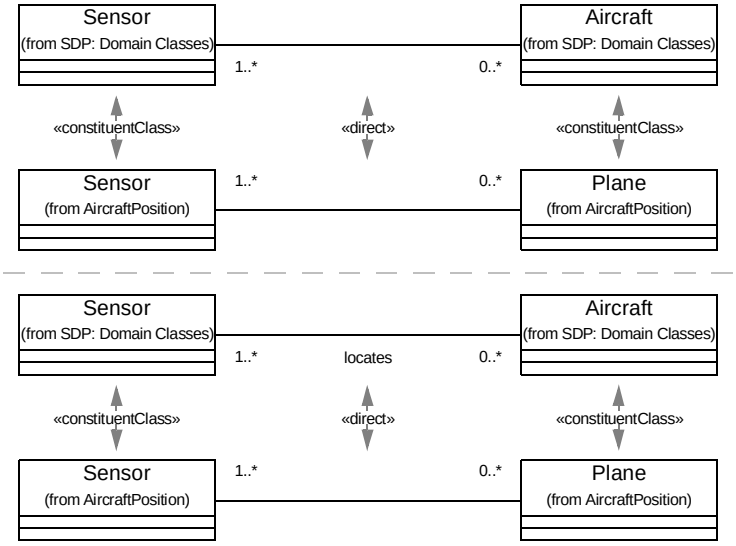
7.3.6.1 Association-Name Correspondence

Within UML relationships, only associations and association classes are named elements. Therefore these *constructs* call for an individual analysis of *correspondence assumptions*.

Association-Association Correspondence

UML associations are named model elements and therefore the *term* labelling a distinct *artefact* also has to express a *concept*. But association names are optional and therefore we may distinguish between four different situations. The particularities of the first two situations are depicted in Figure 71:

Figure 71 Association-Association Correspondence: Constituent Class Correspondence



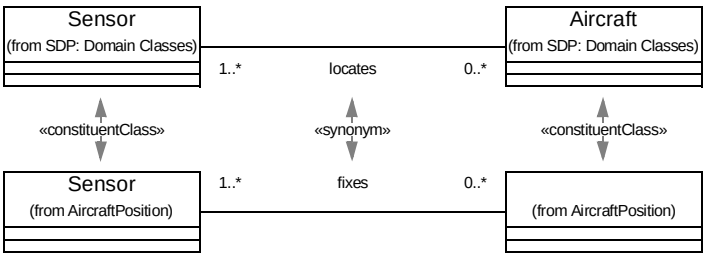
The *correspondence* of two anonymous binary associations and an anonymous and a named association can treated in a similar way, i.e. their correspondence is entirely based on a *constituent class correspondence*.

Definition53 (Anonymous-Association-Association Correspondence) – An *anonymous-association-association correspondence* holds for two anonymous associations if a *constituent class correspondence* holds for their *constituent classes*. ◊

In the case of two named associations we first have to analyse the *correspondence* of its *constituent classes* and further have to incorporate the *concept* expressed by their association names. Figure 72 depicts this situation:

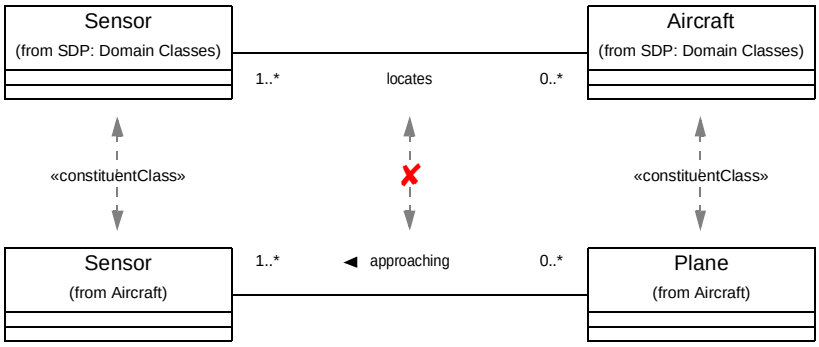
Definition54 (Association-Association Correspondence) –
An *association-association correspondence* holds for two anonymous associations if a *constituent class correspondence* holds and their association names express the same *concept.* ◊

Figure 72 Association-Association Correspondence: Association Name Correspondence



The following example refers to the situation where the association names do not express the same *concept*:

Figure 73 Association-Association Correspondence: Disparate Concepts



Although the *constituent classes* in Figure 73 correspond, different abstract belongings among the modelled real-world things have been depicted. Association names expressing different *concepts* must not be confused with *conflicts*. As depicted in our example, different *aspects* of our perceived reality have been modelled. Although the *constituent classes* will be mapped to each other in an integrated problem analysis model, two distinct associations will link them.

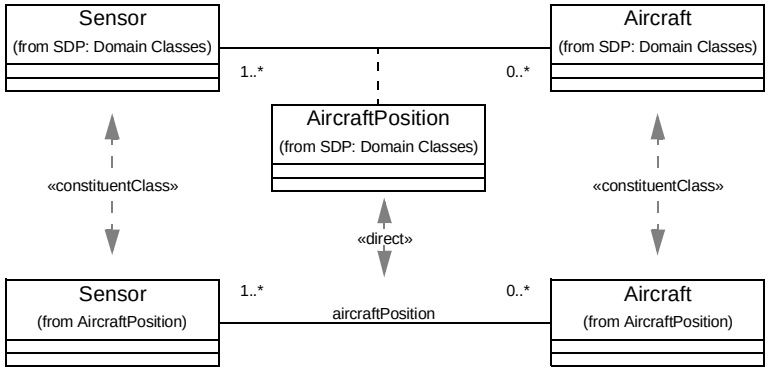
Additional Remarks

In cases where the assumed conceptual similarity has been rejected by the involved analysts, a unique name is assigned to the previously anonymous association, in order to exclude the two relationships from future *correspondence assumptions* (see Figure 73). Although the applied cardinalities provide the second criteria for the formulation of *correspondence assumptions*, as depicted in the bottom half of Figure 71, differing cardinality constraints should trigger a coordination among the involved domain leaders, in order to provide additional criteria for stating or rejecting the *correspondence assumption*.

Association Class-Association Correspondence

Within the surroundings conflicts, *association class-association correspondences* are quite complex. The most simple case is depicted in Figure 74:

Figure 74 Example of an Association Class-Association Correspondence



Following the naming conventions in Section 6.1.1.2, relationship names must be verbs. However, in the example a compound noun has been used, stressing the character of class `AircraftPosition` as a concept in its own right, with special consistency constraints, namely that instances depend on the existence of their *constituent classes*, `Aircraft` and `Sensor`. Thus the assessment of the *correspondence* entirely depends on the underlying linguistic support.

7.3.6.2 Relationship-Type Correspondences

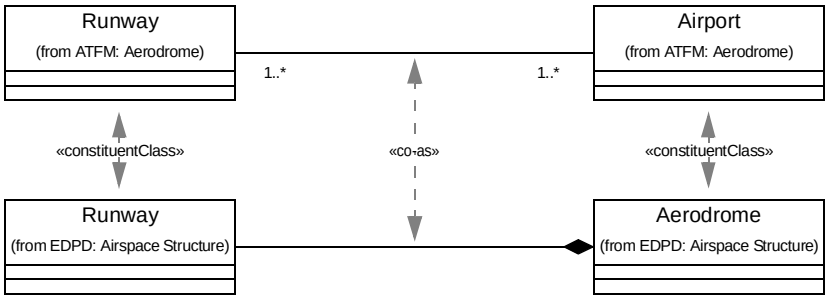
Within *surroundings correspondences* a *relationship-type correspondence* holds when their *constituent classes* correspond and different relationship types link these class specifications. This situation is especially interesting with respect to the additional constraints incorporated in *semantic relationships* such as composition and generalisation. Although the direction of semantic relationships formes an individual category of *correspondences*, we prefer do discuss its implications in conjunction with the *relationship type*.

Composition-Association Correspondence

Compared to ordinary relationships UML composition express additional constraints affecting the life-cycle of instances connected by the composition. Figure 75 gives an example from the air traffic management domain:

From a linguistic point of view, a *composition-association correspondence* holds, when the expressed *concepts* of the linked class specification are related through a holonym/meronyms relationship.

Figure 75 Example of a Composition-Association Correspondence



Definition55 (Composition-Association Correspondence) – A *composition-association correspondence* holds when a *constituent class correspondence* holds and the direction of the linguistic relationship corresponds to the direction of the applied composition. ◊

Figure 76 Example for a Composition-Association Conflict

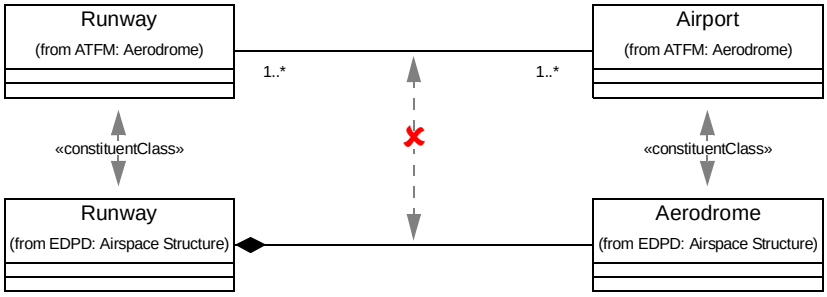
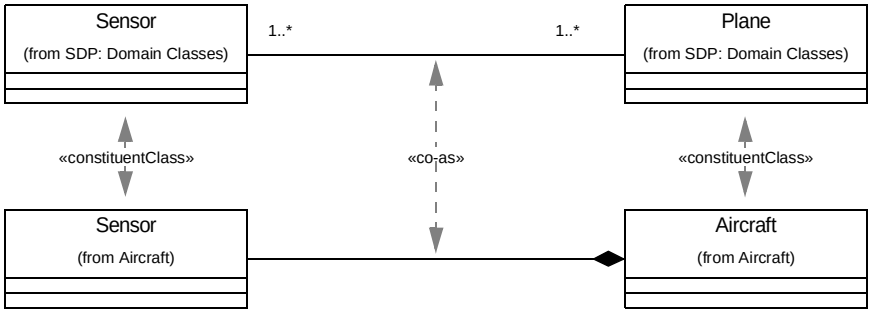


Figure 76, however, models an aerodrome as part of a runway, which contradicts the real-world perception. A domain ontology may state the *correspondence* of the classes `Runway`, `Airport` and `Aerodrome`, but will also indicate that the direction of the UML relationship contradicts the linguistic relationship. This observation leads to our definition of *conflicts, as* a new category of *component schema relationship*.

Definition56 (Composition-Association-Conflict) – A composition-association conflict holds when a *constituent class correspondence* holds and the direction of the composition contradicts the linguistic relationship. ◊

Unlike the entirely structural definition of conflicts in the schema integration community, *component schema conflicts* are based on *conceptual conflict*s. Generally speaking, *component schema conflicts* refer to contradicting directions of semantic and linguistic relationships.

Figure 77 Example of a Composition-Association Correspondence



The second example leads to a more complicated situation. The assumed correspondence in this case is based on a *constituent class correspondence* and might be asserted at first glance. The example, however, involves two completely different contexts, in which similar terms may play an important role. The classes from the package `SDP: Domain Classes` in the *component schema* `SDP` depict the observation of aircraft in the surveillance domain. The classes from the package `Aircraft`, however, depict a set of sensors which can be found in an aircraft. So the assumed *composition-association correspondence*, cannot be stated. Surveillance sensors exist without the presence of aircraft, while the on-board sensors are a crucial for an orderly conduct of a flight. This example demonstrates another conflict category, which we call *concept context conflict*.

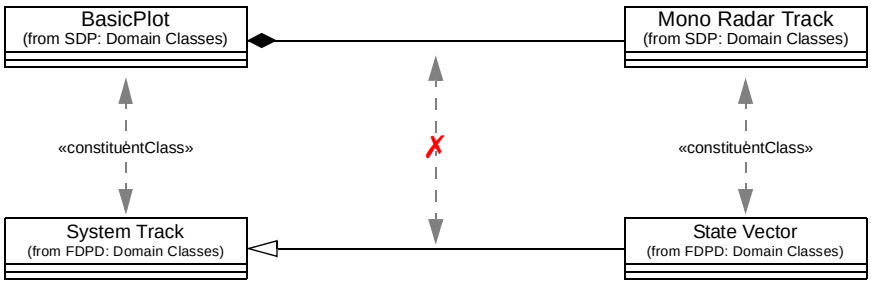
Definition57 (Concept Context Conflict) – *Concept context conflicts* arise whenever the ontological meaning of terms changes with respect to the application domain. ◊

A given *term* expresses several *concepts* and these *concept* are bound to a given application domains. So an air traffic management ontology might not include a detailed description of aircraft equipment and will therefore mistake the homonym »sensor«. *Concept-context conflicts* are related to homonymous terms and require analysts' intervention. *Concept-context conflicts* once again stress the necessity of analysts' review for stating or rejecting *assumed correspondences* between *artefacts*.

Generalisation-Composition Correspondence

At first glance Figure 78 does not require any further consideration, since *specialisations* can by no means be *meronyms* at the same time and therefore directly lead to a conflict.

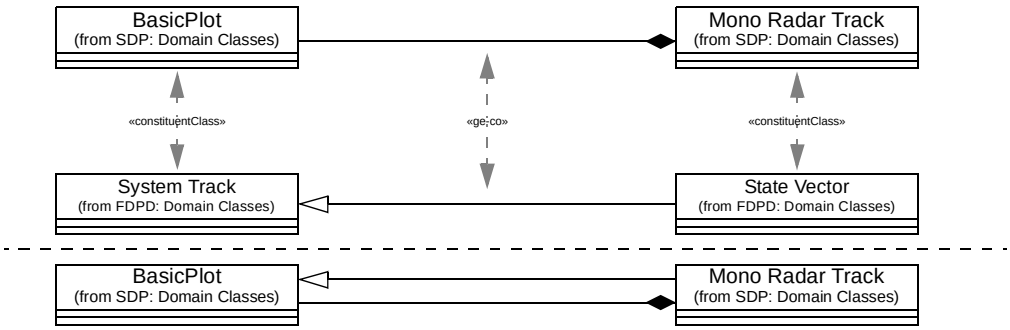
Figure 78 Generalisation-Composition Correspondence and Conflict



Definition58 (Generalisation-Composition Conflict) – A *generalisation-composition conflict* when a *constituent class correspondence* holds and the generalisation and the composition are applied in the same direction. ◊

Applying the relationships in the inverse direction leads to the example depicted in Figure 79. We call this a *compositor-pattern correspondence*, according to Gamma's [GHJ+95] design pattern. Modelling class specifications in the depicted way results in a uniform treatment of composed and simple objects. From a linguistic point of view, this distinction is quite complicated, since in most cases the generalised class does not have a counterpart in the *domain ontology*. In *component schemas* the generalised class will be represented as an *abstract class*, that does not lead to instances. If the class name has a counterpart in the *ontology* a linguistic hypernym / hyponym relationship between the class specifications can be found. Alternatively, the ontology might state a meronym / holonym relationship.

Figure 79 Generalisation-Composition Correspondence and Conflict

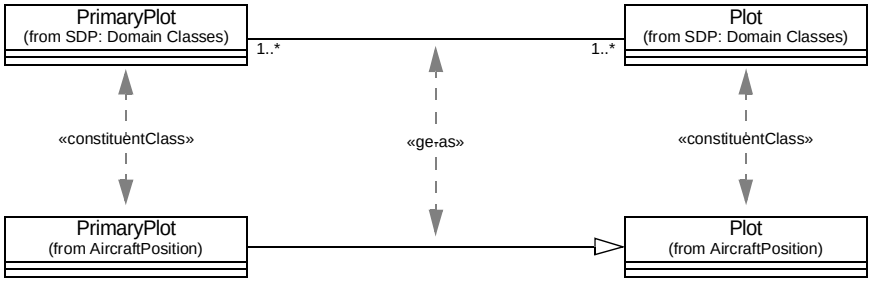


Generally speaking a *generalisation-composition correspondence* calls for careful review by the domain leaders and experts. A *generalisation-composition class correspondence* leads to another particularity. All *correspondences* described so far led to an amalgamation of corresponding *artefacts* in the resulting integrated model, while this one preserves both s*emantic relationships*.

Generalisation-Association Correspondence

In the first example, a *constituent class correspondence* holds and the *concept* »plot« is a *hypernym* of the *concept* »primary plot« in the domain ontology. As a result, the representation in the *component schema* SDP does not regard set inclusion semantics of the *ontology* and therefore proposes wrong cardinality constraints. In brief, the first representation cannot be deemed minimal, since due to the known correspondence from the ontology, a set of synonymous attribute and operation correspondences must exist.

Figure 80 Example of a Generalisation-Association Correspondence

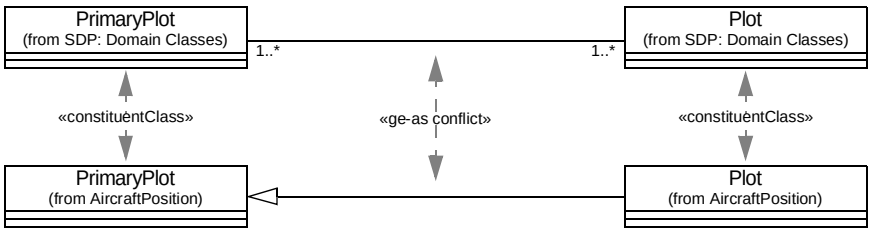


Definition59 (Generalisation-Association Correspondence) – A *generalisation-association correspondence* holds for an anonymous association when a *constituent class correspondence* holds. ◊

Due to the erroneous direction of the generalisation in our second example, a *generalisation-association conflict* can be assumed. Since the term »plot« is listed as a hypernym of term »primary plot«, a *conceptual conflict* arises in the package AircraftPosition. As a result, the minimality of the model is not assured and it may also contradict the experience of domain experts.

Figure 81, p. 153 might be mistaken for a *generalisation-association correspondence* at first glance. Since the generalisation contradicts the linguistic relationship in the ontology, this situation depicts an inconsistent static structure diagram. Consequently the domain ontologies may also ensure local consistency of *component schemas* and once again stress their specific advantages in modelling activities.

Figure 81 Example of a Generalisation-Association Conflict



7.3.7 Subsection Summary: Conflict and Correspondence

Correspondences as introduced in this section are based on the linguistic relationship among *terms* expressing the same *concepts*. Due to a potential homonymy of *terms*, i.e. a different meaning of *terms* in various application domains, this correspondence can only be assumed. Extensive review by the involved analysts has to be applied in order to state or reject the a*ssumed correspondence*. The procedural perspective towards this situation is detailed in Section 8.2. Conflicts in this context are related to UML relationships. In these cases, the linguistic meaning of *terms* labelling *artefacts* contradicts the applied *constructs* in the related *component schemas*. Neglecting an intensive analysis of these *correspondences* directly affects the minimality of specification and may lead to redundant representation of *concepts* in different *component schemas*. In the worst case, these redundantly specified class specifications are even incompatible and have the potential to threaten the implementation of the future system. These various kinds of *component schema relationships* need further consideration, with respect to procedural aspects as well as a representation of *correspondences* in terms of a metamodel. The former aspect in particular involves a set of questions with respect to the actual project organisation and the resulting management processes for the detection, coordination and resolution of *correspondences*.

7.4 Integrating Component-Schema Relationships in the UML Metamodel

Having introduces various kinds of c*omponent schema relationships,* we still have to analyse means for representing them in terms of the UML metamodel, which does not yet incorporate this kind of relationships. In order to avoid direct manipulations of component schema contents, with the risk of misinterpreting and overriding requirements, we have to identify a representation that may facilitate the coordination among the involved analyst teams.

7.4.1 Potential UML Constructs for the Realisation of Component Schema Relationships

Unlike UML relationships, that express mutual belongings among real-world things, *component schema relationships* model mutual belongings among *artefacts*.

This observation may lead to the conclusion that *component schema relationships* are metamodel elements. On the other hand, *component schema relationships* hardly determine structure and semantics of model elements, as the central objective of a metamodel in the sense of the ISO IRDS [ISO10027]. Thus changes to the UML metamodel can be excluded from further consideration. Within the selected constructs of *component schema*'s static structure diagrams, the UML `Trace` relationship provides a starting point for the realisation of *component schema relationships*. The UML metamodel states:

> "A trace is a conceptual connection between two elements or sets of elements that represent a single concept at different semantic levels or from different points of view. However, there is no specific mapping between the elements. [...] The construct is mainly a tool for tracing requirements."
> [OMG97a, p. 47]

The UML `Trace` relationship links arbitrary model elements and therefore meets the previously stated requirement that *component schema relationships* represent mutual belongings among model elements. In order to express the particular semantics of described *component schema relationships*, the UML `Trace` needs further refinement. Introducing specialised metaclasses to the metamodel or attaching stereotypes to existing metaclasses are the principle solutions to this problem.

Introducing new metaclasses threatens the compatibility with off-the-shelf tool implementations, although there is currently no CASE tool which completely realises the UML metamodel. Additional metaclasses cannot be used in off-the-shelf tool implementations. The other major drawback of specialised metamodel elements is their lack of flexibility with respect to adaptations of tool users. Stereotypes in UML directly address this issue, since they allow user-defined categorisation of metamodel elements. Consequently predefined stereotypes can be adapted by the analysts at any time, without putting CASE tool compatibility into question. So *component schema relationships* can be best realised as s*tereotyped trace relationships* in UML. The complete list of stereotypes can be found in Table 11 in Appendix D.1.

7.4.2 Component Schema Relationships: Models, Partitions and Name Spaces

Having clarified the principal realisation of *component schema relationships* in terms of the UML metamodel, we still have to take different repository organisations into account. As stated in Section 4.2, repositories persistently store *component schemas* and can be centralised as well as distributed.

Component Schema Relationships in Centralised Repositories

Using a centralised repository on the basis of *package-based partition*s (see Section 5.3) does not require any further consideration, since the repository implements the metamodel for *component schemas* and therefore arbitrary model elements can be related to each other. *Package-based partitions* ensure the isolation of the assigned analyst teams through the UML *name space concept*. Class specifications with the same name in different packages are considered as independent *artefacts*,

since their name is automatically prefixed with the name of the package-hierarchy. *Correspondence assumptions* may link *artefacts* stemming from different *component schemas*. So centralised repositories allow an efficient implementation of *component schema relationships* in terms of the UML metamodel.

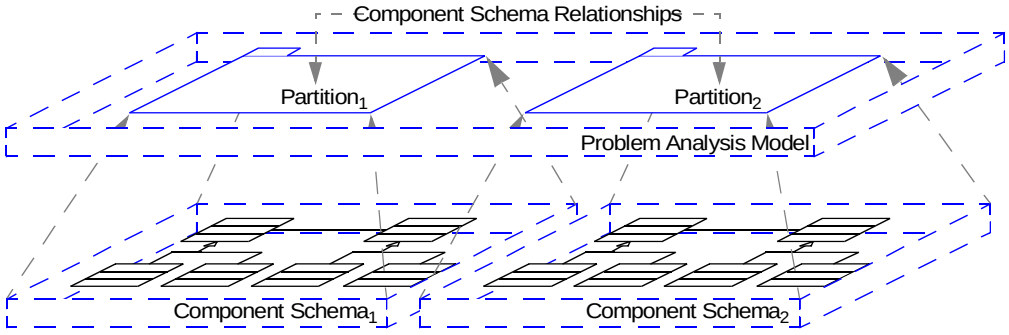
7.4.2.1 Distributed Repositories With Package-Based Partitions

Distributed repositories restrict the access to *component schemas* for analysts who do not belong to the *partition owners*. This situation gives rise to the question of how an integrated perspective to a set of independent but related *component schemas* can be achieved.

Creating an Integrated Name Space

As discussed in Section 5.2.3.3, the identity of elements in a *component schema* is realised through the UML name space concept. The hierarchic organisation of name spaces automatically guarantees the identity of arbitrary model elements in *component schemas*. Consequently, creating a new name space, provides a basis for an integrated perspective towards a set of related *component schemas* and may also incorporate the UML trace relationships representing the *correspondence assumptions*. A new model called *'problem analysis model'* hosts the set of *component schemas* in a *package-based* style. So each initial partition has to be represented as a distinct UML package carrying the partition name. This situation is depicted in Figure 82:

Figure 82 Creation of an Integrated Name Space for the Definition of Correspondence Relationship



Providing read-access to the individual partitions stored in distributed repositories is the minimum prerequisite for this approach. Integrating *component schemas* in a newly created '*problem analysis model*' creates a new name space incorporating all *initial partitions*. But this gives rise to another problem, only the newly created '*problem analysis model*' incorporates *component schema relationships* and they cannot be replicated to the underlying *component schemas*. So read-access for the newly created '*problem analysis model' has* to be provided to all involved analyst teams, in order to enable notification of detected *component schema relationships*.

Model-Based Partitions

Model-based partitions with their single name space require additional preparations. The stereotype «interface» has been tagged to all class specifications forming the partition environment, but there is no distinction between class specifications representing *inter-partition requirements* and class specifications referring to *external systems*. So first of all, *model-based partitions* have to be converted into *package-based partitions.* Then class specifications representing *inter-partition requirements* have to be related to those packages deemed responsible for the settlement of the specified *inter-partition requirements*. Finally a new '*problem analysis model*' may be created and *component schema relationships* will then be detected in the same style as described for *package-based partitions* at the beginning of this subsection

7.4.2.2 Distributed Repositories Using the CORBA OOA & D Facility

With the standardisation of the Unified Modelling Language in the Object Management Group (OMG) a set of specialised interface definitions have been defined for CASE tools realising distributed repositories. As part of the common facilities of OMG's Common Object Request Broker Architecture (CORBA) [OMG98] the so-called *OA&D CORBA Facility* [OMG97d] allows the exchange of model elements among distributed repositories. Interfaces to the repositories themselves are specified in the related Meta Object Facility (MOF) [OMG97e]. Without prescribing a specialised implementation, all OMG standards standardise the access to the specified facilities through a set of interfaces descriptions. Although there are currently no off-the-shelf CASE tools implementing the interfaces, they still provide an interesting approach for future enhancements of tool implementations.

With respect to *component schema relationships*, however, the OMG standards provide the necessary infrastructure for concurrently elaborated *component schemas*. UML model elements are now accessible across the network. *Component schema relationships* based on stereotyped Trace relationships can now directly link model elements in distributed repositories. So using CORBA as an infrastructure for realising *component schemas* closely matches the ideas of Nuseibeh [NKF94], where ViewPoints are distributed objects, encapsulating *style, workplan, domain, specification* and *work record*. Furthermore, the CORBA event management service [OMG97f] also provides the necessary infrastructure for the management of *component schema relationships.*

Using the CORBA infrastructure is applicable to both *model-* as well as *package-based partitions*. Read access to all involved *component schemas* and write access for *component schema relationships* is the minimal prerequisite for this approach. The self-contained character of concurrently elaborated *component schemas* will further be stressed. The quality of the resulting overall problem statement, however, depends on the management of *component schema relationships.* Resolving *asserted correspondences* and guaranteeing the minimality of the overall problem analysis model depends on the cooperation among the involved analyst teams, and consequently can only be supported by an adequate technical infrastructure.

8 Management of Component Schema Relationships

The management of *component schema relationships* comprises all activities related to the detection, propagation and resolution of *component schema relationships*. While the first aspect focuses on the strategies for their detection at a technical level, the latter aspect involves organisational aspects. As detailed in Section 8.2, the detection of *correspondence assumptions* is influenced by the project organisation model. The analyst's horizon determines the visibility of *component schema* contents to other involved analyst teams. The specific representation of *inter-partition requirements* also influences the different kinds of *correspondence assumptions* that can be found. Finally, we have to describe the process of detecting, propagating and resolving *correspondences* and *dependencies* among a set of concurrently elaborated *component schemas* representing the overall problem statement.

8.1 Assessment of Component Schema Relationships

Correspondence between *artefacts* and *link artefacts* differs significantly, since similarity for *link artefacts* also includes the *constituent class correspondence*.

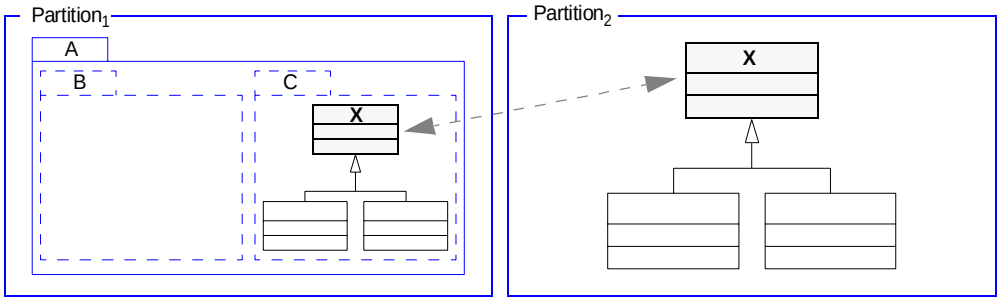
8.1.1 Assessing Correspondence for Artefacts

Assessing the correspondence of model elements stemming from different *component schemas* demands an integration of linguistic as well as structural aspects of the applied modelling technique. Based on the distinction of *correspondence* and *conflict* in Section 7.3, pp. 131 we define *artefact correspondence* as follows:

Definition 60 (Artefact Correspondence) – An *artefact correspondence* holds for two *artefacts* stemming from different *component schemas*, if their *terms* label the same *construct* and express the same *concept*. ◊

Valid constructs within *component schemas'* static structure diagrams are `Class`, `Attribute`, `Operation` and `Package`. The *concept* expressed by a *term* refers to the linguistic correspondence, while the applied *constructs* determine the respective *component schema relationship* in the sense of Section 7.3. A class name and a package name expressing the same *concept*, for instance, lead to a *package class correspondence*. *Construct-kind* and *construct-level correspondences* therefore entirely depend on the applied *constructs* of the two corresponding *artefacts*. Especially *intra-construct-level correspondences* demand further clarification. An *intra-construct-level correspondence* holds for two class specifications determining the same *concept* that reside in different package levels. Figure 83 depicts this situation:

Figure 83 Inter-Construct-Kind Correspondence



Origin of Artefacts

The *correspondence* of *artefacts* stemming from different *component schemas* also demands keeping track of their origin. Here once again the UML name space concept provides the necessary information, since all class specifications are unique within their name space. The name space is spanned by packages. Class specifications stemming form different partitions are prefixed with the individual names of the package hierarchy. Every *component schema* is modelled in a top-level package carrying the stereotype «`System`». In order to distinguish the «`System`» packages from different *component schemas*, we have to prefix the name space with the partition name. Figure 83 depicts the two class specifications named `X` stemming from two different partitons. Since UML name spaces are spanned by packages, we come to the following situation:

```
Partition1::System::A::C::X
Partition2::System::X
```

As a result, the UML name space concept allows us to keep track of the origin of *artefacts* in distributed as well as centralised repositories. Although *artefacts* do not have to be unique for the detection of *correspondence assumptions* as such, it is necessary for correct linking of the affected *artefacts* via stereotyped `Trace` relationships. Such a stereotyped `Trace` relationship represents the *correspondence assumption* in the repositories.

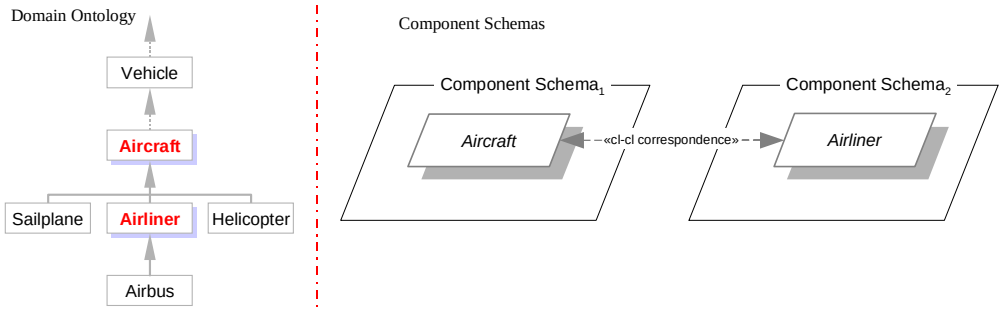
So our current definition of *correspondence* depends on a linguistic clarification of the meaning of *artefacts* used in *component schemas*. In other words, we are looking for *concepts* expressed by *terms* labelling *artefacts*. One source for this clarification are linguistic databases such as WordNet [WN98] or ontologies such as SENSUS [Sen95]. Besides predefined linguistic support, analysts may also clarify the meaning of *terms* manually by providing a concise definition for each *term* and relating synonyms to them.

Hypernyms and Correspondence Assumptions

As mentioned in Section 6.3.1 (see p. 110) the *concept lattice* of lexical databases and ontologies, may further contribute to the detection of *correspondence assumptions*. In this context we are especially interested in the *hypernym* / *hyponym* relationships between *terms* labelling *artefacts*. These linguistic relationships gain special importance for the detection of *class-class correspondence assumptions*. We take the

terms »aircraft« and »airliner« as our example. Obviously both *terms* cannot be synonyms, due to their conceptual difference. However, the term »aircraft« is a hypernym of the term »airliner« in the domain ontology. So without further considering the structural aspects, i.e. attributes and operations, we may assume that the domain owners just chooses a more specific *term* as is in the other *component schema*. This situation is depicted in depicted in Figure 84:

Figure 84 A Class-Class Correspondence Assumption based on Hypernyms in the Domain Ontology



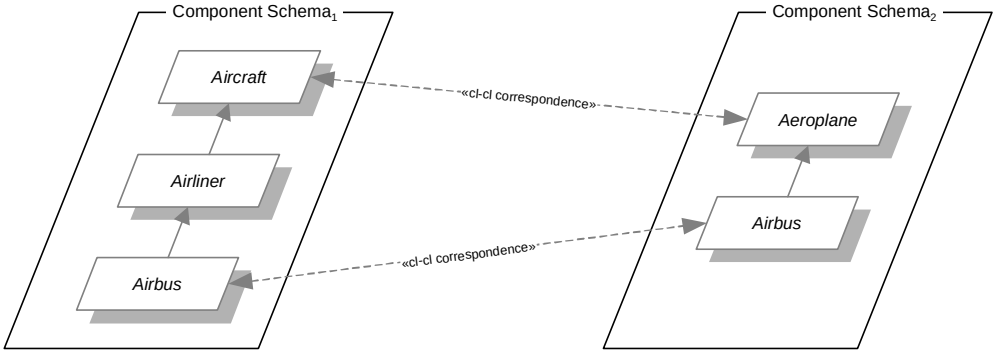
Using hypernyms/hyponyms for the detection of *correspondence assumptions* gives rise to another question, an assume *correspondence* of two *terms* that have a common hypernym, as in the case of the terms »sailplane« and »helicopter« in the domain ontology of Figure 84. Kashyap and Sheth argue that synonyms among independently developed ontologies are infrequent [KaS98]. Therefore incorporating common hypernyms to generation of *correspondence assumptions* seams quite attractive. First of all we have to decide the levels of hypernyms to be incorporated to the generation of this kind of *correspondence assumption*. Taking once again the domain ontology of Figure 84 as our example, we also may consider the term »vehicle« as a hypernym of the term »aircraft«. Since the term »vehicle« also subsumes trucks and vessels, we restrict the detection of *correspondence assumptions* only to direct hypernyms. Otherwise the related real-world things become too different in order to keep the *correspondence assumption* meaningful for the analysts. Incorporating hypernyms to the detection of *correspondence assumptions* may lead to the situation to look for a direct common hypernym of two *terms*. Although the terms »sailplane« and »helicopter« have the same hypernym in the term »aircraft« they very different from a conceptional point of view. So including the common hypernym of two *terms* to the detection of *correspondence assumptions* is critical with respect to difference of the real-world things. As a matter of consequence we do not make use of this linguistic relationship for the generation of *correspondence assumption*.

Integration of Concept Lattices

The *correspondence assumptions* between the *component schemas* in Figure 85, p. 160 give rise to another interesting question, the integration of different class hierarchies. First of all there are two distinct perspectives with respect to this problem. The integration of the resulting class hierarchies and a possible integration of ontologies.The integration of the class hierarchies is basically determined by the application

and hand and therefore has to be considered as a manual process conducted via a *coordination* between the affected *domain owners* and therefore will remain outside the scope of this work. A possible integration of the ontologies, however, requires further considerations.

Figure 85 Class-Class Correspondence Assumptions and Generalisation Hierarchies



First of all the *correspondence assumptions* in Figure 85 is based on a synonym relationship between the terms »aircraft« and »aeroplane« in the domain ontology (see p. 159). In this context we do not intend to alter the underlying domain ontology, we leave it as-is and initiate *coordination* among the involved *domain owners*. Assuming there is a need for integrating new *terms* into an existing domain ontology, description logics are able to classify a given term automatically in the subsumption hierarchy. Kashyap and Sheth [KaS98] use this capability of description logics for query processing in federated databases and therefore prove the minimality of a given query. However, the integration of ontologies is not the concern of this work. In order to avoid unintentional overwriting of *artefacts* we are looking for suitable means for the detection of *correspondence assumptions*. We are using the ontology in this context and we are not detailing how an ontology may be built from *component schemas*. Therefore we leave out this interesting discussion in this work.

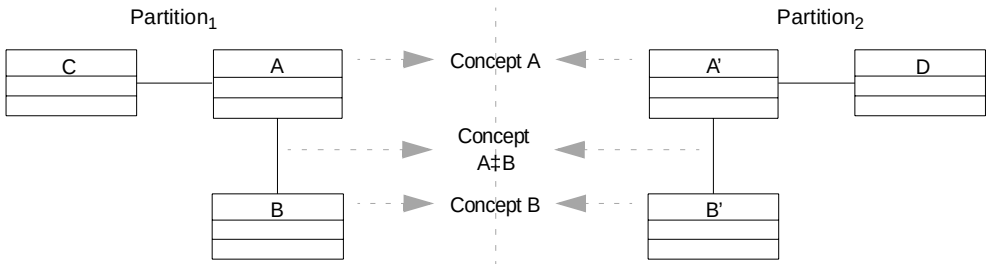
But there are other kinds of *correspondence assumptions* that cannot be detected on the basis of linguistic support alone. WordNet for instance, lists the concept »speed« as a hypernym for the terms »groundspeed« and »airspeed«. But following our distinction in Section 7.3.4.1, these two *terms* have different *terms of references*, which cannot automatically be derived from WordNet. Consequently *terms of reference*, *scale*, *projection correspondences* have to be considered as refinements of *correspondence assumptions*, that can only be assigned manually. Although the *concept* may reveal a correspondence among *terms* labelling different *artefacts*, analysts' background knowledge may even refine this distinction. Especially a difference in scale stresses this particular problem. Taking the terms »altitude« and »flight level« as our example, WordNet [WN98] lists the term »level« – »flight level« is not listed – as a hyponym of the term »altitude«. Consequently, standard linguistic databases do not cover all intricacies of the domain at hand. Only detailed domain ontologies with

their encoded logical theory are able to distinguish between these subtle differences. Automatic assessment of correspondence for *artefacts* may be further detailed by analysts' intervention. But this distinction requires extensive domain expertise, that is usually not encoded in standard thesauri and lexical databases.

8.1.2 Assessing the Correspondence for Link Artefacts

Link artefacts depend on *artefacts* and are represented in *component schemas*' static structure diagrams. Only named binary relationships are covered, since we may retrieve the *concept* denoted by their association names. But the correspondence of their *constituent class specifications* will be neglected and often binary relationships do not carry a name. To complicate matters, semantic relationships – generalisation and composition – are anonymous and therefore their correspondence cannot be assessed in the same way. The directional character of composition and generalisation in UML directly leads to our distinction of *correspondence* and *conflict*. Whenever the direction of a linguistic relationship contradicts the applied direction of a UML relationship in a *component schema*, we call this situation a *conflict.* According to the UML metamodel, all relationships are represented as metaclasses and therefore lead to *artefacts* in their own right. However, the first step in assessing the correspondence of relationships is the assessment of correspondence of their constituent classes. Figure 86 illustrates an example for the remainder of this subsection:

Figure 86 An Example of Artefact Correspondence



Two *artefacts* stemming from different *component schemas* correspond if they are labelled with the same name and express the same *concept*. With respect to Figure 86, the class specifications `A`, `A'` and `B`, `B'` correspond to each other since their class names *denote* the same *concepts*. The class specifications `C` and `D` do not correspond to each other, since their *terms* do not express the same *concepts*. Relationships' *constituent classes* can be regarded as pairs of *concepts*. In other words two anonymous binary relationships correspond, when the particular *concepts* of their *constituent classes* correspond. Consequently constituent class correspondence can be broken down to the concept pair of the constituent classes. So we simply have to form a new concept, based on the concatenation of the constituent classes' concepts.

Definition61 (Concatenated Concept) – The concatenated concept for relationships is constructed from the ordered pair of *concepts* of its constituent classes. ◊

In this context, we only have to make sure that the newly constructed *concept* cannot be mixed up with existing ones. Therefore we introduce a double dagger '‡' as an indicator for *concatenated concepts* denoting relationships. Depending on the specific CASE tool implementation, binary associations may lead to two different concepts, with a varying order of *concepts*. To facilitate the correspondence assessment for relationships, we simply build an alphabetically ordered pair of their constituent classes concepts.

Definition62 (Relationship Concept) – The concept of a relationship is the alphabetically ordered pair of its *constituent classes' concepts*, separated by a double dagger ('‡') ◊

Going back to our example in Figure 86, the resulting concept for the relationship linking the *artefacts* A and B takes the form A‡B. Similar to the correspondence of *artefacts*, we also have to take the *construct* applied into account, i.e. the kind of UML relationship. Possible values for the *construct indicator* in this context are `association`, `associationClass`, `generalisation` and `composition`. So far, *artefacts* and *link artefacts* share the same data structure for the assessment of their similarity, but differ in the concept generation. The *concepts* of *artefacts* can be directly retrieved from a domain ontology, while a newly created *concatenated concept* is used for relationships.

8.1.2.1 Assessment of Correspondence for Named Link Artefacts

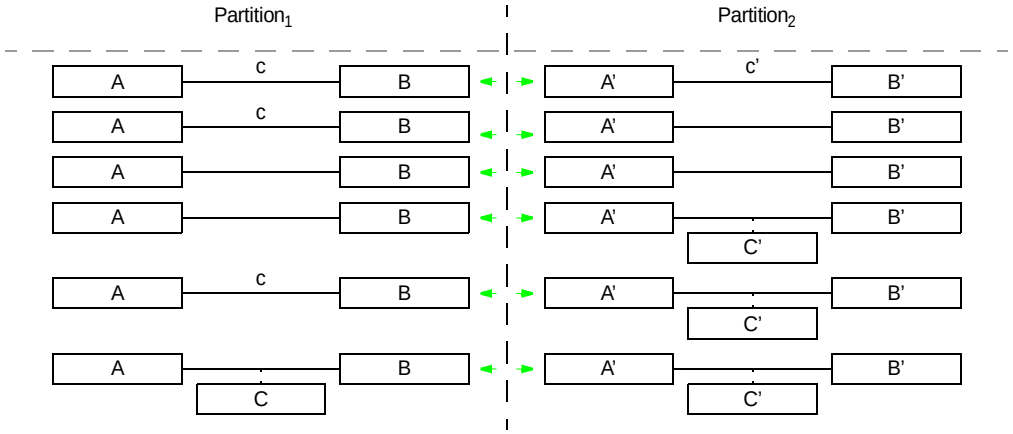
Named link artefacts in this context are binary relationships in UML that take the form of associations and association classes. We further detail the correspondence between anonymous binary relationships and named UML associations as well as association classes. Nevertheless, the entire discussion in this subsection is based on the *constituent class correspondence*. Unlike *artefacts*, *named link artefacts* comprise two distinct *concepts*. On the one hand, the *concatenated concept* formed by its *constituent classes* and on the other hand the *concept* denoting the relationship name. In order to provide a uniform treatment of *artefacts* and *link artefacts* we want to form a single concept.

Definition63 (Named Relationship Concept) – The concept of a relationship is the ordered concatenation of its constituent classes' concepts and the *concept* expressed by the relationship name, separated by double daggers ('‡'). ◊

This newly created concept incorporates the *relationship concept* combined with the *concept* of the relationship name. So we simply concatenate the concept of the relationship name to the constituent classes concept, separated by another double dagger.

Taking the uppermost occurrence in Figure 87 as our example, we obtain the following concept ‘A‡B‡C’. Constituent class correspondence can therefore be realised as a simple substring search, while the assessment of named relationships encompasses the entire concept. Figure 87 depicts the list of assessable correspondences:

Figure 87 Assessable Correspondences for Named Link Artefacts



This definition holds for UML associations, as well as UML association classes, since the central difference between the two UML concepts is formed by properties exclusively belonging to the relationship. With respect to association classes, another significant problem comes into play. Following the conventions in Section 6.1.1.2, verbs name binary associations, while nouns name association classes. Consequently domain ontologies will not be able to assess their corresponding denotation. Ontologies as well as thesauri use disparate hierarchies for the clarification of the individual meaning of verbs and nouns. The OntoSaurus [Ont95] and its underlying SENSUS ontology [Sen95], as well as WordNet [WN98] follow this approach. Since our assessment of correspondence is not entirely based on automated means, analysts may indicate this kind of correspondence manually. The previous definition also holds for anonymous binary relationships with little modifications. In this case, the *constituent class correspondence* already holds for the correspondence between an anonymous relationship with a named association. So we only have to make sure that a single anonymous relationship in one *component schema* will not be related to several named ones in the related *component schemas*. In this situations we arbitrarily select one of the named relationships in the target *component schema*.

8.1.2.2 Correspondence and Conflict for Anonymous Link Artefacts

Unlike the previous subsection, where the correspondence of *link artefacts* is essentially influenced by the relationship name, the correspondence of anonymous and semantic relationships entirely relies on the linguistic relationship of the concepts of

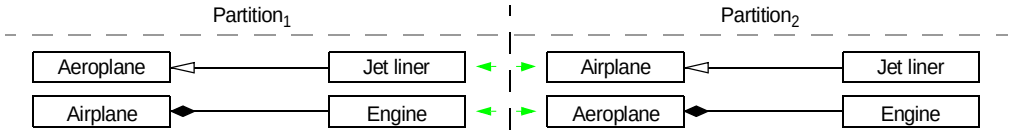
their constituent classes. From the modelling technique perspective, semantic relationships impose additional integrity constraints on the resulting instances of the linked classes. Specialised class specifications are subsets of their generalised class specifications. Compositions impose a coinciding lifetime of the aggregate on its components. In brief, composition and generalisation impose constraints to instances, which are not part of our analysis in this work. But on the linguistic level, these relationships express significant additional meaning with respect to their constituent terms. Generalisation is mirrored in the hypernym relationship and composition in the meronym relationship. So this time the relationship among the concepts of the constituent classes forms the centre of our interest. But the directional character of anonymous *link artefacts* demands for preservation of the original direction of the relationship, since composition and generalisation are directional. In order to keep the comparison of concepts as simple as possible, we keep the alphabetic order of the concatenated concept formed by its constituent classes and add a simple direction indicator, preserving the original direction of the original UML relationship.

Definition64 (Direction Indicator) – The direction indicator states whether or not the order of the *constituent classes' concepts* corresponds to the original direction of the semantic relationship. ◊

Based on this observation we start with a definition of correspondence for anonymous *link artefacts*.

Definition65 (Anonymous Link Artefact Correspondence) – An *anonymous link artefact correspondence* holds when a *constituent class correspondence* holds the *anonymous link artefacts* have the same *constructs* and the same *direction indicators.* ◊

Figure 88 Assessable Correspondences for Anonymous Link Artefacts



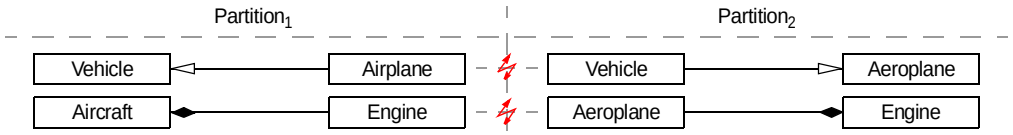
This definition holds for the correspondence between two specialisations and compositions, indicated with the stereotypes «co-co correspondence» and «ge-ge correspondence». The central characteristic in this assessment is the availability of suitable linguistic support, in order to verify if the applied UML construct – generalisation and composition – reflects the linguistic relationship between the *artefacts* participating in the relationship. Essentially we have to verify whether the two terms are hypernyms or meronyms of each other. Figure 88 depicts examples of this situation. Where the direction of the linguistic relationship contradicts the applied direction of the UML relationship, we find a *component schema conflict.*

Definition66 (Generalisation Conflict) – Two generalisations conflict, when their constituent classes determine the same concepts and the direction indicators differ. ◊

Definition67 (Composition Conflict) – Two compositions conflict, when their constituent classes determine the same concepts and their direction indicators differ. ◊

However, automatic detection of this kind of *component schema conflict* depends on the available linguistic support. The domain ontology used has to implement all inverse linguistic relationships. For a hypernym relationship between given terms, the inverse hyponym relationship should also be implemented. But this demand is quite critical with respect to the meronym relationship. Neither the SENSUS Ontology [Sen95] nor WordNet [WN98] implement the holonym relationship for *all* terms. But meronymy can simply by verified with a second query to the ontology with a reverted order of terms expressing the constituent concepts. Examples of these conflicts are depicted in Figure 89.

Figure 89 Assessable Conflicts for Anonymous Link Artefacts



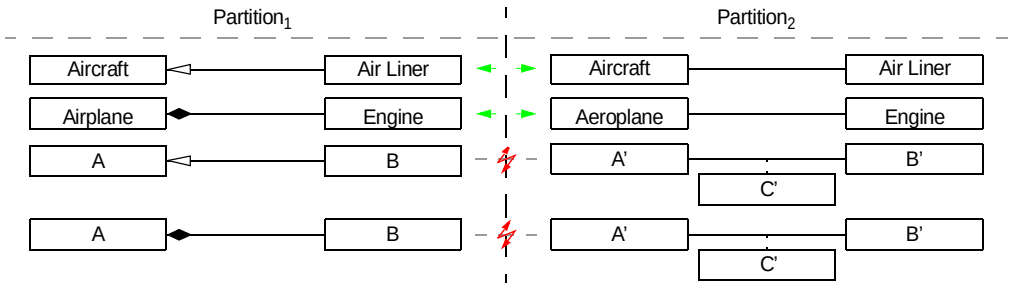
Linguistic support may also reveal linguistic relationships that have not been modelled in individual *component schemas*. While in one *component schema* a binary association links the two class specifications, a composition links the class specifications in the other *component schema*. But linguistic support does not hold for association classes, since properties cannot be attached to semantic relationships in *component schemas*. Whenever two corresponding constituent class specifications have been linked with a UML association class in one *component schema* and a composition in the other one, this leads to a «co-ac correspondence». A similar situation holds for generalisations and associations classes. So this observation leads to the following definition of conflicts:

Definition68 (Association Class Conflict) – Two link artefacts conflict, when a constituent class correspondence holds and an association class is linked to semantic relationships. ◊

So finally a conflict arises automatically when a composition and a generalisation have been applied in the same direction in different *component schemas*. A *term* can never be a meronym and a hypernym of the same concept at the same time. With

respect to our example, the term »aircraft« is a hyponym of »vehicle«, but »vehicle« is never a holonym for the term »aircraft«. So this situation results in a clear *composition-generalisation conflict*, indicated through the stereotype «co-ge conflict».

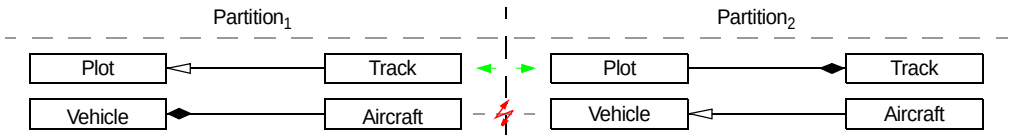
Figure 90 Assessable Correspondences and Conflicts of Different Types of Anonymous Link Artefacts



Definition69 (Compositor-Pattern Correspondence) – A composition and a generalisation correspond, when their *constituent classes* determine the same *concepts*, the *direction indicators* differ and hypernomy holds for the generalised terms and meronymy holds for composed terms. ◊

Definition70 (Composition Generalisation Conflict) – A composition and a generalisation conflict, when their *constituent classes* express the same *concepts* and the same *direction indicator*. ◊

Figure 91 Assessable Conflict and Correspondence of Different Types of Anonymous Link Artefacts



However, this situation requires a very well chosen domain ontology, since usually the hyper-/hyponomy relationships are implemented on a broader range in different linguistic tools. So it is more likely that the linguistic support will state the hyper-/hyponomy relationship between terms. So we prefer to indicate *compositor pattern correspondence* instead of indicating a conflict.

8.1.3 Subsection Summary

The central prerequisite for the assessment of relationship correspondence is a correspondence of their *constituent classes*. Through the creation of a so-called *concatenated concept* formed by the *concepts* of its *constituent classes*, the similarity can be detected with simple pattern matching. A clarification of the real nature of the *corre-*

spondence assumption, however, incorporates the reflection of the individual UML *constructs* and the *direction indicator*. Most of the automatic support is heavily influenced by the level of detail in the available domain ontology. Extensive concept hierarchies and meronym / holonym pairs guarantee broad automated detection of relationship correspondences and conflicts. In this context, a set of queries to the ontology may answer the detailed relationship among the terms of the *constituent classes*. The concrete nature of these queries depends on the *construct* and the direction of the compared relationships. The assessment of correspondence for UML compositions involves two ontology queries, since a holonym relationship has not been implemented for all terms. Since the UML metamodel regards all kinds of relationships as model elements in their own right, the introduction of the so-called *concatenated concept* allows uniform treatment of *artefacts* and *link artefacts* for the assessment of correspondence.

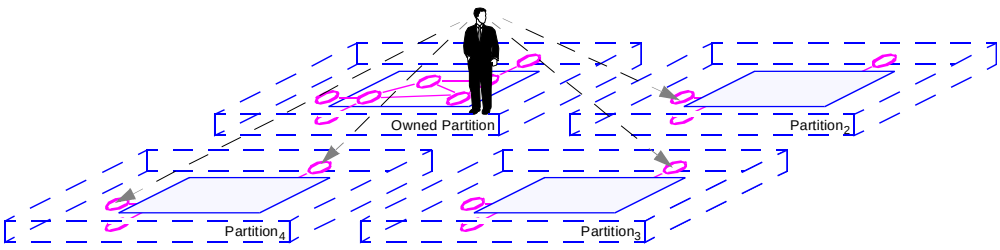
8.2 Component Schema Relationships and the Analyst's Horizon

Minimality and coherence of the overall problem analysis model are affected by the analyst's horizon. In this context we study the influence of the analyst's horizon on the expected *component schema relationships*.

8.2.1 Component Schema Relationships among Opaque Partitions

Opaque partitions reduce the partition owner's horizon to the partition environment of all other partitions outside his own responsibility. Figure 92 illustrates this situation:

Figure 92 partition owner Horizon for Opaque Partitions



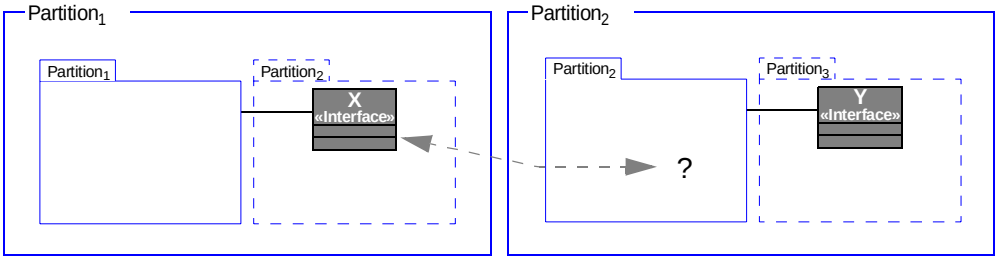
So analysing the environment of other involved partitions has two different objectives. First of all ensuring that all *inter-partition requirements* of the owned partition will be settled by related partitions. We call this objective *inter-partition settlement*. The second objective deals with assessing the impact of the environment of related

partitions with respect to the contents of an owned partition. We call this objective *inter-partition impact analysis*. In this section we simply use the abbreviated terms *settlement* and *impact analysis.*

The *partition environment* generally incorporates two kinds of class specifications. The first category represents the interface to external systems and is marked with the stereotype «interface». Requirements posed to other partitions, so-called *inter-partition requirements*, form the second category of class specifications within the *partition environment.* Thus class specifications within the owned partition depend on services provided by class specifications in other partitions. *Opaque partitions* only present a facade-class, expressing its offered services and its *partition environment* to external analysts, i.e. the domain owners cannot analyse the partition contents for potential correspondences. This situation leads to the definition of a new kind of *component schema relationship*, that we call *component schema dependency.* Figure 93 clarifies this situation[1].

Definition71 (Component Schema Dependency) – A *component schema dependency* refers to a *component schema relationship* that indicates an *unsettled inter-partition requirement* between the involved partitions to the affected partition owners. ◊

Figure 93 Component Schema Dependency in Opaque Partitions



The two depicted partitions, built in a package-based style, hide their internals and only present their environment to the involved analyst teams. $\texttt{Partition}_1$ contains a class specification named X demanded from $\texttt{Partition}_2$. The class specification Z, on the other hand, represents an *inter-partition requirement* in $\texttt{Partition}_2$ demanded from $\texttt{Partition}_3$. Due to the hidden internals of $\texttt{Partition}_2$ the correspondence of *artefacts* cannot be assessed. Since the coherence of a specification is directly influenced by the settlement of *inter-partition requirements*, we need a construct in order to raise the awareness of the affected analyst team. *Component schema dependencies* relate individual *artefacts* to entire *component schemas* and therefore realise reminders for the related analyst teams with respect to coordination of the

1 Note that transparent packages – indicated by dashed lines – do not follow the UML Notation Guide [OMG97b]. UML always reserves a view in its own right, in order to present package contents to CASE tool users.

stated *inter-partition requirements*. *Component schema dependencies* are therefore closely related to the partition owner's first objective, the *settlement* of *inter-partition requirements*.

Analysing the impact of the environment of associated opaque partitions with respect to the contents of the owned partition forms the partition owner's second objective. In this context, *partition owners* try to fulfil requirements posed with respect to their *owned partition*. Due to analyst's autonomy, partition owners proactively take into account *inter-partition requirements* stated by other analyst teams and try to fulfil them on the basis of class specifications forming the *partition contents* of their owned partition. In this context, the mapping of *initial partitions* to *component schema* plays an important role, leading to different activities for *package*- and *model-based partitions*.

Dependency in Package-Based Partitions

Component schema dependencies can easily be realised in *package-based partitions*, since the origin of class specifications representing *inter-partition requirements* has been determined positively. Now an appropriate *construct* has to be found, in order to link class specifications to the packages in different partitions. According to the metamodel document [OMG97a] the UML `Trace` relationship provides a conceptual connection between arbitrary model elements. So class specifications representing the *inter-partition requirement* can be linked to the packages representing the affected partition via a `Trace` relationship marked with the new stereotype «`dependency`». Although at first glance the stereotype «`dependency`» be confused with the UML `Dependency` relationship, there is still a significant difference between the two constructs. The UML `Dependency` can only link class specifications or packages to each other. That means the related *artefacts* have to share the same *construct*. A *component schema dependency*, however, may link class specifications to entire packages.

Dependency in Model-Based Partitions

Model-based partitions do not relate class specifications representing *inter-partition requirements* to related partitions. Consequently, this determination has to be realised before the class specification can be linked to the target *component schema*. Since UML metaclass `Model` is a specialisation of metaclass `Package`, *component schema dependencies* may be realised in the same way as in the *package-based partitions*. However, there is a significant difference as we will see. While *package-based partitions* directly relate the *inter-partition requirement* to a defined partition, *model-based partitions* do not provide this distinction. The *partition environment* formed by class specifications marked with the stereotype «`interface`» does not distinguish between external elements, outside the authority of the future systems, and *inter-partition requirements* which are located within the authority of the future system. Linking *inter-partition requirements* presupposes a manual review of the involved partitions and depends on the analyst's domain expertise. In other words, the analyst's judgement relates given class specifications to a distinct partition. This observation leads to another distinction of class specifications representing *inter-partition requirements*. So-called *identified component schema dependencies* describe the partition responsible for its provision, while in *unidentified component schemas depend-*

encies the target partition has to be detected by a review, either by the master team or by interested autonomous analyst teams. Consequently, model-based partitions only comprise *unidentified component schemas dependencies*, while *package-based partitions* only comprise identified ones, due to the replication of all involved partitions in the form of packages.

Component schema dependencies in *opaque partitions* represent reminders for the affected analyst teams that given structural and behavioural requirements have to be fulfilled by this partition. Due to this assumed dependency, the analyst team has to check whether an existing class specification within their partition contents fulfils the requirements. *Partition owners* therefore have to analyse the impact of *inter-partition requirements* on their *owned partition*. In order to fulfil the requirement there are two different strategies. In the first one, the class specification fulfilling the request has to be made visible to external analyst's. At the same time the original *component schema relationship* has to be linked to the class specification fulfilling the *inter-partition requirement* and then has to be changed into a *correspondence assertion*. The second strategy leads to new properties added to the facade-class representing the entire partition. In this case, the new properties are linked to the class specification in the target partition using a *correspondence assertion*.

Character of Component Schema Correspondence in Opaque Partitions

The character of *component schema relationships* between opaque partitions is basically influenced by the representation of *inter-partition requirements*, i.e. whether a *class-based* or a *facade-based* approach has been taken. While *facade-based inter-partition requirements* lead to *inter-construct level correspondences*, *class-based inter-partition requirements* lead to *intra-construct level correspondences*. Generally speaking, the specific representation of *inter-partition requirements* essentially influences the kind of expected *correspondence assumptions*. Thus the analyst's horizon can be regarded as less dominant in this context.

Impact of Opaque Partitions on Coherence and Minimality

Due to the hidden partition internals, the coherence of the overall problem analysis model formed by the set *component schemas* entirely depends on the cooperation of the involved analyst teams. Since *component schema dependencies* can be regarded as reminders for coordination among the involved analyst teams, there is no guarantee that the reminder will be changed into a *correspondence assertions*, leaving the overall problem analysis in an incoherent state. *Inter-partition requirements* simply have not been fulfilled. At the same time, the minimality of the specification cannot be guaranteed, since the hidden partition internals prohibit a detailed analysis of redundant *artefacts*, possibly induced by equipollence. Consequently, coherence can be assured on the basis of a cooperative coordination between the involved analyst teams.

Component Schema Dependency Revisited

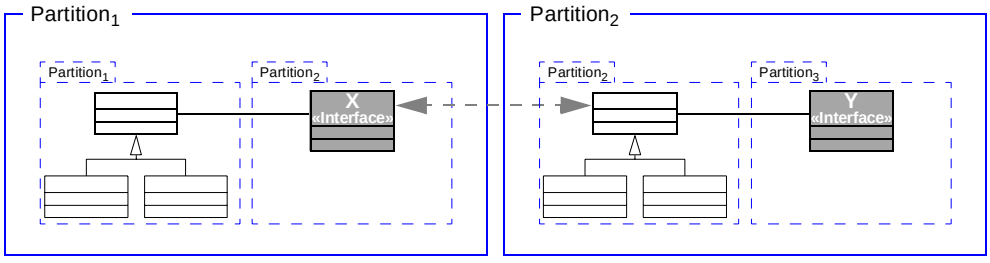
Unlike *component schema correspondence*, *component schema dependency* can be characterised as an organisational relationship driving the coordination of the involved analyst teams. In fact they can be regarded as *to be done* requirements, since a clarification of the concrete nature of the specified structural and behavioural requirements is vital with respect to the coherence of the overall specification. In

other words, *component schema dependencies* are a structural representation of a necessary coordination among the involved analyst teams on the level of *component schemas*. The coordination itself however remains outside the scope of *component schemas* and is entirely embedded in human communication. Consequently, the integration of *component schemas* requires that all *component schema dependencies* have to be resolved first, i.e. have to be changed into *correspondence assertions*. So the number of *component schema dependencies* always indicates pending coordination between the involved analyst teams, regardless of the concrete project organisation model and can therefore be regarded as a first handle for the quality of the overall specification with respect to minimality and consistency.

8.2.2 Component Schema Relationships among Transparent Partitions

Transparent partitions present all *artefacts* of a *component schema* to external analysts. Following the results of Section 7.3, pp. 131 all kinds of *component schema* relationships might exist. This observation leads to an order of applying the analysis for the various kinds of *component schema relationships*, from *component schema correspondences* to *component schema dependency*. In order to guarantee the coherence of the overall problem analysis model, *inter-partition requirements* have to be analysed first. This examination is related to all class specifications carrying the stereotype «`interface`». In the case of *identified component schema dependencies*, these class specifications have to be compared to the internals of the related *component schemas*. Then the minimality of the specification can be improved by an analysis of the *artefacts* representing the key abstractions of the involved partitions. Depending on the specific way the partition environment has been modelled, i.e. *facade-based* versus *class based inter-partition*, *intra-* as well as *inter-construct level correspondences* may be found.

Figure 94 Component Schema Dependency in Transparent Partitions with Class-Based Inter-Partition Requirements

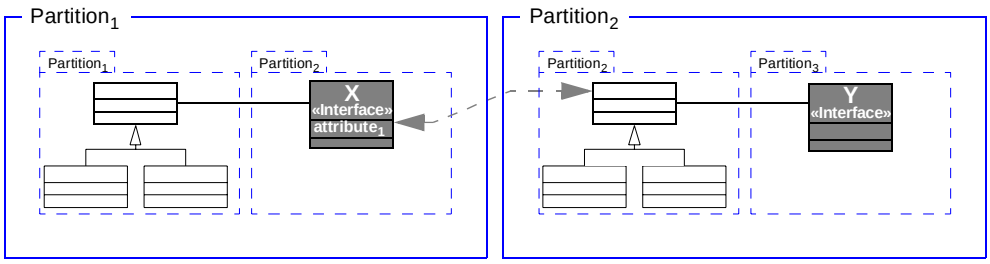


Class-Based Inter-Partition Requirements

As depicted in Figure 94, *artefacts* in *component schemas* with *class-based inter-partition requirements* have to be compared with *artefacts* forming the key abstractions of the related partitions. In this context, basically *inter-construct-kind relationships*, i.e. mostly class-class correspondences, have to be detected. If automatic as well as manual reviews do not detect any related *artefacts*, at least a *component*

schema dependency has to connect the artefact with the related *component schema*, as a reminder for coordination of the involved analyst teams. This analysis ensures the consistency of the overall specification and guarantees that once coordination has taken place between the involved analyst teams, *inter-partition requirements* will be fulfilled. Although class-class correspondences in this context are the most probable correspondences, differing levels of detail in the related *component schemas* might also lead to the *inter-construct-kind relationships* and consequently to *inter-construct-level relationships*. In other words, besides class-class relationships we may expect package-class relationships.

Figure 95 Component Schema Dependency in Transparent Partitions with Facade-Based Inter-Partition Requirements



Facade-Based Inter-Partition Requirements

As depicted in Figure 95, *artefacts* in *component schemas* with *facade-based inter-partition requirements* have to be compared with *artefacts* forming the contents of related partitions. *Facade-based inter-partition requirements* combine a set of structural as well as behavioural requirements in a single class specification representing an abstraction of an entire *component schema*. Attribute names in this facade-class denote real-world things on a fairly abstract level. The same observation holds for operations, but with a significant difference. According to naming conventions, verbs are used for operation names (see Section 6.1.1.2). The following quotation clarifies this situation:

> "Basic Functions supplies MONA with the necessary information relating to *Letters of agreement*, *Standard operating procedures*, *Airspace data* and *IAS limit data*." [Eur96a, p. 2-2]

The term »basic functions« is a synonym of flight data processing and denotes a facade class called `FA_FDPD_AAF` representing the *inter-partition requirements* between the partitions `FDPD` and `MONA`, each realised as a facade class. Although the terms »letter of agreement«, »Standard operating procedures«, »Airspace data« and »IAS limit data« refer to structural requirements, they have been modelled as operations, since the ORD did not detail their structure. All three terms have been prefixed with the term »provide« and represented as operations in a facade class. This realisation results in *inter-construct-kind* and *-level correspondence*s between *artefacts* stemming from different *component schemas*. In this context, *attribute-* and *operations-class correspondence*s form the most important *component schema relationship*. In differing levels of abstraction between the analysed *component schemas*, *package-operation* as well as *package-attribute correspondence*s may be found.

Package- versus Model-Based Partitions

Correspondences among partitions mainly depend on the way *inter-partition requirements* have been built. In other words, *package-* as well as *model-based partitions* lead to the same *correspondence assumptions*, but demand for different manual preparations. Since *package-based partitions* already relate the *inter-partition requirement* to a distinct partitions, they comprise so-called *identified component schema dependencies*. This connection still has to be established in *model-based partitions*. This can be accomplished on a manual basis or on basis of an analysis of correspondences for all involved *component schemas*, leading to increased efforts. In other words, *model-based partitions* first have to be transformed into *package-based* ones in order to relate *inter-partition requirements* to distinct partitions. This activity directly reduces the correspondence analysis effort, since at least the analysis with respect to the consistency of the overall specification can be simplified.

Improving Minimality

While the discussion at the beginning of this subsection basically dealt with guaranteeing coherence with respect to the overall problem analysis model, an improvement of the minimality of the specification leads to an analysis of partition contents. The central objective for this task is the detection of synonymous and equipollent *artefacts* in the *component schemas* involved. The main reason for this analysis is the fact that redundant (i.e. synonymous) *artefacts* may lead to subtle differences with respect to structure and behaviour of the resulting class specifications, which may hinder proper interfacing of the designed system. Resolving redundancies also reduces design and implementation efforts, since a single class specification can be used throughout the entire system. This activity also leads to an improved understanding of the overall problem statement, since less class specifications are generally easier to understand for software designers and the domain experts.

Improving the minimality of a specification therefore leads to an extensive analysis of all involved *component schemas*. Consequently *intra-* as well as *inter-construct-kind correspondence*s have to be detected at *intra-* as well as *inter-construct-level*. In order to keep the analysis efforts as low as possible, this analysis should only be applied at the end of requirements elicitation and negotiation. So this analysis combined with a coordination of the detected correspondences may prepare the following integration of *component schemas* towards a single integrated overall problem analysis model and therefore reduces the risk of misunderstanding structural as well as behavioural requirements stated in different *component schemas*.

8.2.3 Component Schema Relationships among Opaque and Transparent Partitions

Component schema relationships between *transparent* and *opaque* partitions reflect the situation encountered in our requirements engineering projects in air traffic management. Partition owners in this setting analyse *component schema relationships* of their transparent partition with the partition environment of a set of opaque partitions. This kind of analysis can be distinguished into two separate phases, where the first

one ensures the consistency of the environment of the owned partition and the second one discusses the impact of inter-partition requirements stated by other involved analyst teams on the partition at hand.

Consistency of the Partition Environment

Depending on the specific representation of the partition environment in the involved partitions, *intra-construct-kind correspondence*s can be expected for *class-based inter-partition requirement*s, while facade-based ones mostly lead to *inter-construct-kind* and *inter-level correspondendences*. Whenever *inter-partition requirements* cannot be fulfilled by the affected partition, i.e. correspondences can be found neither automatically nor manually, a *component schema dependency* has to be posted. *Component schema dependencies* draw the attentiton of the affected analyst team to this problem. The following coordination between the analyst teams should resolve the detected inconsistency among the partitions. The organisational implications of this activity are detailed in Section 8.3. The central objective in this activity is still ensuring that the partition environment of the owned partition will be satisfied by other involved partitions. In other words, structural as well as behavioural *inter-partition requirements* stated in given *component schemas* have to be fulfilled by other *component schema*, in order to avoid inconsistencies and to improve the coherence of the overall specification defined in a set of concurrently elaborated *component schemas.*

Impact of Inter-Partition Requirements Stated in other Partitions

While the previous paragraph dealt with ensuring the consistency of a partition with respect to other involved partitions, here we want to analyse the inverse direction, i.e. assessing the impact of inter-partition requirements stated in other opaque partitions on the owned partition elaborated by the assigned analyst team. Once again the expected correspondences depend on the specific way the *inter-partition requirements* have been represented in the other *component schemas*. *Class-based partitions* mostly lead to *intra-construct-kind correspondences*, while *facade-based* ones automatically lead to *inter-construct-kind* and *inter-construct-level correspondences*. Assessing the correspondence of the contents of an owned partition with the environment of other involved partitions indicates which class specifications are required in other partitions. Changes to the affected class specifications automatically threaten the consistency of the overall problem statement. Deleting these class specifications or dividing this class specification into a set of collaborating class specifications, i.e. application of a design pattern, will automatically leave unresolved inter-partition requirements in the other affected partitions. Comparing an owned partition with the environment of other involved partition, also directly influences the autonomy of the assigned analyst team. Class specifications with existing correspondences cannot be changed without prior agreement of the affected analyst team. All other can be changed at any time. So generally speaking, correspondences as well as dependencies among *component schemas* provide indications for the actual degree of mutual autonomy of given *component schemas*. The organisational aspects mentioned with respect to correspondence management are further detailed in Section 8.3.

8.2.4 Summarizing Component Schema Relationships and Analyst's Horizon

The discussion within this subsection clarified the relationship of the analyst's horizon on the different kind of *component schema relationships* in concurrent requirements analysis. Besides the analyst's horizon restricting the visibility of partition contents to external analysts teams, correspondences are heavily influenced by the specific way inter-partition requirements have been represented in the involved partitions. Facade-based inter-partition requirements automatically lead to inter-construct-kind as well as inter-construct-level correspondences, since attributes as well as operations are realised in terms of UML class specifications in the related partitions. The correspondence between operations and class specifications can only be assessed within defined limits. Whenever the naming conventions described in Section 6.1.1.2 have been followed closely, a domain ontology facilitates automatic detection of correspondences. But we have to stress that this observation is influenced by the requirements basis for the individual partitions and the *component schemas* describing them. Operational requirements documents with their high-level description of their partition environment are more likely to produce this kind of correspondences between facade-classes from external partitions and class specifications in the owned partition. Use cases, on the other hand, provide more detailed behavioural requirements which lead to operation names that match to the normative language reconstruction as described by Schienmann [Sch97] and therefore require more elaborate tools such as ontologies or a thesauri.

Dependency among Hidden Partitions

Opaque partitions, where only the partition environment is visible to external project participants, lead to a new *component schema relationship*, the *component schema dependency. Component schema dependencies* relate class specifications to entire *component schemas*. Consequently, dependencies directly represent potential inconsistencies between the affected *component schemas* and are drivers for the coordination between the involved analyst teams. This specific character of dependencies stresses their mainly organisational character, contrasting the structural linguistic character of correspondences. As a side effect, dependencies may also be used to post *inter-partition requirements* to *hidden partitions*. Although *hidden partitions* can be regarded as critical with respect to minimality and consistency, *component schema dependencies* allow at least a participation of these partitions in the coordination process between the involved analyst teams. *Inter-partition requirements* can be posted and therefore provide a reminder to the affected analyst team. We have to admit, at this point, that the integration of *component schemas* at least requires *opaque partitions* in order to ensure the consistency of the overall specification. Minimality can only be achieved with *transparent partitions* where redundancies among *artefacts* can really be assessed on the basis of *correspondence assumptions* detected on an automatic as well as a manual level. Analyst horizon and the specific representation of the partition environment are already influenced by the project organisation and more precisely by the coordination process between the involved analyst teams. So at this point we have to focus on primary organisational and, in this context, procedural aspects of concurrent requirements engineering. This observation becomes especially evident with respect to the resolution of detected correspondences and

dependencies. Renaming a class specification after the detection of synonym class-class correspondence leads to a direct class-class correspondence in the next assessment cycle. Excluding resolved correspondences from further assessments directly decreases the coordination effort and has the potential to improve a smooth elaboration of the involved partitions and the *component schemas* representing them.

8.3 The Management Process

Component schema relationships indicate potential redundancies and incoherence between the problem partitions elaborated by the involved analyst teams. Realising *component schema relationships* in terms of stereotyped UML `Trace` relationships enables coordination between the involved analyst teams without direct and uncontrolled changes to *component schema* contents. This approach overcomes the inherent risk of misunderstanding and overriding requirements stated in related *component schemas* and also increases analyst autonomy. *Component schema relationships* can also be used to facilitate supervising ongoing problem analysis or preparing the integration of individual *component schemas* towards an integrated problem analysis model, documenting the entire problem statement. Having discussed the nature of *component schema relationships* and necessary techniques for their identification, we now have to focus on organisational implications. Within this aspect we have to develop a management cycle for *component schema relationships* and analyse the general impact of different project organisations on their management.

8.3.1 The Management Cycle for Component Schema Relationships

Component schema relationships indicate assumed correspondences between a set of *component schemas*, representing concurrently elaborated partitions. Direct manipulations of individual *component schemas* can be avoided. Similar to the ViewPoint framework [NKF94], we decouple the detection of correspondences from their resolution. We still have to keep in mind that due to a homonymy of terms, disparate real-world concepts may have been modelled. Consequently, the correspondence assumption has to be rejected in the following review by the involved analyst teams. The other important demand with respect to this work is to provide an infrastructure for concurrent requirements engineering. *Component schema relationships* require a detailed review among the affected analyst teams in order to assert *component schema relationships*. On the basis of Batini's schema integration process, this subsection defines a management process for *component schema relationships* and proposes a realisation in terms of the UML metamodel.

8.3.1.1 Component Schema Integration Strategies

Before actually proposing a management cycle for *component schema relationships*, we want to take a closer look at the schema integration process, as defined by Batini [BLN86]. This stresses once again the specific differences between schema integration for federated database systems on one hand and concurrent requirements engineering on the other.

Schema Integration Process

Batini [BLN86] divides the schema integration process into the following phases:

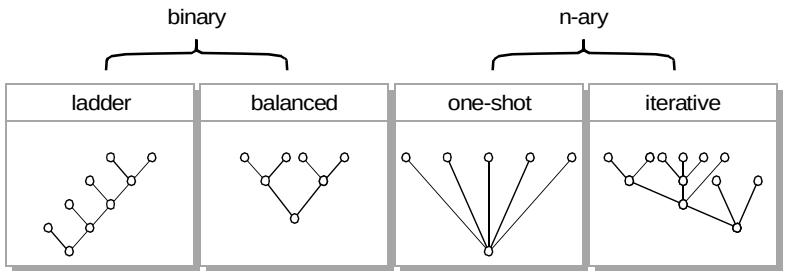
1 *Preintegration*
The sequence and grouping of schemas for the actual integration has to be considered. Especially the *integration processing strategy* has to be chosen in this phase (Figure 96)

2 *Comparison of schemas*
Detecting all kinds of conflicts is the central activity within this phase.

3 *Conformance of schemas*
In this phase the schema transformation actually takes place, resolving the conflicts detected in the previous phase. Renaming is the most primitive operation for the conformance of several *component schemas*. Type transformations, algebraic operations and redundancy elimination are examples of advanced techniques for conformance of conflicts over a set of *component schemas*

4 *Merging and restructuring of schemas*
Having resolved potential conflicts between *component schemas*, the integration is accomplished by merging the *component schemas*.

Figure 96 Types of Integration Processing Strategies [BLN86, p. 343]



Component Schema Comparison

Comparison of *component schemas* is detailed elsewhere in this work. Section 7.3.2 introduces *component schema correspondence* and *conflict* as the central prerequisites of this work. Section 8.2 analyses the impact of the analyst's horizon and specific representations of partitions and *inter-partition requirements* on the expected *component schema relationships* and also introduces the *component schema depend-*

ency as a new relationship in concurrent requirements analysis. Unlike correspondence, *component schema dependency* has an organisational character, in order to promote coordination between the involved analyst teams.

Schema Conformance

The intensional character of *component schemas* complicates the conformance of the involved *component schemas* considerably. Unlike schema integration, where instances are the primary concern, concurrent requirements analysis focuses on the conceptualisation of the application domain at hand. So the analysts are looking for a suitable representation of structural and behavioural requirements for relevant real-world things in the application domain. Since concrete instances do not have to be taken into account, analysts have considerably more freedom to conform *artefacts* specified in different *component schemas*. Consequently the range of solutions is far bigger. The only concrete criteria in this context are coherence – see Definition 16, p. 57 – and minimality of the overall specification. A specification can be called minimal, when a real-world thing does not lead to several *artefacts* in the participating *component schemas* expressing the same *concept*.

Merging and Restructuring of Schemas

Merging *component schemas* towards an integrated problem analysis model and restructuring the results paves the way for the creation of software requirements specifications covering the entire problem statement. While restructuring in schema integration also aims for optimised access to federated databases, i.e. improving the system's performance, concurrent requirements engineering demands minimal and coherent representation of the entire problem statement. This representation has to be solution-neutral and has to meet the attributes of good software requirements specifications in the sense of the IEEE-830 standard. Generally speaking, merging and restructuring of *component schemas* works under a considerably different set of concerns compared to schema integration.

Preintegration

The specific way partitions have been mapped to *component schemas* and the particular representation of inter-partition requirements does not impose an order of comparing and integrating *component schemas*. Although the processing strategy is relevant for both comparison and integration of *component schemas*, the autonomy of analyst teams demands for a clear distinction between the two phases within the overall process.

Component Schema Comparison Processing Strategy

The processing strategy for a comparison of *component schemas* focuses on the question of how analysts compare their assigned *initial partitions* to other involved *component schemas*. Due to the different project organisation models, we have to reconsider this idea. The central difference to schema integration can be found in the fact that an integration of the *component schema* is not foreseen in this phase. At first glance, a binary processing strategy may have been applied, since domain owners only compare their owned partition with potentially related *component schemas* on an individual basis. Due to the fact that this comparison has to be performed for *all* other involved *component schemas*, we will end up with a one-shot strategy, since the contents of the owned partition have not been changed. Instead comparison indicate assumed correspondences, conflicts and dependencies.

The autonomy of analyst teams working under different project organisation models, modifies this observation. In a master-multiple organisation model a centralised comparison of *component schema* will be performed, which leads to a *true one shot strategy.* But in autonomous analyst teams, each team may check the partition assigned to all other teams leading to a *distributed one shot strategy* with respect to the owned partitions. This observation also holds for the situation where partition owners check their assigned partitions against the other participating *component schemas* in a master-multiple organisation.

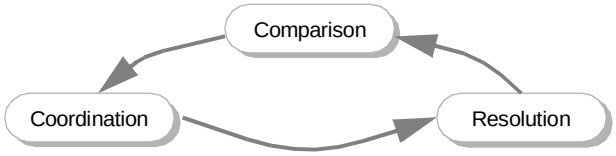
Processing Strategy for Component Schema Integration

Binary integration strategies have to be considered as least complicated, since only two component schemas have to be integrated at a time in a *ladder* or a *balanced strategy* (see Figure 96). One shot strategies, on the other hand, propose the integration of all involved *component schemas* towards an integrated problem analysis model in a single step. While binary strategies imply the resolution of comparison on a one by one basis, *one shot strategies* can only be applied when all comparison have been resolved for all participating *component schemas*. Consequently, a master-multiple organisation where comparison are maintained on a centralised basis suits one shot strategies, while autonomous organisations suit binary strategies, since comparison have been clarified only among given partitions.

8.3.1.2 Activities in the Management Cycle

Individual activities within the management cycle for comparison have not been discussed yet. Regardless of the actual project organisation, the management of comparison can roughly be divided into the three following activities: *comparison*, *coordination* and *resolution.*

Figure 97 The Management Cycle of Component Schema Relationships



Comparison

The first activity within the management cycle is called *component schema comparison.* Based on the assessment techniques described at the beginning of this chapter, various correspondences and conflicts can be detected using both automated as well as manual means. The related contents of *component schemas* will be linked with a UML `Trace` relationship. The actual kind of the detected *component schemas* relationship, is then indicated using the UML `Stereotype` proposed in Appendix D.1. Finally, component schema relationships linking contents of *component schema* from different *component schemas* represent potential incoherence with respect to the overall problem analysis model.

Coordination

Component schema relationships therefore have to be resolved as soon as possible, in order to assure a coherent overall problem analysis model. But the complex nature of problem analysis in industry-scale applications requires extensive *coordination* between the affected analyst teams, before changes to the affected *component schemas* actually resolve these conflicts. The autonomy of analyst teams working on distributed repositories further stresses the necessity to find a solution which suits the individual perspectives of involved analyst teams, without the inherent risk of misunderstanding and overriding structural as well as behavioural requirements. So *coordination* incorporates all activities in order to find a consensus among the involved analyst teams as to how the detected incoherence can be solved and a minimal overall analysis model can be assured. Means for finding this kind of consensus mostly belong to the realm of human communication and therefore remain outside the scope of this work.

Polysemy of terms as well as missing insights to related partitions are reasons for erroneously indicating conflicts or correspondences between *artefacts* stemming from different *component schemas*. Thus *component schema relationships* can only be regarded as assumptions requiring extensive review by the involved analyst teams. Acknowledging an assumed *component schema relationship* leads to the notion of *correspondence assertions*, as described by Larson [LNE89] and Spaccapietra [SPD92] in the schema integration literature.

Resolution

Resolution finally encompasses all activities to apply a coordinated solution to the affected *component schemas*. Renaming class specifications is only one of the possible solutions for resolving *component schema relationships*. Besides the individual actions resolving the detected *component schema relationships*, resolution gives rise to the question of how *component schema relationships* themselves have to be treated. Principally there are two different solutions to this question. The first one leads to erasing the *component schema relationship*, while the second one includes changing the *component schema relationship* into a standard UML relationship. The settlement of inter-partition requirements may for instance lead to replacing the *component schema relationship* with a UML `Dependency` relationship. Nevertheless, the actual solution entirely depends on the actual capabilities of the tool support and therefore is not detailed here.

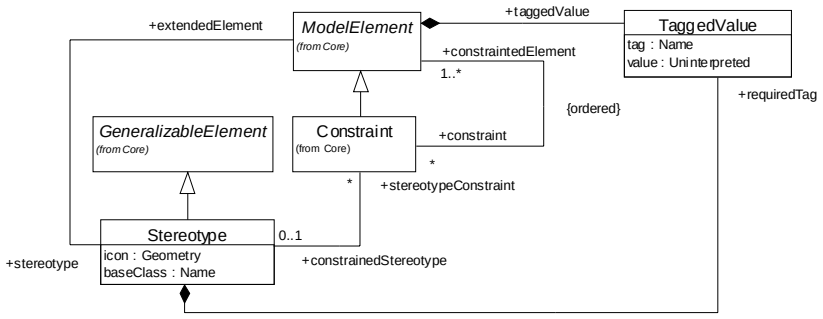
Metamodel Representation of the Management Cycle

The status of the management cycle for *component schemas* should be reflected in the metamodel. Assessing the remaining efforts for coordination as well as resolution of *component schema relationships* representing potential incoherence in the overall problem analysis model may be supported on this basis. Project management as well as individual analyst teams may profit from this realisation. A closer look at metamodel representation of stereotypes leads to the following observation. Stereotypes are realised via the composition of the metaclasses `ModelElement` and `TaggedValue`. This metaclass contains the property `tag`, carrying the name of the stereotype and the property `value`. The latest revision of the metamodel states:

> "A tagged value is a value-selector pair that may be attached to any element (including model elements and presentation elements) to carry various kinds of information, generally secondary to element semantics but possibly important to the modelling enterprise." [RJB99]

Figure 98 depicts the UML package `Extension Mechanism`, representing the metaclasses `Stereotype` and `TaggedValue`:

Figure 98 UML Package Extension Mechanism



Based on the various correspondences identified in Section we introduce the tag `CoordinationStatus` with the possible values `assumed`, `coordinated` and `resolved`. A complete list of all described stereotypes representing *component schema relationships* can be found in Appendix C. Attaching value pairs to UML `Trace` relationships allows to keep track of the coordination status between the involved analyst teams. Tagging the coordination state to the stereotypes provides guidance for actions to be undertaken by individual analyst teams and also indicates the project progress from a project management perspective.

Component Schema Relationship Notification

Based on the management cycle for *component schema relationships*, the notification of affected analyst teams still has to be discussed. Independent of the applied project organisation, which is detailed in the following subsection, we define two principle notification possibilities. For the time being we also abstract from the infrastructure realising these different notification styles. Due to the autonomy of analyst teams, we also have to take deferred reactions to the notification into account. The following two notifications styles can be distinguished: *push* and *pull notifications.*

- Push notifications:
  Comparison of *component schemas* is started by a given analyst team and all detected *component schema relationships* are propagated immediately to the affected analyst teams.
- Pull notifications:
  In this context detected *component schema relationships* are stored in a log file that can be accessed by all other analyst teams.

Pull notification is based entirely on the good will of the involved analyst teams and therefore bears the risk that detected *component schema relationships* will never be gathered by the affected analyst teams. In order to ensure at least the coherence of the entire problem specification, all analyst teams have to be *'eager'* to coordinate. On the other hand, pull notifications represent the most autonomous organisation, since only read access to the involved partitions is necessary. Push notifications at least guarantee that correspondences are propagated to the affected analyst team, which does not guarantee the following coordination and resolution of detected *component schema relationships*.

8.3.2 Project Organisation and the Management of Component Schema Relationships

Analyst teams in concurrent requirements engineering may be organized according to the following three forms: *centralised*, *autonomous* and *master-multiple*. Different project organisation models also give rise to the question at which points in time coherence can be achieved with respect to the overall problem statement. Especially with respect to autonomous analyst teams elaborating their partitions currently, coherence is a must, in order to meet the criteria of good requirements specifications as proposed by the IEEE-830 standard [IEEE830].

Centralised Analyst Teams

A centralised analyst team works under centralised project management on a single repository. Due to the centralised repository, the partitions are represented in package-based style and review and coordination are driven by the centralised project management. Coordination between the analyst groups is triggered and enforced by the project management. Due to the tight management the coherence of the specification can be guaranteed at given points in time. In other words, whenever all detected *component schema relationships* have been coordinated and resolved under the supervision of the project management, a specification may be called coherent, since all inter-partition requirements have been fulfilled.

8.3.2.1 Autonomous Analyst Teams

Autonomous analyst teams work *without* centralised project management and thus represent the most complicated form of project organisation. The absence of a centralised authority leaves coordination and resolution of correspondences entirely to the good will of individual analyst teams. Autonomous analyst teams base their work on an agreement concerning the responsibility for the involved problem partitions, prior to the actual start of requirements elicitation. The management cycle for *component schema relationships* in autonomous team organisation is also influenced by the specific way in which partitions have been mapped to *component schemas*, i.e. *package-* or *model-based partitions*.

Package-Based Partitions

The involved analyst teams elaborate their *component schemas* in distributed repositories. All *component schemas* reflect the initial and agreed problem decomposition in terms of UML packages. Whenever a given analyst team encounters inter-partition requirements, they are able to assign them directly to the related partition, since their *component schema* already incorporates packages representing the other involved partitions. This assignment automatically facilitates correspondence detection in the sense that the UML class specifications representing structural as well as behavioural requirements can be compared directly to the contents of *component schemas* deemed responsible for fulfilling these inter-partition requirements. That means ensuring the coherence of the entire problem statement leads to reduced correspondence detection efforts, since the source of the requirement and consequently the relevant class specifications to be compared are known beforehand.

Model-Based Partitions

In *component schemas* realising model-based partitions, the partition environment, marked with the stereotype «`interface`», is not assigned to given partitions. As a result there is no distinction between class specifications representing external systems and inter-partition requirements. So the entire partition environment has to be compared to *all* involved partitions, leading to dramatically higher detection efforts compared to package-based partitions.

The Correspondence Management Cycle

Recalling the beginning of this section, the management cycle for *component schema relationships* has been divided into the states *assumed*, *coordinated* and *resolved*, represented as tagged values to UML `Stereotypes`. The partition owners can check all other *component schemas* for *component schema relationships* at any time. Independent of a push or pull notification cycle, the coherence can only be achieved on a partial level, i.e. when all requirements formulated by other teams have been reflected in the *component schema* at hand. Local coherence can only be regarded for the owned partition with respect to all other *component schema* linked to it via a *component schema relationship*. In other words, a partition can be called *coherent*, when all *component schema relationships* carry the state ‘resolved’. The autonomy of analyst teams, on the other hand, does not guarantee that formulated inter-partition requirements will be reflected by the other analyst teams. Consequently, coherence as well as consistency of the overall specification cannot be guaranteed at any time. Moreover, coherence can only be reduced to the *component schema* of the assigned analyst team. Ensuring coherence of the overall problem analysis model implies a high degree of cooperation among the analyst teams and direct responses to all notified correspondences.

8.3.2.2 Master-Multiple Analyst Teams

A master-multiple organisation model combines autonomous analysis of problem partitions with centralised project organisation. Project management defines the *initial partitions* and assigns them to the participating analyst teams. Although the master team may also be responsible for one of the *initial partitions*, its central responsibility is reviewing all *component schemas* representing the overall problem

statement. So the management cycle for *component schema relationships* is initiated by the master team and detected *correspondence assumptions* lead to notifications for the involved analyst teams. Independent of using push or pull notification, the remaining coordination efforts for a given cycle can be assessed centrally, without interventions to the individual *component schemas*. Another advantage of this organisation form can be found in relating *individual inter-partition requirements* found in model-based partitions to the responsible analyst team, which further reduces correspondence detection effort. The complete problem analysis model can be regarded as coherent, when all detected *component schema relationships* have been changed to the state 'resolved'. This leads at least to coherence at given moments in time, which cannot be guaranteed in autonomous analyst teams. The minimality of the overall problem specification still depends on the horizon of the master-team. In the case of transparent partitions, where the contents of a *component schema* are entirely visible to the master team, the integration of *component schemas* can safely be accomplished whenever all correspondences have been resolved and the elicitation and negotiation for all involved *component schemas* has been finished.

Master-multiple organisation, thus reconciles the conflicting objectives *autonomy* of analyst teams and *coherence* of the overall problem analysis model in a suitable manner. From a practitioners point of view, one of the central prerequisites of efficient and coherent problem analysis in this context is a cooperative atmosphere among the participating analyst teams. A single analyst team may threaten the entire project success by ignoring notified *component schema relationships*. Consequently, the coherence of the entire problem analysis model remains in question. This risk can only be ruled out at the price of sacrificing analyst autonomy through centralised project organisation. Autonomous analyst teams as well as master-multiple organisations still require substantial cooperativity between the participating analyst teams. Transparent partitions then can easily be integrated since all correspondences have already been resolved. Project progress can also be monitored, which is a strong point for general management and leads to improved the cost efficiency.

8.3.3 Subsection Summary

The management cycle for *component schema relationships* can be divided into the three general states *comparison*, *coordination* and *resolution*. These three activities are reflected in our proposed representation for *component schema relationship* in terms of the UML metamodel. Due to the potential deficiencies of *terms* labelling *artefacts*, all detected *component schema relationships* are marked with a `TaggedValue` *assumed*. The state of a *component schema relationships* has to be changed to *coordinate* when coordination between the affected analyst teams starts. Finally, after an agreement between the teams, the value is changed to *resolved* when a suitable representation has been found for the mutual requirements. Due to frequent *component schema* changes, especially in the earlier phases of requirements analysis projects, we do not detail strategies for excluding stated *component schema relationships* from future assessments. We simply assume that we start a new cycle where

uncoordinated and unresolved *component schema* from a prior detection, lead to the same result in the following cycle. The central problem in this context is the evolving character of *component schemas*. When no changes have been applied to given component schemas, we receive the same *component schema relationships* as in the previous cycle. But due to the autonomy of analyst teams, new correspondences, which were not detected in the previous cycle will probably be found. Depending on the *component schema notification style*, i.e. push versus pull, this leads to different reactions of the tool environment. Push style notification shall lead to a removal of all *component schema relationships* from the previous cycle, over all participating repositories. Pull notification style, which leads to overriding the local *component schema relationship* protocol, only leads to local changes and therefore tends to be easier to implement.

9 Tool Support

This chapter outlines tool support for concurrent requirements engineering in industry-scale projects. We identify major components in this context and outline their basic capabilities. Besides an integrated solution combining the identified components, we also consider tool support based on a combination of off-the-shelf tools.

9.1 Components for a Concurrent Requirements Engineering Workbench

In this section we identify major requirements for a concurrent requirements engineering workbench for *component schema relationships*. The central objective in this context is supporting all described project organisation models, in order to suit the particular situation for future projects.

Workbench Components

A concurrent requirements engineering workbench can roughly be divided into the components *CASE tool*, *linguistic support* and *communication infrastructure*. The CASE tool in this context realises the selected subset of the UML metamodel. Drawing capabilities and the provision of persistent storage for UML models are the most important features from a user perspective. Code generation is neglected due to the solution-neutral character of requirements specifications. Another important aspect lies in the definition of conformance criteria with respect to the metamodel. Although *Rational Rose 98* claimed to be UML compatible, there are still considerable omissions with respect to the published metamodel versions [OMG97a, RJB99]. Linguistic support primarily compensates the deficiencies of off-the-shelf tools, where even a trivial support for managing the domain vocabulary is not supported. In the sense of Schienmann, we want to add method-neutral management of *terms* to off-the-shelf tools. Especially within industry-scale problem statements, the concrete meaning of *terms* cannot be presupposed at project start. Therefore we want to add linguistic support to off-the-shelf tools. We also want to differentiate the individual capabilities of potential linguistic support. Finally requirements for a communication infrastructure supporting the analyst teams concurrently elaborating their partitions have to be identified.

9.1.1 Linguistic Support

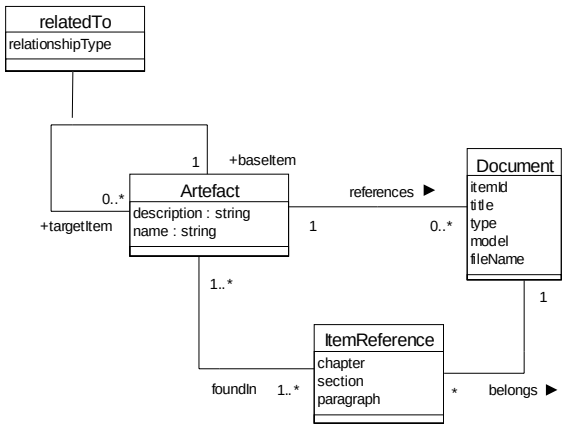
Dealing with the specific *terms* of a given application domain is one of the most important tasks in industry-scale problem analysis. Off-the shelf CASE tools usually neglect this aspect completely. The meaning of *terms* is not clear in the earlier phases of requirements elicitation and negotiation. Suitable object models cannot directly be derived from an insufficient understanding of intricacies of the *terms* in the applica-

tion domain at hand. Relevant *terms* have to be collected first, then their specific meaning has to be clarified through a concise definition. Finally *artefacts* may be constructed on the basis of clarified and well defined *terms*. Within the linguistic support differing levels of sophistication can be identified, from simple relational databases realising simple glossaries to domain ontologies providing a logical theory for terms within an entire application domain. In this subsection we want outline the differing capabilities of each technique.

9.1.1.1 Relational Database Systems

Simple linguistic support for concurrent requirements engineering can be provided in the form of data dictionaries realised on top of relational database management systems (RDBMS). In our air traffic management projects, we developed such a simple tool [HFW97], based on Microsoft Access for Windows [Mic97]. This data dictionary followed two main objectives. Tracking the sources of terms, with references to the ORDs and providing simple mechanism for defining arbitrary relationships among terms. Figure 99 depicts the UML static structure diagram of this data dictionary [HFW97].

Figure 99 UML Static Structure View of the CNS/ATM Data Dictionary

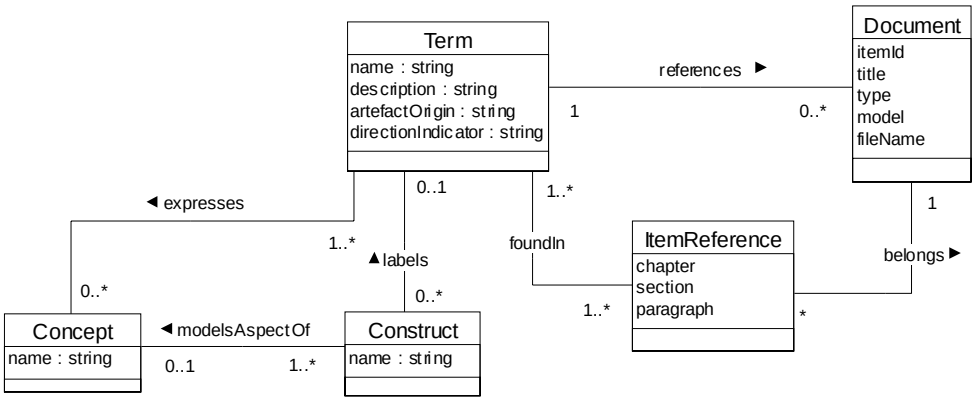


User-defined relationships between terms have been modelled via the UML association class `relatedTo`. Particular values in the property `relationshipType` realise a discriminator for user-defined relationships. The simple dictionary structure does not distinguish between linguistic relationships and relationship constructs offered by the used modelling technique. In other words, linguistic synonymy and UML composition only lead to distinct values of the property `relationshipType`.

The same observation holds for the terms themselves. The class specification `Artefact` does not distinguish between *terms* expressing *concepts* and the labelled *construct*. Consequently a clarification of the meaning of terms in the ontological sense is not possible in this data dictionary implementation. Default discriminators for user-defined relationships between terms are `generalisation`, `antonym`, `synonym`, `attribute`, and `aggregate`. The query capabilities of the relational data model, however, cannot deal with the intensional structure created by recursive term relationships. Existing specialisation hierarchies on a terminological as well as an artefact level call for computational completeness, as provided by programming languages. On the basis of Microsoft's Visual Basic for Applications, we implemented a simple browsing mechanism for ascending and descending the intensional structure defined by generalisation/specialisation relationships among terms.

In order to realise the *term*, *artefact*, *concept* triangle outlined in Section 8.1, the relational structure has to be extended. Besides these three elements we have to keep track of the applied UML construct for given artefacts, as well as their origin, i.e. their complete access path as realised in the UML name space concept. This leads to the following UML static structure diagram:

Figure 100 Extended Definition for Simple Management of Terms



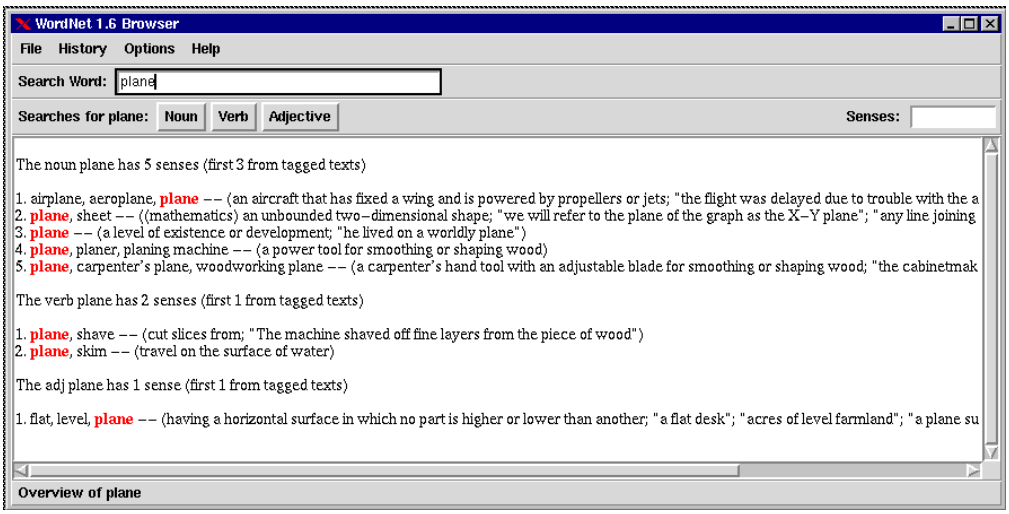
With the exception of the *concept*, all information can be derived directly from component schemas, without detailing this task here. In order to facilitate the comparison of terms, they have to be transformed to their stems, which can either be realised with a morphological function, as a C library function offered by WordNet [Mil95], or manually by the analysts. The meaning of individual terms – the *concept* – can either be added manually or retrieved from thesauri and domain ontologies. Simple relational queries can be used for an analysis of the used domain vocabulary. Queries such as '*Which class specifications are related to the concept »xyz«* can now be posed to the database. The central difference in our refined data dictionary is the clear separation of *terms*, *constructs* and *concepts*, previously treated in a uniform manner. Realising linguistic support on a relational database allows manual clarification of the meaning of terms, as represented by the specification `Concept`.

This overcomes the restricted capabilities of CASE tools on a terminological level, but puts the burden of clarifying the meaning of terms entirely on system users, for which it enables external sources, such as thesauri and domain ontologies.

9.1.1.2 Thesauri and Lexical Databases

Thesauri provide a more sophisticated linguistic support compared to simple data dictionaries, by offering a full set of linguistic relationships such as synonym, hypernym, hyponym, meronym and holonym. Our example in this context is the public domain tool WordNet developed at Princeston University's Cognitive Science Laboratory [WN98]. WordNet provides broad coverage of natural language terms and is therefore not specialised to a specific domain. WordNet treats nouns, verbs and adjectives separately, in order to facilitate retrieving *terms*. Separating the vocabulary into syntactical categories also allows the definition of distinct hierarchies of terms for nouns, verbs and adjectives, which becomes far more complicated in an integrated database treating all syntactical categories together. Figure 101 depicts the WordNet output for the term »plane«.

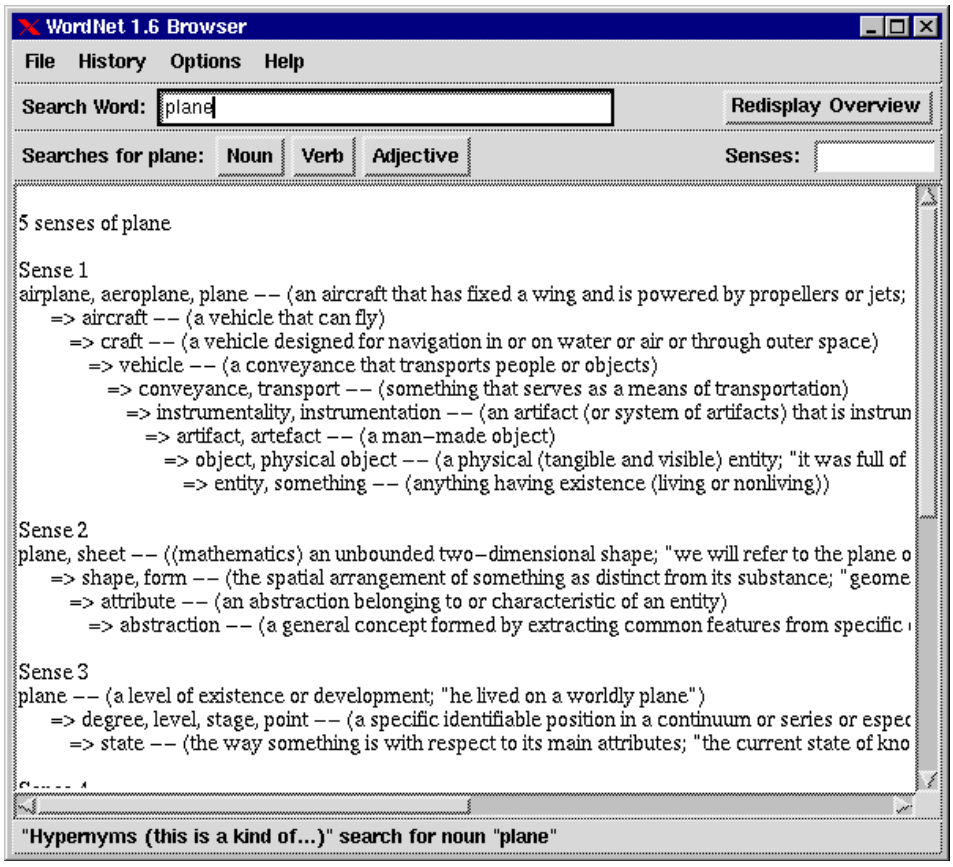
Figure 101 WordNet - A Lexical Database for English



The terms in our example are grouped according to their different senses and their different syntactical category. Each sense is documented by set of synonyms and a description clarifying the meaning of the word sense at hand. Definitions combined with the set of relevant synonyms distinguish the different meaning of *terms* allowing users to differentiate between a »plane« in the sense of an '*unbounded two-dimensional shape*' in mathematics and an »aircraft« in the air traffic management domain.

WordNet's most interesting feature, however, is the built-in *concept lattice* based on the linguistic hyper-/hyponomy relationship. Figure 102 depicts the hypernym hierarchy for the term »plane«.

Figure 102 WordNet - Hypernyms for the Term »Airspace«



Due to WordNet's broad coverage of natural language, it does not provide hyponyms and meronyms for the term »airspace«. So its implemented *concept lattice* may be used as a starting point for the creation of a terminological knowledge base for the application domain at hand. Swartout et al. [SPK+96], for instance, used WordNet in order to extend an existing top-level ontology for military air campaign management. The *Large-Scale Distributed Information Systems Department* at the University of Georgia built a bibliographic ontology on the basis of WordNet terms [KaS98]. Consequently, WordNet provides an ideal starting point for extending the existing lexical database to cover specialised domains, such as air traffic management.

The depicted examples of WordNet make use of the graphical user interface, the so-called WordNet Browser, providing integrated access to all implemented functions [Mil95]. Alternatively to the graphical user interface, WordNet provides direct access

to its database via a set of C library functions, documented in the introduction to WordNet [MBF+93]. In this context, the library functions `findtheinfo` and `morphword` are the most interesting ones for this work. The function `morphword` can be used to retrieve the word stems of individual terms, facilitating the syntactical comparison of terms. Plural forms may be changed into singular ones and therefore ease pattern matching. The function `findtheinfo` looks up a word in the database and retrieves the different incorporated senses. These two functions provide the central interface to WordNet for tools making use of its lexical database. So we can look up the base form, a set of synonyms and the definitions for given terms describing concepts in the application at hand. However, looking up terms still requires user interaction in order to select the appropriate meaning of given terms in the application domain at hand. The user always has to select the appropriate sense in order to choose the correct meaning of the term (see Figure 101, p. 190).

WordNet Contribution

Although WordNet does not provide a specialised domain vocabulary, its broad coverage of natural language terms and the elaborated hierarchy of terms contribute to the clarification of terms in a given domain. Its morphological function also helps to derive word stems from given terms. Word stems in particular facilitate a comparison of terms and ease the clarification of their meaning in arbitrary domains. This function even works when WordNet's built-in database does not include the respective term in question. Therefore WordNet provides a nucleus for the classification and clarification of domain vocabulary, as found in requirements specifications. The provision of so-called lexicographers files also allows a specialisation of the database towards the domain at hand. The term »airspace« can be refined through the provision of meronyms and hyponyms as modelled in a *component schema*, making use of the conceptualisation represented in the problem analysis model. Problem analysis models always incorporate a linguistic clarification of terms based on a method-specific reconstruction of an expert language. Customisation of the database is not further detailed in this work.

9.1.1.3 Ontologies

Ontologies exceed the capabilities of thesauri through the provision of a complete logical theory concerning the expert vocabulary of the domain at hand. A central difference between thesauri and ontologies is the tailoring of latter's vocabulary for a given domain. Unlike WordNet, where a term may be related to a set of different meanings stemming from different domains, an ontology only contains a fixed meaning of given terms for distinct domains. Based on various description logics, such as CLASSIC [BBM+89], their semantic relationships especially enable reasoning with the generalisations, the hypernym relationship in linguistics. The entirely intensional character of *component schemas* only makes use of the so-called *terminological component* neglecting the so-called *assertional component* [Klu98, MIB96], where terminological reasoning is combined with a reasoning over properties of instances. Hence reasoning is reduced to the terminological level. Description logics have become quite popular in the federated database community, due to the combination of termi-

nological and assertional reasoning [KaS98, LRO96]. Description logic's most interesting feature with respect to this work is the automatic computation of subsumption, i.e. computing whether a terminological description is a subtype of another type, mirroring the UML generalisation hierarchy. The SENSUS Ontology, developed at the Information Science Institute of the University of Southern California provides an example of this kind of ontology [Sen95]. It comprises a hierarchy of word meanings, meronyms and synonyms. Although this ontology has a general purpose character – created for machine translation – it provides a good starting point for domain specific extensions, such as air traffic management.

The primarily linguistic character of object-oriented analysis models also provides a good starting point for enhancing existing ontologies. An application ontology comprises all *terms* relevant in a given application. These *terms* form a subset of the expert vocabulary used in the application domain at hand. So incrementally extending the vocabulary stored in an ontology widens its scope towards a domain ontology. Ontologies provide automated clarification of word meanings and detection of synonyms in the domain at hand. Reasoning with hypernyms is the central advantage of ontologies compared to thesauri, due to their encapsulated logical theory. Thesauri contain basically the same information, but the access is different with respect to an ontology. While users may browse thesauri, ontologies may reason about the meaning of terms. The central advantage of an ontology can be found in the clarified meaning of terms. Replacing synonyms by their *concept*, i.e. the meaning of synonymous terms, directly facilitates an automated detection of correspondences. In this context, *specialisation correspondence* (see Figure 67, p. 140) and *attribute-class correspondences* (see Figure 70, p. 143) can be detected automatically. But we also have to admit, that the efficiency of this kind of automated detection directly depends on the number of terms stored in the ontology.

9.1.2 Communication Infrastructure

The second important building block for tools supporting concurrent requirements engineering can be found in the communication infrastructure, realising the notification, coordination and resolution of detected *component schema relationships*. Within the communication infrastructure three different levels of sophistication may be distinguished, from simple e-mails to an integrated tool environment using CSCW techniques. In other words, this section basically describes workflow support for concurrent requirements analysis.

9.1.2.1 E-Mail

E-mail can be regarded as the most simple and minimal support for autonomous analyst teams. In this context, detected *component schema relationships* can be propagated either directly to the affected analyst team or to all involved teams. *Component schemas* have to be exchanged between the analyst teams and each team detects

potential *component schema relationships* in isolation. Afterwards the results, i.e. detected *component schema relationships*, are described in textform in an e-mail and sent to the affected analyst team. In order to clarify the detected *component schema relationship*, parts of the affected *component schemas* can be attached to the e-mail.

E-mail as communication infrastructure has to be regarded as insufficient. *Component schema relationships* do not have a physical counterpart in the affected models. All involved analysis documents, i.e. *component schemas*, data dictionaries and textual information, have to be kept redundantly, giving rise to inconsistency problems. Resolving *component schema relationships* belongs entirely to the discipline of the affected analyst team. In short, e-mail has to be regarded as baseline support for concurrent requirements analysis, although the support is far too rudimentary with respect to complex industry-scale applications such as air traffic management.

9.1.2.2 The ISST-SPE

The Software Production Environment (SPE) for Small and Medium Enterprises, developed at Fraunhofer ISST [KASP96, KKA+97] defined a software development environment based on off-the-shelf tools such as Rational Rose 98, SNiFF+, Microsoft Project for Windows and Lotus Notes. Although specified for concurrent software development for a consortium of participating companies, the SPE provides a suitable communication infrastructure for concurrent requirements analysis. Dirk Kurzmann's diploma thesis describes extensions to the SPE, in order to suit concurrent requirements analysis [Kur98]. In this section we would like to summarize the central features of SPE [KASP96, KKA+97] and the proposed extensions towards concurrent requirements engineering.

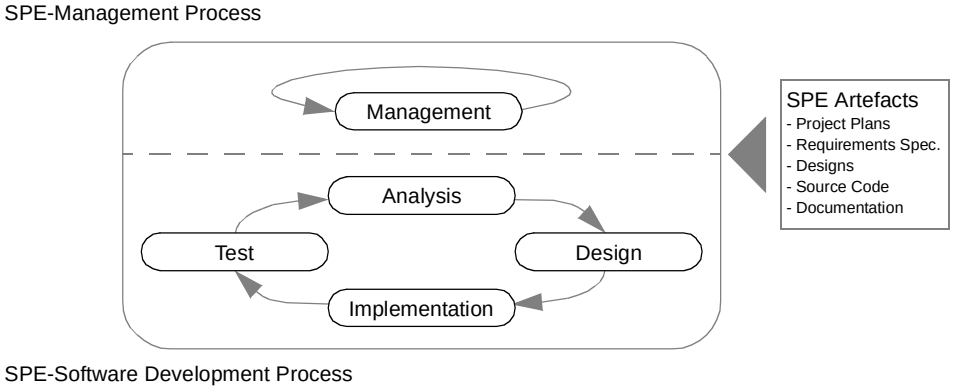
The SPE approach basically distinguishes between two development cycles in concurrent analysis, the *management process* and the so-called *software development process*. The software development process incorporates the well-know activities of the software development cycle, such as analysis, design, implementation and test. The management process encompasses the organisation and budgetary control of a concrete software development project. Figure 103, p. 195 depicts the general SPE process.

All involved documents within a project are subsumed under the term *SPE artefact*. This comprises project plans, requirements specifications, design documentation as well as source code. In this section we are especially interested in the management process, organizing the pursuit of the given projects and the underlying communication infrastructure. The project plan in SPE is generated using Microsoft Project for Windows[1]. Individual project activities are structured and assigned to required

1 Copyright by the Microsoft Corporation

resources, i.e. analysts, designers and reviewers. Design and development activities are then coordinated via a groupware solution based on Lotus Notes[1]. Lotus Notes realises two important functionalities in this context. First of all it provides a centralised repository maintaining all *SPE artefacts*. Second, *SPE artefacts* can automatically be replicated to Lotus-Notes instances of the project participants. The project plan is the basis for triggering individual project activities, for elaboration by assigned participants. The central basis for accomplishment of individual tasks is the so-called »saved«, »proposed«, »published« cycle enabling distributed elaboration of assigned tasks. We briefly want to outline how this cycle can be used in concurrent requirements analysis with master-multiple team organisation.

Figure 103 The SPE Process

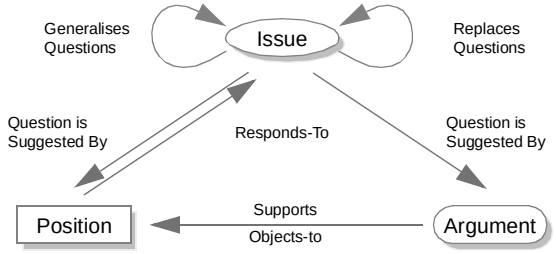


Using the ISST-SPE for Concurrent Requirements Analysis

Having defined the initial project plan, the initial partitions are then assigned to the analyst teams. The assigned analyst team then elaborates the partition in isolation. In this phase, the partition is set to the state »saved«. Whenever the resulting *component schema* reaches a sufficient degree of stability it is set to »published«. Replication in Lotus Notes automatically propagates the updated *SPE artefacts* to the distributed Lotus Notes databases of all other involved teams. *Component schema* analysis can afterwards be applied in order to detect correspondences, conflicts and dependencies. Finally, the state of the partition is set to »published« when all *component schema relationships* have been resolved and the next requirements elicitation cycle is started. Lotus Notes in this context only provides a distributed repository for SPE artefacts. Since Rational Rose, the off-the-shelf CASE tool in SPE, does not maintain distributed repositories we need additional means for representing *component schema relationships*. Dirk Kurzmann's diploma thesis [Kur98] proposes interesting extensions to the current ISST SPE prototype. The IBIS structure, proposed by Rittel and Kunz [RK70], provides the central metaphor for this realisation. Figure 104, p. 196 depicts the IBIS structure:

1 Copyright by Lotus Corporation

Figure 104 The IBIS Structure



Issues in this context represent *component schema relationships* to be discussed among the involved analyst teams. *Positions* represent potential solutions, for instance renaming a class specification in given *component schemas*. *Arguments* express reasons to support or object to positions suggested on a given issue. Issues themselves may regard potential problems in a generalised form or may even replace previously stated issues. The IBIS structure can be used for clarification of *component schema relationships* in the following way. Each detected correspondence leads to an *issue*. An *argument* presents the assumed reasons for the detected *component schema relationship*, while the *position* contains the affected *artefacts*. *Positions* may further propose suitable solutions in order to resolve the detected *component schema relationship*. The IBIS structure can therefore be used for efficient documentation of *component schema relationships*, incorporating assumptions with respect to their reasons and potential solutions. *Issue*, *position* and *argument* provide comprehensive and traceable documentation of the entire management cycle of *component schemas*. Dirk Kurzmann proposed a realisation of the IBIS structure on top of Lotus Notes, realising centralised persistent storage of *component schema relationships*. Due to its groupware functionality, Lotus Notes can automatically replicate new issues to all affected analyst teams, so that all activities comprised in the management cycle of *component schemas* can automatically be forwarded to all involved analyst teams.

SPE and Ibis Contributions

Integrating the management cycle of *component schemas* in the ISST SPE, with Lotus Notes as the central backbone, has a number of advantages. First of all, the SPE provides an integration of all relevant analysis documents through its persistent storage. The built-in replication functionality of Lotus Notes realises the notification of all involved analyst teams. An implementation of the IBIS structure realises *component schema relationships*, but without direct access to the involved model elements, since the IBIS structure itself is not integrated in the UML metamodel, as the basis for our *component schemas*. In other words, realising IBIS on top of Lotus-Notes provides a *loose* realisation of *component schema relationships*, since direct access to *artefacts* in the CASE tool's repositories is not provided. IBIS also abstracts from the concrete assessment of *component schema relationships*, i.e. IBIS may support automated as well as manual detection of *component schema relationships*. The central contribution of this realisation, besides its persistent storage capabilities, is improved automated support for the management of *component schema relationships*, which is the major reason for listing this solution in the context of communication infrastructure. Galler summarizes this functionality as a *cooperation functional-*

ity [GHS96]. Another positive aspect of integrating concurrent requirements analysis in the ISST SPE is the provision of an integrated environment supporting the entire software life-cycle. In this context, objects become the sole paradigm throughout the entire software life-cycle, from requirements engineering to maintenance and system evolution.

9.1.2.3 CSCW-Support for Integrated Tools: Information Subscription and Monitoring

The described extension of the ISST SPE in the previous subsection introduces a solution based on the IBIS structure and basically provides a *cooperation functionality* in terms of the groupware community [GHS96]. In this subsection we want to discuss a solution based on the UML metamodel and distributed repositories realising *coordination functionality.* According to Galler, coordination functionality in this context involves the following aspects [GHS96]:

- Access to meta-data (i.e. information about who changed data and why),
- Information supply:
  - shared project documents
  - information subscription
- Monitoring:
  - product monitoring user
  - activity monitoring.

Tool implementations based on the *OA&D CORBAfacility Interface Definition* as specified in the OMG standard [OMG97d] satisfy the first aspect. Within information supply we want to highlight information subscription. In the context of this work, information subscription applies to two distinct aspects: the notification of *component schema relationships* and change propagation for individual elements of *component schemas*. *Inter-partition requirements* provide an interesting example of the latter case of information subscription. *Component schema dependencies* announce inter-partition requirements to be settled by another analyst team. A new UML class specification representing the requirements has to be added in the *component schema* deemed responsible for the settlement of these requirements. Whenever project progress leads to additional structural and/or behavioural requirements, notifications can automatically be generated and forwarded to the affected analyst groups.

While in the previous subsection the IBIS structure provided the integration of related project documents via a meta-structure, information subscription also requires integration into the UML metamodel. IBIS issues only contain textual references to elements of *component schema* and the *component schema relationships* linking them in the involved partitions. Information subscription can only be realised on the basis of a distributed repository using the *OA&D CORBAfacility Interface* [OMG97d]. This demands two important prerequisites: precisely defined *component schema relationships,* and a notion of *event, condition, actions* rules in the sense of

active databases [ElN94]. With respect to Nuseibeh's ViewPoint framework [NKF94] *intra-viewpoint check actions* and *assembly actions* as part of the *workplan slot* provide a basis for implementing information subscription. The other important aspect described by Galler is a monitoring functionality, allowing on-line monitoring of group activities. This functionality may inform the project management concerning the individual status of *component schema relationships* and which management cycle of *component schemas* have successfully been accomplished. CSCW support for *component schema relationships* therefore has three important prerequisites, access to distributed repositories, a precise definition of *component schema relationships* in terms of the UML metamodel, and a communication infrastructure for the notification of affected analysts teams. Galler et al. demonstrate this kind of functionality in a prototypical tool called ContAct [GHS96], supporting business process reengineering activities. In their prototype the ARIS toolset [Sche94] has been linked to IBM's FlowMark workflow management system [IBM95].

Component Schema Relationships and CSCW Support

Integrating CSCW support to *component schema relationships* enables coordination among the involved analyst teams. Unlike the SPE solution, where the *component schema relationships* are represented in terms of IBIS structure realised on top of a Lotus Notes database, all changes lead to a replication of the affected SPE artefact to all involved analyst teams. Notification between distinct analyst teams in terms of information subscription is not possible in the ISST SPE. On the other hand, this kind of solution is far more demanding with respect to the SPE environment. First of all, distributed access to repositories storing *component schemas* has to be provided. Then changes to the repository of *component schemas* have to be structured in terms of the ViewPoint framework [NKF94]. Finally, CSCW support in terms of the ContAct prototype has to realise the actual information subscription.

9.1.3 CASE Tool Support

This subsection details CASE tool support for *component schemas*. The most important requirement in this context is a realisation of the entire UML metamodel. But before going into the details of CASE-Tools we also have to generalise the concern how the definitions in this work can be mapped and realised in CASE-Tools.

Current off-the-shelf tools such as Rational Rose98 [Rat98] or SELECT Enterprise [Sel97] do not realise the entire UML metamodel. Both mentioned tools for instance neglect UML `Trace` relationships, as the central representation of *component schema relationships* in this work. Without going into the details of the mentioned CASE tool implementations, we classify the CASE tool support according to the following three aspects:

- Provision of a UML subset tailored for requirements engineering
- Repository implementation
- Scripting Language Support

9.1.3.1 Provision of the UML Subset Tailored for Requirements Engineering

Most off-the-shelf CASE tools implement all UML views, i.e. static structure, collaboration, interaction, state machine, use case, physical and activity view. Within this work, we selected the static structure, the interaction and the use case view, for requirements engineering. We further restricted these selected UML views, in order to provide a solution-neutral representation of requirements. All constructs related to a specific object-oriented programming language have been rejected within *component schemas*, such as the C++ `friend` relationship. Thus model management of the respective CASE tool shall support the restricted UML views as described in Section 5.2. In this context we suggest a realisation providing different modes, i.e. a requirements engineering mode only supporting the described UML subset for *component schemas* and a design and implementation mode realising the entire UML specification, i.e. including programming language specific aspects, necessary for design and implementation of the future system. Whenever tools do not implement the UML `Trace` relationship, the UML `Dependency` relationship may be used for a realisation of *component schema relationships*. However, this leads to restricted expressive power of *component schema relationships*, since the UML only allows connecting `Classes` and `Packages` with `Dependencies`. Consequently, *inter-construct kind* and *inter-construct level relationships* among *component schemas* cannot be adequately represented.

9.1.3.2 Repository Implementation

The repository realises the persistent storage of UML *component schemas*. As outlined in Chapter 6, problem analysis models have a primarily linguistic character, i.e. they mostly structure the *terms* in a method-specific form. Automated detection of *component schema relationships* requires repository access as the central prerequisite for tool support. Since off-the-shelf tools do not offer linguistic support, repository entries provide access to linguistic information. In this context we may identify differences in the format and the access of repositories. The repository format is related to the way the repository is stored on the hard disk, e.g. the file format. In this context we may identify binary and ascii-text formats as well as database systems as possible repository formats. So-called single-source repositories provide another interesting realisation of persistency, since *component schema* information is directly represented in code frames of the target programming language. Besides the format, the access to repository contents is the other central distinction in this context. Repository access can be divided into a stand-alone, centralised and shared form.

Stand-Alone Repositories

Stand-alone repositories are the most primitive implementation for storing *component schemas*. Regardless of the actual format, *component schemas* are stored on the local hard disk of the host computer and can only be accessed by a single user via the used CASE tool. Off-the-shelf tools such as Rational Rose 98 [Rat98], SELECT Enterprise [Sel97] or Together/C++ [Tog99] take this approach.

Shared Repositories

Shared repositories allow model access from remote clients. In this context there are two possible implementations for shared repositories. The first relies on a relational database as a persistent storage engine, which has been used for Software through Pictures (StP) [StP99]. The UNIX version uses Sybase relational DBMS and therefore enables remote client access to the repository. The other potential realisation of shared repositories relies on OMG's CORBA [OMG98] and the *OA&D CORBAfacility Interface Definition* [OMG97d]. Combined with a proprietary repository, this kind of implementation allows access to all model elements stored in a repository. Using the *OA&D CORBAfacility* provides the best implementation, since it allows remote access to all UML metamodel elements and also allows a direct realisation of *component schema relationships* as stereotyped UML `Trace` relationships.

Binary Repository Format

The repository format defines the way given repository managers realise the persistent storage of *component schemas* on the hard disk. The repository format is especially relevant for the extraction of *terms* labelling *artefacts* in *component schemas*. This linguistic information is the prerequisite for provision of linguistic support. Analysing the repository implementation is the prerequisite for the extraction of *terms* labelling *artefacts*. In this context, binary repository formats usually allow fast access to elements of *component schemas*. On the other hand, the binary format needs a detailed format specification in order to access *component schemas* from external programs and is highly dependent on changes of the actual format chosen by the tool vendor. This problem also becomes apparent in the context of text processing, where import filters always have to be updated when a new version arrives on the market. Therefore, the efforts for maintaining filters in order to provide access to *component schemas* can be regarded as fairly high. Currently there is no off-the-shelf CASE tool using a binary repository format.

Ascii-Text Repository Format

Unlike binary repository formats, ascii-text formats are easier to comprehend and consequently easier to access. As depicted in Figure 105, p. 201, where the extract of a Rational Rose 98 repository represents the CNS/ATM Data Dictionary, ascii formats can easily be parsed in order to provide access to external tools. Parser generators, such as lex and yacc in the UNIX environment, can be used for writing filters that allow the extraction of linguistic information stored in a repository. Usually asii-text formats are far more robust with respect to format changes and therefore favour access for external tools. As mentioned above, Rational Rose 98 [Rat98] and SELECT Enterprise [Sel97] realise their repositories in an ascii-text format.

Single-Source Repository

An interesting variant to the ascii format repository representation has been chosen in the CASE tool Together/C++ from Object International Software [Tog99]. While Rational Rose and SELECT Enterprise separate the repository from the generated code, Together/C++ proposes a so-called single-source repository, where model information and the generated code are stored in a single repository. UML static structure elements are directly represented in terms of C++ header files and UML model management information. For instance, the positions of individual class specifications on the screen are stored in comments of the respective code frames. Although this representation is not solution-neutral with respect to the used program-

ming language, it represents an interesting alternative to all other discussed repository formats. The specific value of this representation lies in the stability of its essential information, since programming languages such as C++ are vendor independent and therefore the repository format can be regarded as the most stable one, compared to all other representations.

Figure 105 Extract of a Rational Rose 98 Repository

Class `ItemReference`

```
(object Class "ItemReference"
    quid       "36E91DA4C948"
    class_attri3butes (list class_attribute_list
    (object ClassAttribute "chapter"
      quid       "36E91DC00C70")
    (object ClassAttribute "section"
      quid       "36E91DC76C20")
    (object ClassAttribute "paragraph"
      quid       "36E91DCB2A90")))
```

Association `referencedIn`

```
(object Association "referencedIn"
   quid       "36E9047AEE00"
   roles      (list role_list
     (object Role "$UNNAMED$0"
       quid       "36E9047B6A80"
       supplier   "Document"
       quidu      "36E904192890"
       client_cardinality (value cardinality "0..n")
       is_navigable TRUE)
     (object Role "$UNNAMED$1"
       quid       "36E9047B6A81"
       supplier   "Item"
       quidu      "36E903403910"
       client_cardinality (value cardinality "1")
       is_navigable TRUE)))
```

9.1.3.3 Towards a Formalised Representation of Component Schemas

Although it is not the intention of this section to offer a formal mathematical model for *component schemas* and *component schema relationships*, we at least want to describe starting points for improved and rigorous CASE tool support. UML in its current version has been defined on a metamodel, which we consider as „intuitive“ semantics and therefore cannot be regarded as sufficiently formally rigid. A formalisation of our UML subset used for requirements engineering therefore has to be considered as a first and important step in this direction. Temporal object logics in the sense of Lamsweerde [LDL98] or graph grammars in the sense of Whitmire provide formalism for proper mathematical foundations of the UML. But formalising UML is a rather ambitious endeavour, since the amount of UML constructs probably exceeds a single PhD thesis.

Towards a Formalised Representation of Correspondence Assumptions

Besides the formal model for the UML there is another important topic of this work that has to be refined. Our definition of equivalence of class specifications stemming from different *component schemas*, a so called *correspondence assumption*, requires further investigation. Currently our definition of *correspondence assumptions* entirely relies on the meaning of a given *term*, i.e. the *concept*, that may be retrieved in a domain ontology [Gua98]. Due to the recursive relationship of *primary* and *secondary predicators* (see Figure 49, p. 105) new metrics have to be defined that reflect the *linguistic meaning* of class and attribute names. In other word structural equivalence for class specifications, expressed through the equivalence of *domains* (see Definition 29, p. 123) of attributes and operations [Spa94, Whi97] has to be combined with a new metric that takes the linguistic meaning of class and attribute names into account. Another important aspect is the possibly to integrate different *domain ontologies* in sense of Wiederholt's algebra for ontology composition [Wie94a]. This means the integration of *component schemas* may directly lead to an integration of the used expert terms in the domain ontology.

Towards a Formalised Representation of Conflict and Coherence

Our definition of *coherence* and *conflict* also needs careful revision. Both definitions are inherently vague and this vagueness is the central problem in this context. The *coherence* of two *component schemas* can only be proved indirectly in this work, by checking the state of *component schema relationships*, which has been realised in the UML Tagged Value `coordinationStatus`. This check does not incorporate the „real" relationship between given class specifications stemming from different *component schemas*. This check neglects *structural* in the sense of [BLN86] as well as *linguistic aspects* that indicate the equivalence of given class specifications. In brief the definitions of *conflict*, *coherence* and *correspondence assumptions* can be improved and equally important automated through sophisticated metrics taking structural as well as linguistic aspects of class specifications in to account.

Finally there are still elements left that are not yet defined in the UML, since *system* and *partition boundarie*s have not been defined as constructs in their own right [OMG97a]. Year Wand and Ron Weber [WW93], however, consider them as a vital *construct* for all modelling techniques. Instead of a *system* or *partition boundary construct* UML proposes *stereotypes* as means for distinguishing class specifications inside the system boundary from those ones outside. Class specifications marked with the stereotype «`interface`» refer in a *component schema* to class specifications outside the *system boundary*, while UML packages marked with the stereotype «`partition`» refer to entire problem partitions. Summarizing can be said, that the definition of new metrics that combine linguistic *with* structural aspects will lead to an improved definition of *coherence* and *conflict* and therefore offers new possibilities for an automated tool support.

9.1.3.4 Configuration Management

Concurrent requirements engineering also involves aspects of configuration management, as detailed in this subsection. Configuration management, as part of software engineering, deals with maintaining the set of all involved documents forming a complex product in a mutually consistent manner. With respect to concurrent requirements engineering, the entire problem specification has been decomposed into a set of partitions, represented as *component schemas*. So the overall product in requirements engineering is formed by a set of *component schemas*. Due to their concurrent elaboration, *component schemas* forming the entire problem analysis model are updated at different times and not necessarily with the same frequency. Thus *component schema relationships* represent references among elements of *component schema* that need to be maintained consistently. Susan Dart outlines the following operational aspects of configuration management [Dar90, p. 1]:

1 Identification:
an identification scheme is needed to reflect the structure of the product. This involves identifying the structure and kinds of component, making them unique and accessible in some form by giving each component a name, a version identification, and a configuration identification.

2 Control:
controlling the release of a product and changes to it through the life-cycle by having controls in place that ensure consistent software via the creation of a baseline product.

3 Status accounting:
recording and reporting the status of components and change requests, and gathering vital statistics about components in the product.

4 Audit and Review:
validating the completeness of a product and maintaining consistency among the components by ensuring that components are in a appropriate state throughout the entire project life-cycle and that the product is a well-defined collection of components.

All four aspects apply to concurrent requirements engineering, although the number of involved products is significantly smaller compared to system implementation and operation. In this context identification of involved documents and version identifications are the central prerequisites for concurrent requirements engineering using *component schemas*. With respect to *component schemas*, we have entirely left out version identification so far.

The most important CASE tool feature in this context, independent of the concrete repository access and format, is version management. Autonomy of analyst teams and the ongoing requirements elicitation and negotiation process demand a stable basis for assessing the coherence of the overall problem analysis model, split into a

set *component schemas*, at given moments in time. Milestones in the project plan or just routine checks initiated by the project management may be the reason to detect *component schema relationships* before the actual end of requirements elicitation and negotiation. Essentially, the incremental development of *component schemas* in object-oriented requirements analysis tends to neglect the traceability of defined stages in the analysis of *component schemas*. Defining the change sets between milestones is an additional reason to provide version management. Rational Rose offers version management through an interface to standard tools such as RCS [Tic85] or SNiFF+ [SNi96]. This feature is especially important for the detection of *component schema relationships*. A version provides a defined context for set a of *component schema relationships*. Otherwise, the autonomy of analyst teams leads to a constant evolution of *component schemas* that in the worst case may constantly override *component schema relationships*. Thus versions can be regarded as synchronisation points among the involved analyst teams in order to fulfil a complete management cycle for *component schema relationships*.

9.1.3.5 Scripting Language Support

Unlike repository access, where *component schema* access is discussed independently of the tool creating it, scripting languages such as Microsoft's ActiveX or Java Script, provide access to elements of *component schemas* on the level of tools. Although access to *component schema* in this context is restricted to the capabilities and especially the openness of the used CASE tool, scripting languages are more robust with respect to changes of the repository and therefore provide more stable access to elements of *component schemas*. The other positive aspect of scripting languages lies in the computational completeness of the language, i.e. they offer data types, variables and flow control to the users. All in all scripting languages offer a flexible mechanism for accessing tool repositories and also provide user-defined extensions that cannot be realised otherwise. Rational Rose 98 [Rat98], for instance, supports Microsoft's VisualBasic for Applications, which has been renamed ActiveX.

9.2 Realisation

Tool support for concurrent requirements engineering can be divided into different alternatives. Integrated tools offer the described functionality in a single tool, incorporating linguistic support and a communication infrastructure. Alternatively tool support can be built using off-the-shelf tools and realising the concrete integration in terms of small programs realising the required *'glue code'* for the integration of independent and distinct components.

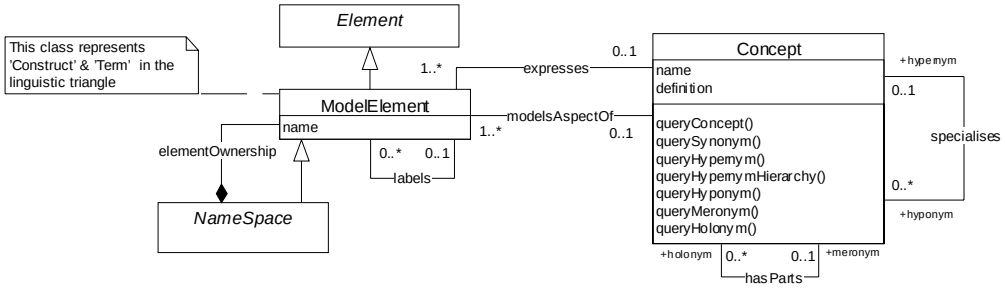
9.2.1 An Integrated Concurrent Requirements Engineering Workbench

In this subsection we outline the basic components of an integrated concurrent requirements engineering workbench. Following the distinction of Section 9.1, the basic components comprise a realisation of the entire UML metamodel, linguistic support and groupware functionality realising the information subscription between the involved analyst teams. We believe that the outlined components, linguistic support is the most important one, especially for supporting industry-scale applications. Current off-the-shelf components do not provide minimal linguistic support in terms of glossaries and data dictionaries. Although it is a matter of debate whether a tool should offer integrated data dictionary support or only provide external interfaces, the management of expert terms is one of the crucial success factors in complex application domains such as air traffic management. In this work we propose integrated support, leading to enhancements of the current version of the UML metamodel.

Enhanced UML Metamodel for Dictionary Support

In order to enable clarification of the *terms*, a set of synonymous *terms* and their meaning – the *concept* – has to be provided for each *artefact* in the *component schema*. With respect to the UML metamodel, the *term* labelling an *artefact* can be found in the property `name` in metaclass `ModelElement`, as part of the UML package `Core`. The *construct* applied to an *arteface* can be distinguished through its actual metaclass, i.e. the actual metaclass a model element belongs to. So we have to provide the additional metaclasses `Concept` and `Term`. Removing the abstract character of metaclass `ModelElement` leads to a simplification of the resulting metamodel. Allowing instances of metaclass `ModelElement` that are not further specialised, i.e. they do not lead to concrete UML constructs allows us to remove the metaclass `Term`. Consequently the relationship `labels` has to be changed into a recursive one for the metaclass `ModelElement`. In order to keep the distinction between *artefacts* and their real-world meaning, we propose to keep their real-world meaning in the separate metaclass `Concept`.

Figure 106 Integrating Concept and Term to the UML Metamodel



In order to increase flexibility, the linguistic relationships *hypernomy* and *meronymy* also have to be realised on a metamodel level. This leads to the recursive relationships `hasPart` and `specialises` linked to metaclass `Concept`. Although at first glance, these relationships seem to replicate the entire dictionary structure, they

enable the clarification of terms independently of the availability of a domain ontology. Instead of querying an ontology, the relationships among *terms* have to be defined by the tool users. Analysts can only browse through hierarchies of *terms*, without an automatic classification of given input terms. Metaclass `Concept` already incorporates operations for interaction with a domain ontology. In other words, this metaclass can be regarded as a proxy for an ontology server. Figure 106, p. 205 depicts the revised UML metamodel.

The proposed enhancements to the metamodel integrate linguistic support on a metamodel level. The choice of an adequate domain ontology now allows a clarification of terms. In order to guarantee the domain-independent character of CASE tools we outline a primitive interface to external ontology servers. Whenever an ontology is used, we assume that the linguistic relationships `specialises` and `hasPart` are not instantiated, in order to be flexible with respect to changes in the underlying ontology. On the other hand, this representation also allows manual classification of expert terms without using a concrete domain ontology and therefore increases the flexibility of linguistic support.

Repository Implementation

After the integration of linguistic support on the metamodel level, we have to focus on suitable repository implementations. In this context we propose a repository implementation based on OMG's *OA&D CORBAfacility* [OMG97d]. This implementation guarantees repository access to remote clients via CORBA's object request broker. Consequently, stereotyped trace relationships as our implementation for *component schema relationships* may link *artefacts* in distributed repositories. Versioning, as discussed in Section 9.1.3.4, is another important tool requirement, not yet addressed in the *OA&D CORBAfacility* [OMG97d]. While the CORBA interface guarantees repository access for remote clients, elements of *component schema* also need a distinct version identification. Due the evolution of concurrently elaborated *component schemas*, detected *component schema relationships* require defined terms of reference, which can only be applied through version management. Consequently, the repository manager has to implement version management and therefore keeps track of the actual versions of model elements related via a *component schema relationship*.

Communication Infrastructure

Similar to the repository implementation, CORBA also facilitates the implementation of the communication infrastructure. Information subscription can now be realised on a distributed implementation of Gamma's observer pattern [GHJ+95, pp. 293], based on CORBA's event service [OMG97f]. Without detailing the concrete realisation, there are principally two variants for this implementation. In the first, the state of the *component schema relationship* is observed, allowing project management to monitor the coordination among affected analyst teams. The second variant also includes the elements of *component schema*. In this context, changes to the artefacts can also be monitored by interested analysts. This approach is very promising with respect to *inter-partition requirements*, which demand increased attention, especially in industry-scale projects. As described in Chapter 8, the actual independence of individual analyst teams is especially restricted by *inter-partition requirements*.

Whenever individual inter-partition requirements remain unsettled, the consistency of the overall problem specification cannot be assured. Therefore information subscription based on a CORBA implementation of a distributed observer pattern is the most promising solution for an integrated CASE tool for concurrent requirements engineering.

9.2.2 A Concurrent Requirements Engineering Workbench Based on Off-the-Shelf Tools

The used CASE tool can be regarded as the backbone for a requirements engineering workbench based on off-the-shelf components. We assume that these tools realise a stand-alone repository in an ascii-format. From this point on, we discuss how linguistic support, *component schema relationships* and coordination support may be realised in order to support analyst teams in concurrent engineering projects.

Linguistic Support: Accessing Repository Information

The lack of linguistic support in off-the-shelf CASE tools requires external solutions, to realise data dictionary functionality. Consequently, *terms* labelling *artefacts* have to be extracted from given *component schemas* and stored in an external data dictionary. An extraction of *terms* labelling *artefacts* can now be realised following two different strategies depending on the actual tool capabilities. For CASE tools supporting scripting languages, simple programs may retrieve all *terms* labelling *artefacts* from *component schemas* at hand. With the *term* labelling the *artefact* at hand, we also have to extract the UML construct labelled by this *term*. This information can either be stored in an ascii format file or fed directly into a data dictionary. Figure 107 depicts a sample script written in VisualBasic for Applications retrieving the class names in Rational Rose98. CASE tools without scripting language support using ascii-text format repositories require different extraction techniques. In this context the repository has to be parsed using an external tool. The *terms* labelling *artefacts* have to be extracted and stored either in an intermediate file format or directly in the data dictionary. UNIX tools such as lex and yacc provide a suitable basis for the implementation of this kind of parsers for ascii-text repository formats.

Figure 107 Sample Program Retrieving all Class Names in a Component Schema for Rational Rose 98

```
Sub Main
   ' Set up the array used to initialize the list box control
   For i% = 1 to RoseApp.CurrentModel.GetAllClasses.Count
      AllClassNames$(i% - 1) =
                RoseApp.CurrentModel.GetAllClasses.GetAt (i%).Name
   Next i%
End Sub
```

Linguistic Support: Data Dictionaries and Ontologies

Scripting languages or parsers realise the extraction of *terms* labelling *artefacts*. These *terms* are then fed into a database system realising the linguistic support. For a clarification of the concrete meaning of *terms* labelling *artefacts*, we now have to

integrate lexical databases or available domain ontologies. In this context we propose external tools for retrieving the *concept*. Interfaces to WordNet [WN98] as a thesaurus implementation or an ontology server such as OntoSaurus [Sen95] therefore have to be provided.

Component Schema Relationships in Distributed Stand-alone Repositories

The stand-alone character of tool repositories prevents direct linkage of elements of *component schema* with the detected *component schema relationship*. Principally there are two different possibilities for realising them. In the first option the tool repositories containing related *component schemas* have to be updated. In this context, an *artefact* in the owned partition is related to an *artefact* in a target partition. So the related *artefact* from our owned partition has to be replicated to the repository containing the *component schema* of the target partition. On this basis we can also realise *component schema relationships* on top of distributed stand-alone repositories. However, depending on the mapping of partitions to *component schemas* different activities have be accomplished. *Component schemas* realising package-based partitions already comprise all *initial partitions*, represented as UML packages. So a related artefact from an owned partition can be replicated to the package representing the owned partition in the target *component schema*. *Model-based partitions* have to be changed into *package-based* ones before the replication of the affected *artefact* from the owned partition can be realised. Although there is no risk of directly overriding class specifications in the target *component schema*, this solution still requires direct access to the involved repositories of the used off-the-shelf CASE tool.

The IBIS structure, as described in Section 9.1.2.2, provides realisation of *component schema relationships* without accessing the repositories of distributed stand-alone repositories. Independent of the CASE tool's repository realisation the IBIS structure on top of Lotus-Notes realises *component schema relationships* and integrates the coordination functionality at the same time. The ISST-SPE provides a particularly suitable environment for the management of all involved documents in a concrete requirements engineering project, since version management can be done automatically. On the other hand, this solution requires a set of small tools for the extraction of *terms* labelling *artefacts* and import filters to the Lotus Notes databases.

9.2.3 Section Summary

Tool support for concurrent requirements engineering contains three distinct components: a CASE tool, linguistic support and a coordination infrastructure realising the notification of the involved analyst teams. These three components may be realised in a single integrated tool or as an environment built from off-the-shelf components. The integrated tool realises *component schemas* in terms of distributed objects and therefore closely follows Nuseibeh's [NKF94] characterisation of ViewPoints as distributed encapsulated objects. *Component schema relationships* become references to distributed objects maintained by an object request broker. In this context, we have to refer to the diploma thesis of Ute Schirmer, where she describes all necessary slots of the ViewPoint Framework for our selected UML subset [Schi98]. Combining the

results of this diploma thesis with our definition of *component schema relationships* and the described linguistic support may bring the ViewPoint Framework to life. The second possible tool concept involves a set of off-the-shelf components and a set of filters realising the necessary data exchange between previously unrelated tools. The central drawback of this solution is the dependency on the vendors repository format and their stand-alone character. In this context, the management of data dictionaries becomes an important question, not dealt with in this work. In a master-multiple organisation, the master-team maintains the data dictionary and therefore provides centralised control of the expert vocabulary. However, there is a central problem with respect to versioning of the vocabulary. Each assessment cycle may lead to new meanings of particular terms. Therefore, feeding *terms* labelling *artefacts* directly into the data dictionary may lead to another uncontrolled overriding, similar to elements of *component schemas*. This problem can be smoothed out by using the ISST-SPE with IBIS structure. Here the detection and maintenance of *component schema relationships* has to be accomplished by members of the master-team and therefore remains under analyst's control.

10 Validation of UML in the Air Traffic Management Domain

In this chapter we validate our concurrent requirements engineering approach under three distinct perspectives. Our experience gained in a number of projects in the air traffic management domain [Wor96, WCK97,WF97,WFW97] provides the main criteria for this validation. First of all we discuss the specific value of *component schemas* for the air traffic management domain. In this context we are especially interested in the expressive power of UML for problem analysis. We summarize deficiencies of UML with respect to requirements engineering, as encountered in our projects. Then we discuss the influence of the outlined tool environment for running industry-scale projects. Based on the first two subsections, we outline advantages as well as deficiencies of our proposal. Finally, we discuss the generation of software requirements specifications on the basis of *component schemas.*

10.1 Requirements Analysis in the Air Traffic Management Domain Using Component Schemas

Object-oriented analysis techniques have been used in a number of different requirements projects in different domains. Although almost every text book in requirements engineering presents small and simplified examples related to the application of object-oriented analysis techniques in air traffic management [Dav93, SS97], there is little documented experience with respect to real projects and so far there is no entirely object-oriented project on its way. Even though Ball and Kim [BK91] published the first study presenting an object-oriented analysis model for the air traffic management domain in 1991, there is still a considerable difference with respect to our own studies [Wor96, WCK97,WFW97]. Ball and King mostly concentrate on the structural aspects of air traffic management, while behavioural aspects fall short. The structural part of their model resembles a taxonomy of expert terms, i.e. a precursor to a domain ontology, represented in the Coad Yourdon notation [CY91]. The frequent use of multiple inheritance in their model provides suitable indication that this model was never meant for design. Introducing use cases to requirements elicitation was a novel approach in this field. This subsection basically reflects our experience concerning the use of UML in the air traffic management domain.

Use Case Analysis in ATM

Although the value of use cases in requirements elicitation has been widely stated in a series of references [JCJ+94, RAB96, AEQ98], there are still considerable deficiencies with respect to requirements engineering. Especially the C^3I character of air traffic management systems led to a number of challenges in these projects. A major drawback with respect to use cases soon became apparent. Use cases documenting the interaction of controllers and pilots during sector hand-over contained a number of related activities that appeared in nearly the same order in a number of use cases.

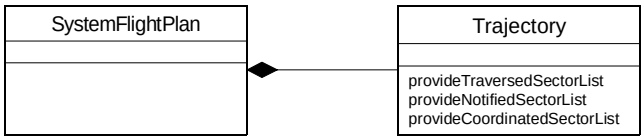
This observation gave the impetus to the development of our complex use cases. Regnell's *use case episodes* [RAB96] propose a decomposition mechanism for related activities *within* a single use case. Activities in air traffic management, however, demanded construction mechanisms for complex activities built on top of atomic ones. Jacobson's original use cases with their *extends* and *uses* relationships do not offer suitable means for the construction of complex use cases. Therefore we defined atomic use cases as self-contained entities, with preconditions determining the conditions to be met before their invocation and a defined state after their termination through postconditions. *Path expressions* [CH74] finally provided a suitable construction mechanism for complex interaction built on top of self-contained atomic use cases. With respect to the decomposition of user interaction, John Arlow [AEQ98] refers to the risk of mimicking functional decomposition techniques in terms of use cases. This may result in too fine-grained descriptions of system behaviour and may therefore threaten the solution-neutral character of requirements specifications. Although this risk tends to be a question of experience, in object-oriented analysis as well as the domain at hand we believe that our composition mechanism provides adequate means for a composition of complex user activities on the basis of atomic units. Nevertheless, finding the *right* level of abstraction will remain a matter of analyst experience and therefore remains outside the scope of this work.

The other drawback of use case driven object-oriented analysis is related to the missing prioritisation of requirements represented in *component schemas*. John Arlow calls this problem the *trivialisation of requirements* [AEQ98]. Although from a *component schema* perspective this observation is generally true, this statement needs further consideration. At first glance all model elements represented in *component schemas* have the same relevance. Arlow's statement on the other hand, only holds if the analysis model already comprises the *entire* requirements basis and no information remains outside the model. The kind of requirements already represented in *component schemas* are detailed in Section 10.3.1. With respect to use cases alone, we tackled this problem using two different strategies. First of all, use case activities became part of the metamodel and therefore may be implemented in the repository of a CASE tool. Our approach overcomes the entirely illustrative character of original use case diagrams in the UML semantics document [OMG97a]. As metamodel elements, individual use case activities may be linked to their derived elements, depicted in sequence as well as static structure diagrams. Consequently, making individual activities unique in the larger context also allows a prioritization of requirements, although we have to admit that this cannot be done in a *component schema* alone. We have the feeling that prioritization seems to be an issue of the accompanying documentation set, such as software requirements specifications (see Section 10.3). But the integration of textual use case descriptions into the metamodel overcomes the criticised *"clumps of functional requirements"* as John Arlow puts it in a discussion on the SRE listserver from Feb 24. 1999. Individual activities can be uniquely identified through the UML name space concept and also become traceable within *component schemas*, which is supported neither in the original OMG specification [OMG97a] nor in the revised metamodel [Rat99].

Operations, Behavioural Requirements and Control Objects

The two quoted EUROCONTROL operational requirements documents [Eur96a, Eur97] followed functional decompositions techniques and represent behavioural requirements only from a structural perspective. They describe how functions may be decomposed, but their interaction remains undocumented. Compared to ORDs, the representation of behavioural requirements in *component schemas* stresses system interaction through use case activities and further structures these activities in terms of sequence diagrams. But independently of the applied level of detail, operations in the resulting *component schemas* still have a fairly high-level character and even most complex calculations end up as a simple operations in given class specifications. Figure 108 illustrates this claim:

Figure 108 Granularity of Operations in Component Schemas



All three operations in the class specification `Trajectory` can be directly related to statements in the Flight Data Processing ORD [Eur97]. The operation `provideTraversedSectorList` calculates all traversed sectors for a given flight, based on the information derived from its filed flight plan. However, all three operations have to take the current airspace configuration into account. The airspace configuration is one of the most complex geographic objects in this context and the calculation of entry and exit points involves complex calculations. Planning ahead also involves taking a potentially different airspace configuration into account. Thus the actual airspace configuration always has to be retrieved from the database for given flights, further increasing the effort. Erronenous calculations in this context may lead to critical situations, especially when sector controllers are not informed by the system that given flights will cross their assigned control sector. So operations in component *schemas* do not immediately reflect their enormous importance with respect to the overall system. In this context, operations provide an excellent example of the claimed *trivialisation of requirements* [AEQ98].

In order to stress the importance of complex operations, we may represent them as a class specification in its own right, similar to Jacobson's *control objects* [JCJ+94]. Referring to the calculation of traversed control sectors, which incorporates the current airspace configuration, it is neither part of the trajectory nor part of the air space configuration. A control object therefore seems to be a logical consequence. The same observation holds for the entire ATM Advanced Function domain, where monitoring aids, safety nets and arrival manager mostly represent complex calculations. In this case we represent these functionalities as control objects, since all three functions incorporate different algorithms, which are mostly working on the same set of class specifications. Nevertheless, *component schemas'* representation of behavioural requirements still has to be considered as superior, compared to the entirely structural

perspective of behaviour applied in functional decompositions. This observation leads to another encountered problem, the representation of *global* and *inter-partition requirements*.

Global and Inter-Partition Requirements

The concurrent elaboration of *component schemas*, based on ORDs, brought up this important issue. ORDs usually reserve a distinct chapter for the documentation of dependencies between partitions and external systems. We also had to find an adequate way to represent these dependencies in *component schemas*. Our solution led to a facade-based representation of inter-partition requirements, which represents the *owned partition* as well as related partitions as class specifications in their own right. Essentially behavioural *inter-partition requirements* are represented as operations attached to these facade classes. Most partitions maintained dependencies to a number of related partitions, leading to different class specifications representing the demands from the perspective of various partitions. In order to provide uniform access to services offered for dependent partitions, all facade-classes representing the various demands become subclasses of an abstract class representing the domain. Upward inheritance then leads to a unified facade-class representing all services offered by a given domain. The representation of global requirements in this context depends on the mapping of partitions to *component schemas*. *Model-based partitions*, due to their self-contained character, automatically restrict the validity of requirements to the partition at hand and therefore fail to represent global requirements. *Package-based partitions* do not suffer this problem, since all initial partitions are mapped to individual packages grouped in the top-level UML package «`system`». So the UML package «`system`» comprising the entire problem analysis model provides a suitable place for representing global requirements.

10.2 The Project CNS/ATM Overall Analysis Model

While the previous subsection validated the use of *component schemas* in requirements engineering, this subsection is dedicated to organisational aspects of concurrent requirements engineering projects and our specific experience gained in the air traffic management domain.

EUROCONTROL – Coordinating Multi-National Expert Teams

One of the driving forces behind the applied project organisation in our reference projects is the supranational character of the European Organisation for the Safety of Air Navigation – EUROCONTROL – with its current total of 28 member states. This specific character also significantly influences EUROCONTROL's project organisation. Usually a distinct member state is responsible for the elaboration of given partitions within the overall project frame. The distinct responsibility for given partitions gave rise to the conflicting objectives '*coherence of the overall problem specification*' versus '*autonomy of analyst teams*' studied in this work. The concrete project organisation therefore led to self-contained partitions elaborated by autonomous analyst teams, following functional decomposition techniques. The autonomous project organisation, in a multinational project, lead to a number of subtle differences in the individual ORDs. Often, the perspective on specific requirements was influenced by

national standards and procedures. So on an organisational level, concurrent requirements engineering in autonomous analyst teams with a master-multiple organisational model was the only possible project organisation in this context. The master-team provided consultancy in the application of UML and was not directly assigned to a given partition.

ORDs versus Component Schemas

The central problem in this context was assessing the quality of the individual ORDs with respect to the coherence of the overall specification. Entirely textual requirements specifications complicate this assessment, due to their lack of problem analysis models. We believe that adequate problem analysis models with CASE tool support provide compact and concise road maps to complex requirements documents, regardless of the applied modelling technique. Object-oriented techniques, however, have important advantages, especially with respect to systems integration. The current European ATM infrastructure has been built entirely under the authority of the national administrations. Within the administration, two basic strategies have been followed. The first one, applied in France, incorporates analysis, design and implementation under the authority of the national administration. The second one, applied at Deutsche Flugsicherung, comprises analysis under the authority of the national ATM service provider and customisation of off-the-shelf products according to the stated requirements. Besides these different national procedures, most systems only provide interfacing in terms of voice communication via telephone lines to external systems. The increasing traffic rates in central Europe and the scarce resources airspace and airport capacity require an exchange of on-line information between the involved systems. This ubiquity of information cannot be supported by functional decomposition techniques focusing on functionality instead of data. Moreover, the variety of national standards and procedures further hinders the harmonisation of the functionality on a European level.

Ubiquity of Information

Taking two modern ATM systems, the functionality is roughly the same, but the individual grouping of functional blocks varies significantly. Instead of harmonising the functional decomposition in lengthy discussions, it proved to be more efficient to represent the information stated in the ORDs directly in terms of an object-oriented analysis model using *component schemas*. Compared to the elaboration efforts spent for each ORD, representing all seven ORDs in terms of *component schemas* required only 18 man months [Eur98a, p. 2], while each ORD incorporated approximately the efforts of 7 man years! Besides the little effort spent for their creation, *component schemas* also incorporated complementary information to the existing ORDs. Sufficient requirements source traceability may overcome the criticized trivialisation of system behaviour as modelled in operations. The primarily linguistic character of *component schemas* does not really reflect the complexity of involved algorithms behind a simple operations name such as `provideTraversedSectorList`. ORDs, on the other hand, are too detailed with respect to the demanded functionality and therefore tend to fall into premature design. So in this context, ORDs and *component schemas* can be regarded as complementary information. The specification of structural requirements is clearly superior in *component schemas* when compared to

ORDs following functional decomposition techniques. Object-oriented analysis models are therefore more likely to tackle the ubiquity of information, one of the central demands of future integrated ATM systems.

Coherence in Partitioned Requirements Documents

The quality of individual ORDs describing distinct problem partitions and the coherence of the overall problem statement are too complicated to assess entirely on the basis of textual requirements statements. Especially on the structural level, *component schemas* improved coherence among the involved partitions dramatically. Severe differences between the individual expert groups' perspectives were able to be detected and solved during review sessions organised by the master-team. Hidden as well as explicit dependencies between the involved partitions were revealed. The explicit ones became apparent in facade-classes representing *inter-partition requirements*, while hidden ones mostly emerged in terms of conceptual overlaps. Consequently, coherence as well as minimality of the specifications could be improved through the application of *component schemas* as concise and compact road maps through complex textual requirements.

Lacking Tool Support

Adequate tool support for concurrent requirements engineering in industry-scale application domains proved to be complicated with respect to two different aspects. Firstly, the repository implementation and secondly, of all the support for *component schema relationships* provided the central motivation. Industry-strength tools, such StP UML [StP99], could not be used on notebook computers due to the repository implementation, which is realised on top of the Sybase RDBMS. This concept results in horrendous resource consumption, which requires workstations on a LAN as adequate working environment[1]. Rational Rose98 [Rat98] with its ascii-text repository, on other hand, worked pretty well on notebook computers, but did not offer the repository extensibility provided by StP. Supporting concurrent requirements engineering and *component schema relationships* provided the other important aspect. This work contributed to this problem in several ways. First of all, proposing linguistic support to the analysts facilitates detection of *component schema relationships* both manually, and on an automated basis. Independently of the chosen support (i.e. data dictionary, thesaurus or ontology) the inherent complexity of expert language in a given domain has to be mastered in order to guarantee project success. On the basis of linguistic support, the definition of *component schema relationships* further improves concurrent requirements engineering. Means for assessing the similarity of *artefacts* and notification of involved analyst teams without actually overriding requirements is an innovation. *Component schema relationships* combined with the UML `TaggedValue` concept allow assessment of the remaining coordination efforts and therefore improve the management of autonomous analyst teams. Inte-

1 Tool evaluation took place in 1996, right at the beginning of the project! The new StP version for Microsoft Windows NT 4.0 works on a repository based on Microsoft's Jet Engine, better known from Microsoft Access for Windows

grated CSCW support in terms of information supply (see Section 9.1.2.3) further improves the coherence of autonomous project organisation, which cannot be reached otherwise.

Coordination among Multi-National Expert Teams

This improvement has also been extended to a management level through the identification of analyst horizon's and the resulting impact on minimality and coherence of the specification. Improving coherence without actually restricting the autonomy of analyst teams especially suits the complicated situation encountered in projects on a European level. Most text books concerning requirements engineering do not cover management aspects or only propose centralised organisations with clear responsibilities. Explicitly incorporating these aspects into the detailed analysis of concurrent requirements engineering further improves handling of complex project environments. This idea also takes national habits and the idiosyncrasy of individual analysts into account. We would like to stress the important influence of human relationships on overall project success. *Component schema relationships* as organisational entities suit the complex nature of industry-scale projects, in domains characterised by national diversity and heterogeneity on the level of existing ATM infrastructure.

Coherence versus Autonomy

Increasing coherence in autonomous project organisations significantly improves the maturity of the requirements process in the sense of Kotonya and Sommerville [KoS98]. The management of *component schema relationships*, as defined in Chapter 8, describes a defined process model for concurrent requirements engineering using *component schemas*. These processes are the central prerequisite for gaining higher levels of requirements engineering process maturity in terms of Paulk's *Capability Maturity Model* (CMM) [PWC95]. Sommerville and Sawyer propose an adoption of the *CMM* tailored for requirements engineering [SS97, KoS98]. The defined process, taking various project organisations into account and use cases as the central basis for requirements elicitation, automatically ranks our concurrent requirements engineering approach on the *repeatable level*. The only missing aspect in this thesis is process improvement, which has been excluded in this work. So the defined process under special consideration of project organisation has the potential to improve the quality of the outcome of given projects and achieving a higher maturity level at the same time.

10.3 Generating Software Requirements Specifications from Component Schemas

Problem analysis models generally aim to facilitate thorough understanding of complex problem statements for the involved stakeholders. *Component schemas*' detailed description of structural and behavioural requirements provide a compact and comprehensible description of the problem statement. The sheer size of industry-scale applications demands this kind of representation. This observation gives rise to important questions in the context of requirements engineering. The first one deals with the relationship between requirements in the broadest sense and their representation in *component schemas*. In this context we want to consider what kind of requirements have been captured already in *component schemas* and what kind of informa-

tion is possibly missing. We also focus on the question of how problem analysis models using *component schemas* may be converted into software requirements specifications. In this context we take the IEEE-830 standard [IEEE830] as our guideline. This question is mostly related to project efficiency, i.e. making optimal use of available information and minimizing rewriting the same information several times. According to our experience, problem analysis models already incorporate an extra value. They not only facilitate understanding of the intricacies of given problem statements, they also contain most necessary information for the generation of a software requirements specification. Especially the seamless transition from analysis to design, as the major claim in object-orientation, is one of the essential values of problem analysis models. Making optimal use of existing information, i.e. object-oriented problem analysis models using UML, does not automatically imply that no further work is necessary in order to receive a complete software requirements specification. So in the second subsection we basically evaluate the value of *component schemas* with respect to software requirements specifications.

10.3.1 Reflected Requirements in Component Schemas

A slight shift in our perspective towards components schemas will widen our scope from problem analysis towards software requirements specifications, as the central outcome of requirements engineering projects. While the major part of this work has focused on gaining coherent problem analysis models in the presence of autonomous analysts teams concurrently elaborating initial partitions, we come back to the question of the value of problem analysis models with respect to the generation of software requirements specifications. So we have to analyse what kind of requirements have already been modelled in our *component schemas* and which kind of information is still missing.

Following our definition of requirements in Chapter 4, we distinguish between *structural*, *functional* and *non-functional requirements*. Behavioural requirements are basically elicited in our complex use cases and may be transformed into UML `Messages` exchanged between objects in UML sequence diagrams. These objects, which are called `ClassifierRoles` in the UML metamodel, finally lead to class specifications in static structure diagrams. Although use cases model the interaction between stakeholders and the future system very well, we want to have a closer look at the way requirements are elicited in use cases and at their specific representation in *component schemas*. In this context we would like to refer to the diploma thesis of Dörte Godulla, which evaluates the deficiencies of software requirements specifications built from use case driven analysis models [God97].

Functional Requirements

Complex use cases, as described in this work, represent clusters of behavioural requirements. Preconditions define necessary conditions for their invocation, while postconditions describe conditions to be met after the having completed the last activity. Reactions to potential error conditions have also been described. All this information has been specified in textual form. Individual activities to be accom-

plished in the course of the analysed business process are the most interesting information. As a rule of thumb, nouns within these activities denote objects, while verbs correspond to *event predicators*or in the sense of Schienmann [Sch97]. So the verbs are represented as `Messages` in the derived sequence diagram. `Messages`, however, directly represent behavioural requirements, defining the functionality of the future system. Assigning an `Operation` to a `ClassifierRole` determines the behaviour to be implemented by a given `Class` in the static structure diagram. The construct `Class` in UML integrates the notion of behaviour and structure into a single construct. This gives rise to the question: where does the structure come from?

Structural Requirements

Pure use cases only describe the behaviour between a stakeholder and the future system in given business processes. Consequently, they do not contain any information for determining the static structure of an object referenced in the use case. This general deficiency of use cases can be mastered by annotations of individual use case activities. Whenever real-world things are significant in given use cases, a description of their structure has to be added. So extra information has to be added to individual use case activities, e.g. describing the internal structure of data elements referenced in it. The lack of structural requirements can only be compensated with this workaround.

Temporal Requirements

Non-functional requirements as described in Section 3.3.1 fall into temporal and non-temporal requirements [OKK97]. Since temporal requirements describe response times, we first have to clarify which model elements within our *component schemas* they actually apply to. Response times may be attached to entire business processes or to individual activities within these business processes. Both interpretations lead to different representations within the *component schema's* metamodel. Whenever the temporal requirement is attached to the entire business process, the use case modelling this process is the relevant model element in order to keep this kind of information. If individual activities are the relevant unit, then the metaclasses `Activity` and `Message` in sequence diagrams have to be adorned with this temporal information. Since non-functional requirements are clearly outside the scope of the UML specification, we propose a new tagged value that has to be attached to the identified metamodel elements. Tagged values may capture requirements in *component schemas* that have not been covered so far. So the tag `responceTime` may be attached to the metamodel elements `Activity`, `Message`, `Operation` and `DecomposableUseCase`. Besides having identified relevant metaclasses for capturing temporal requirements, we also have to stress that this kind of non-functional requirements have not been elicited in use cases so far. Consequently, temporal information may be gathered either within the initial creation of the use cases, or in a review session with stakeholders. The templates for complex use cases should also support this kind of information.

Non-Temporal Requirements

Non-temporal requirements describe aspects such as system availability and security [OKK97], two aspects of tremendous importance, especially in life-critical applications such as air traffic management. Kotonya and Sommerville extend this aspect with the properties *performance*, *reliability*, *size*, *robustness* and *portability* [KoS98].

Performance in this context describes the number of transactions per time unit, *robustness* quantifies the time to restart the system after failure and *portability* describes the number of target systems. *Component schemas* do not cover this information so far. Once again, we have to identify metamodel elements that are related to non-temporal requirements and also provide means to represent this kind of requirements in the problem analysis model. According to Oliver [OKK97], non-temporal requirements are bound to structural components. Consequently, this kind of information has to be bound to model elements in the static structure view and the UML metaclasses `Package`, `Class` and `Attribute` are the principal candidates. Since attributes only describe the structure of class specifications, they can be neglected. As a result, the tagged values `availability`, `performance`, `size`, `reliability`, `robustness` and `portability` have to be attached to the UML metaclasses `Package` and `Class`.

Design Requirements

Design requirements constrain design options without directly prescribing the entire solution. Such requirements may be stated by RDBMS or middleware products used. This category of requirements mostly applies to the entire specification. However, quite often, individual components of a system are based on different enabling technologies, leading to differing scope of this kind of requirements. Since *component schemas* can be regarded as an object-oriented representation of initial partitions, this kind of information can only be related to entire partitions. But how can design constraints applying to the entire specification be modelled, like to non-temporal requirements, design requirements apply to structural elements within *component schemas* and the UML metaclasses `Package` and `Class` are primary candidates. Unlike the tag `availabilty` incorporating a single value, design requirements may incorporate sets of values denoting various design constraints. Thus the corresponding `TaggedValue 'designRequirement'` may only contain a comma-separated list of values, represented as a string. Although this is a cumbersome solution, a defined place for capturing this kind of information has been proposed.

Design Requirements & Partition Mapping

The UML name space concept always restricts the validity of design requirements to the enclosed model elements in a package. So the distinction between *inter-partition requirements* and *global requirements* depends on the mapping of *initial partitions* to *component schemas*. *Model-based partitions* therefore only allow the specification of partition requirements. The validity of individual design requirements for the entire problem analysis model has to be coordinated among all involved analyst teams. *Package-based partitions* already incorporate the notion of the overall problem specification. Analysts may attach `TaggedValues` to the top-level package, indicated by the stereotype «`system`». But we have to stress that *inter-partition* as well as *global design requirements* are subject to extensive discussions among the involved analyst teams. Describing a representation in terms of the UML metamodel does not automatically imply their correctness. The complexity of design requirements generally cannot be discussed in terms of their representation. Furthermore, this is entirely subject to the experience and the judgement of the involved analyst teams. We also

have to stress that defining adequate locations for representing non-functional requirements improves the quality of the problem analysis model and also facilitates deriving software requirements specifications from *component schemas*.

Prioritization or Requirements

A major problem within object-oriented problem analysis is the so-called *trivilisation* [AEQ98] of requirements. The *component schemas* model functional as well as structural requirements of given problem statements and therefore provide a road map through complex requirements stated in various forms. But at first glance all elements of *component schemas* have the same priority. On a model level alone, the individual importance of class specifications and operations with respect to the overall problem statement does not become apparent. But this information is essential to risk management, for instance assessing impact of the failure of a system component with respect to the overall system. Another important deficiency from a software requirements specification point of view is the lack requirements traceability matrices. Although non-functional requirements can be attached to individual UML constructs, as discussed in the beginning of this section, mutual dependencies between functional and non-functional requirements cannot be expressed on a model level. Essential, mandatory and optional components within *component schemas* may be indicated using stereotypes. This approach works efficiently for class specifications and packages, but cannot be visualised adequately for operations. Regardless of whether stereotypes or tagged values are preferred by the project management, UML extension mechanisms allow non-functional requirements to be maintained in the *component schema* repository.

Subsection Summary

The deficiencies of *component schemas* discussed in this section give rise to the more general question of what has to be expressed in given problem analysis techniques. Modelling in most general terms aims for the reduction of analysis complexity in abstracting from irrelevant aspects and concentrating on important ones. So there is always a trade-off between regarded and disregarded aspects of the problem at hand. Requirements engineering aims to describe what a required product shall do, instead of detailing how it can be accomplished. Going back to *component schemas*, they provide a suitable description of problem structure and behaviour. Therefore gaining insight into complex applications is the main objective, while capturing all possible requirements falls short. But we believe that problem analysis using *component schemas* is effective and that identifying suitable concepts to gather requirements not covered in UML is a suitable approach to improve the quality of the resulting SRS. The suggestions made in this section enable persistent storage of non-functional requirements in the *component schema* repository. Besides the improved coherence of *component schemas* elaborated by autonomous analyst teams, this work also contributes to improved management of non-functional requirements which facilitates the creation of meaningful and consistent software requirements specifications.

10.3.2 Generating Software Requirements Specifications from Component Schemas

While the previous subsection identified deficiencies of *component schemas* from a software requirements specification (SRS) perspective, we now want to describe how *component schemas* can be used for the generation of SRS. Increasing the overall project efficiency is the main motivation in this context, since rewriting requirements always demands additional effort and provides another source of errors as well as misunderstandings. The guideline for the following discussion is the IEEE 830 standard for recommended practice for software requirements [IEEE830] in its 1993 revision. In the remainder of this section, software requirements specifications built according to the IEEE standard are called IEEE-SRS. The organisation of an SRS and in particular its Chapter 3, called *Specific Requirements* in the IEEE jargon has to be discussed. Generally the standard proposes seven different SRS organisations models for this chapter. Since objects provide the sole paradigm in *component schemas*, organising the SRS by *objects* is the only logical conclusion. So the union of all classes representing interfaces to other systems has to be gathered in Section 3.1.3 called 'Software interfaces', while all other classes forming the problem statement at hand will be listed in section '3.2 Classes'. Figure 109 depicts the relevant outline following IEEE-830 standard.

Figure 109 The Proposed IEEE Outline for Object-Oriented Software Requirements Specifications

Table of Contents

1 Introduction
- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, acronyms, and abbreviations
- 1.4 References
- 1.5 Overview

2 Overall description
- 2.1 Product perspective
- 2.2 Product functions
- 2.3 User characteristics
- 2.4 Constraints
- 2.5 Assumptions and dependencies

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communication interfaces
- 3.2 Classes
  - 3.2.1 Class 1
    - 3.2.1.1 Attributes
      - 3.2.1.1.1 Attribute 1
      - ...
      - 3.2.1.1.2 Attribute m
    - 3.2.1.2 Functions (Operations)
      - 3.2.1.2.1 Functional requirement 1
      - ...
      - 3.2.1.2.2 Functional requirement n
  - 3.2.2 Class 2
  - ...
  - 3.2.p Class p
- 3.3 Performance requirements
- 3.4 Design constraints
- 3.5 Software system attribute
- 3.6 Other requirements

Appendix

Non-functional requirements have been grouped in the sections 3.3 to section 3.6 in the IEEE-830 recommendation. Referring to our project, we would end up with 391 classes with an average of ten operations and five attributes per class [Eur98a, p. 4]. Use cases and UML sequence diagrams are not included in this standard. But even in the static structure, two important groupings within the original problem analysis model get lost in the IEEE SRS organisation, namely packages and initial partitions. Especially packages, as logical groupings for model elements, may provide meaningful groupings for its encapsulated model elements. The outline in section 3 does not

match the diagram hierarchy in the used CASE tool, reducing the comprehensibility of the specification. The other drawback is related to non-functional requirements, which have been grouped at the end of the section. Our extensions to the metamodel always attach the tagged values to the metaclass the requirement actually applies to. Consequently, non-functional requirements appear directly where they have been specified, i.e. in `packages`, `attributes`, `operations` and `class specifications`.

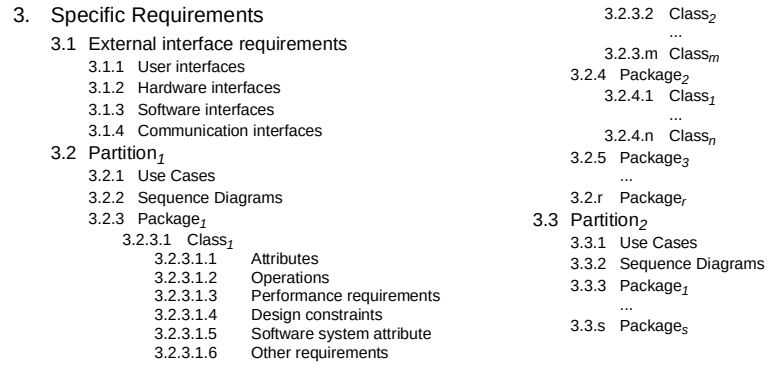
Alternative IEEE-830 Outlines

The 'flat' character of the IEEE-830 outline for objects, represents neither packages nor partitions. Going back to the standard, there is no direct equivalent to partitions mentioned. The only more or less corresponding outline groups the demanded functionality according to the perspective of given stakeholders, which are called *user class* in the standard. Although partitions in this work may incorporate several user classes, i.e. actors in use cases, this grouping still roughly corresponds to *user classes* in the sense of the IEEE-830 standard. For these cases, the standard also proposes so-called *multiple organisations*. So in order to increase the comprehensibility of an SRS with respect to the problem analysis models, we propose the use of partitions as the top-level grouping for an SRS derived from *component schemas*. The original grouping encountered in the CASE tool's diagrams may also be preserved through adding packages as a second-level grouping mechanism. Putting everything together we end up with the structural part of the specification, which roughly corresponds to Firesmith's proposed grouping for object-oriented requirements engineering [Fir93, p. 371].

So far we have completely left out the diagrammatic representation of our problem analysis models, which provides extremely useful information due to its concise and compact form. The IEEE specification only supports textual representation, leaving all diagrammatic information to the appendix. According to our experience, diagrams largely facilitate familiarization with complex application domains. Therefore we added all diagrammatic information to our reports [Wor96, Eur98a]. The potential *trivialisation* [AEQ98] has been avoided by means of textual descriptions for each diagram. A similar strategy has been applied by John Arlow and his colleagues in a large-scale project for British Airways [AEQ98]. The entirely structural perspective of the IEEE-830 standard leaves us with use cases and sequence diagrams. In this work we propose to integrate them in the SRS, in order to provide various representations of the same information. Especially the broad spectrum of readers, from top management to system designers, demands a representation of requirements that suits the different capabilities and interests of the involved stakeholders. Mixing diagrammatic with textual information always guarantees that the important requirements and business issues can be understood without *literacy* in given modelling techniques. Thus managers do not have to study the modelling guidelines [WF97] in order to understand the requirements stated in the specification. The central grouping for SRS derived from *component schemas* are the involved partitions. Then the internal organisation of each partition – its package structure – provides outline for the SRS to be generated. Finally there is an option on whether all involved class specifications stemming from given partitions are grouped together at the end of the subsection or

are ordered with the diagrams the class specification appears in. Although the second alternative seems to be the most comprehensible one, this is still a matter of the existing tool support. Rational's report writing software SoDa [SoDa98] is not able to present only those attributes and operations in the textual description depicted in the actual UML static structure diagram. In order to preserve redundant class specifications, all classes are gathered by the end of each document. Therefore the EUROCONTROL reports [Eur98a - Eur98d] start each subsection by documenting a partition with the diagrams and finishes the chapter with all involved classes.

Figure 110 The Proposed Outline for Software Requirements Specifications based on Component Schemas



10.4 Summary

This chapter primarily evaluates *component schemas* from a software requirements specification (SRS) perspective. Analysing the kind of requirements already represented in *component schemas* and identifying possibilities to incorporate those not yet covered is the primary concern of this chapter. These contributions of this work further increase improvements with respect to the coordination among involved analyst teams concurrently specifying their partitions. First of all, *component schema relationships* and their automated detection overcome the inherent risk of misunderstanding and overriding requirements stated in related partitons. Second, the definition of *component schemas* also has the potential to increase coherence and minimality of the overall problem analysis model. Minimality and coherence essentially influence the quality of all derived products downstream in the software engineering process. So improving the coordination between concurrently working analyst teams provides an important improvement in quality of the problem analysis models and consequently strengthens the thorough understanding of industry-scale applications. This major achievement of this work has been contrasted to the wider frame of software requirements specifications. Directly incorporating *component schemas* into an SRS guarantees efficient and cost-effective use of information already gathered in problem analysis. The discussed adornments with respect to non-functional requirements further narrow the gap of missing information. The use of the *component*

schema repositories further reduces redundant information sources. As a result, making direct use of the entire information stored in a *component schema* reduces involved information sources and increases the input for writing quality software requirements specifications.

Application Domain Achievements

With respect to the air traffic management domain, extensive data driven analysis, as described in this work, improves the planned system integration efforts in the near future. While today's flight data processing systems have to be updated via manual controller input, communicating via telephone lines, future systems should exchange data on an automated basis. In this specific sense, *component schemas* provide suitable means to tackle the desired ubiquity of information for future air traffic management systems. Especially the existing heterogeneity in the European air traffic management infrastructure calls for reference models, in order to allow the definition of unified component interfaces, instead of doing the actual integration on a one-by-one basis.

What has been left out so far?

While *component schemas* facilitate thorough understanding of complex application domains, there are still significant activities left that remain completely outside the scope of problem analysis models. The solution-neutral subset of UML increased the orthogonality of *constructs* of *component schemas* and facilitated the familiarization of complex application domains for larger audiences, from top management to programmers. On this basis an effective participation of all stakeholders can be achieved. But two important aspects remain unreflected, since they belong entirely to they realm of business decisions. First of all, the validation of requirements, which has been only partially tackled so far. Improved coherence and minimality of involved *component schemas* largely reduces the risk of obvious contradictions on a model level. However, there still remain hidden issues, subject entirely to the expertise of involved domain experts and the senior analysts. Inaccurate specifications of requirements may only cover 80% of the routine business cases. The remaining 20% still might lead to erroneous calculations and therefore may threaten system security. Although *component schemas* provide a tool for understanding complex requirements, they do not provide the frequently quoted '*silver bullet*' for concurrent requirements engineering. Missing risk management aspects and requirements that have not been specified by the analysts cannot be provided by *component schemas*. According to Robert Glass, the dreadful history of large-scale software failures has been provoked by insufficient risk assessment in 38% of the cases [Gla97]. Further 55% of the failed projects did no risk management at all [Gla97]! Risk management can be facilitated on the basis of quality requirements specifications. The other point left out in this work is the business aspect of requirements specifications. Software requirements specification tend to present the ideal world to its readers, but often the specified system is far too expensive for the ordering company. So *essential*, *mandatory* and *optional requirements* have to be indicated. *Component schemas* are able to maintain this kind of information in the repositories, but finally this kind of decision is entirely the responsibility of senior management.

11 Conclusion and Future Work

11.1 Conclusion

This work described the ingredients for concurrent requirements engineering in industry-scale applications using the Unified Modelling Language [OMG97a]. Reflecting our practical experience gained in the air traffic management domain [Wor96, WCK97, WCW97] we introduced a methodology including organisational as well as technical aspects for running concurrent requirements engineering projects. In this context, we mainly concentrated on improving the demanding conceptualisation of industry-scale problem statements within the larger frame of requirements engineering. On the technical level, the definition of *component schemas* proposes a solution-neutral subset of the popular Unified Modelling Language through focusing on relevant views for requirements documentation. All *constructs* within the selected UML views have further been clarified, in order to increase their orthogonality and comprehensibility of *component schemas* to experts in the application domain. Leaving the metamodel unchanged ensures CASE tool compatibility on the one hand and preserves valuable design and implementation constructs that may come into play at later stages within the software design cycle.

Component Schema Relationships

Component schemas may be used for concurrent elaboration of initial partitions forming a larger problem statement. The introduction of *component schema relationships* tackles the inherent problem of consistency and redundancy in concurrently elaborated *component schemas*. Unlike correspondence assertions, known from the schema integration community, both linguistic as well as structural aspects of the modelling technique have been incorporated in our definition of *component schema relationships*. The essentially linguistic character of *component schemas* demands an enhanced definition of *correspondence* compared to the schema integration literature. *Correspondence assumptions* indicate potential conceptual correspondences between class specifications stemming from different *component schemas.* Using the object-oriented modelling technique UML for requirements documentation also leads to a number of different relationships currently not covered in the schema integration community. In this context we stress the significance of *inter-construct-kind* and *inter-construct-level relationships*, indicating a potential *correspondence assumptions* between *artefacts* on different levels of abstraction. A potential relationship between a package and a class specification stemming from different *component schemas* indicates a potential mismatch in the initial balancing of the applied level of abstraction in individual *component schemas*.

Coherence of Component Schemas

From a broader perspective *component schema relationships* have the potential to improve the coherence of concurrently elaborated *component schemas*. Analyst teams may develop their initial partitions in isolation and unintentional overriding of requirements from analyst working on other initial partitions is excluded due to the use of distributed repositories. *Component schema relationships* indicate correspondence and conflicts between *artefacts* stemming from different *component schemas*.

These conflicts potentially threaten the consistency of the overall problem analysis model and have to be indicated to the affected analyst teams. Indicating potential conflicts to the affected analyst teams and resolving them in a cooperative manner overcomes the risk of misunderstanding complex requirements stated in other partitions. Unintentional overriding of requirements can also be avoided. *Component schema relationships* provide project management with means to assess the coherence and minimality of the overall problem statement. At the same time the project progress can be controlled.

Improving Project Management

Component schema relationships are entirely realised in terms of the current UML metamodel version and can therefore be realised on top of off-the-shelf CASE-tools. Defining *component schema relationships* in terms of the metamodel and incorporating status information also provides a driver for the necessary coordination of participating analyst teams working on distributed repositories. Going through the correspondence management cycle in a master-multiple organisation at least guarantees coherence of the overall problem analysis model at given moments in time and therefore dramatically reduces the risk of overriding requirements for the integration of *component schemas* by the end of requirements documentation. Potential conflicts can be indicated to the affected analyst teams at any time. At the same time, integration effort can be avoided in earlier phases of problem analysis, where the conceptualisation of the problem statement tends to evolve rapidly. *Component schema relationships* help to clarify the conceptual relationship between artefacts stemming from different *component schemas* and therefore help to overcome the rather self-contained perspective of concurrently elaborated *component schemas*. Therefore *component schema relationships* have the potential to improve the coherence of the overall problem analysis model without restricting the autonomy of individual analyst teams.

From Component Schemas to Software Requirements Specifications

The improved coherence of *component schemas* directly affects the product quality downstream in the requirements engineering cycle. A validation of contents of the *component schemas* finally clarifies which information is still required for quality software requirements specifications (SRS). Here we are especially interesting in overcoming deficiencies of *component schemas* with respect to the representation of non-functional requirements, that cannot directly be modelled in UML. Using *component schemas* for the creation of software requirements specifications improves project efficiency and also improves the participation of business users and domain experts. The initial partitions form the outline for the SRS and contents of *component schemas* represent elicited structural and behavioural requirements. The involved stakeholders do not loose track of their contributions. Providing multiple access to the same information improves communication among all stakeholders. Analysts with an IT background may use the models to gain a quick overview of complex requirements. Complementary plain-text descriptions explain the depicted problem partition to business users and managers. So the diverse representations of complex requirements have the potential to suit a broad audience.

Improving Requirements Engineering

The management cycle for *component schema relationships* increases the maturity of the requirements engineering processes and therefore avoids flaws and omissions in the derived software requirements specification. Discussing a suitable representation for non-functional requirements immediately increases overall project efficiency, since *component schemas* may be used directly for the generation of software

requirements specifications. Proposing a suitable outline based on the IEEE-830 recommendations for software requirements specifications also increases the traceability between the problem analysis model and textual adornments necessary in a complete SRS. So the coherence of *component schemas* has been increased in order to improve thorough understanding of industry-scale problem statement, and make optimal use of the gathered information for the derived products, such as software requirements specifications. Using *component schemas* for the communication between involved stakeholders also improves the maturity of requirements processes and therefore has the potential to improve the quality of all products downstream in the software development cycle.

11.2 Future Research

Generation of Linguistic Support

This work identified a number of interesting research problems that have not been tackled so far. First of all an improved definition of metrics incorporating linguistic as well as structural aspects of class definitions may lead to an automated detection of correspondence assumptions. A rigorous formal model, combined with these mentioned metrics also leads to an enhanced definition of *conflict* and *coherence*. Currently the *coherence* of *component schemas* can only be assed on the basis of attributes of *component schema relationships* proposed by analysts, while advanced metrics combining structural aspects such as *attribute domains* with linguistic aspects as retrieved from domain ontologies, may directly assess this characteristic on the level of elements stemming fromvarious *component schemas*.

The essentially linguistic character of *component schemas* leads to a method-specific language reconstruction in the sense of Ortner and Schienmann [OS96]. This leads to the follow-up question of how problem analysis models can be used efficiently for the generation of thesauri and ontologies. Linguistic support for novel problem domains is quite unlikely. That means structuring the domain vocabulary actually takes place in problem analysis models. Directly making use of the expert language found in *component schemas* will especially improve system evolution in the presence of constantly emerging new demands that are to be implemented in operational systems.

Impact of Design Considerations on Problem Analysis Models

Another interesting question is related to the impact of design considerations on initial problem analysis models, as stated in software requirements specifications. While analysis models in requirements engineering represent key concepts relevant in the business at hand, design issues significantly modify these models. This leads to different problems. First of all, domain experts loose track of ‘their’ initial problem analysis model and therefore are not able to assess the impact of emerging requirements on operational systems. They simply do not recognize their models any more and therefore the necessary cooperation between domain experts, analysts and designers is threatened. The other important point is the relationship between problem analysis models describing key concepts of an idealised reality and design models reflecting the necessities of the enabling technologies from a technical point of view. Enabling technologies lead to dramatic alterations of the problem analysis models and therefore create a new reality that does not necessarily correspond to the perspective of involved stakeholders. The other important aspect of design issues is

traceability, i.e. alterations of the problem analysis model are originated by given design decisions. This question leads to the broader question of architectural abstractions in software design. Stefan Tai [Tai99], for instance, discusses architectural abstractions for systems based on object request brokers, such as OMG's CORBA [OMG98].

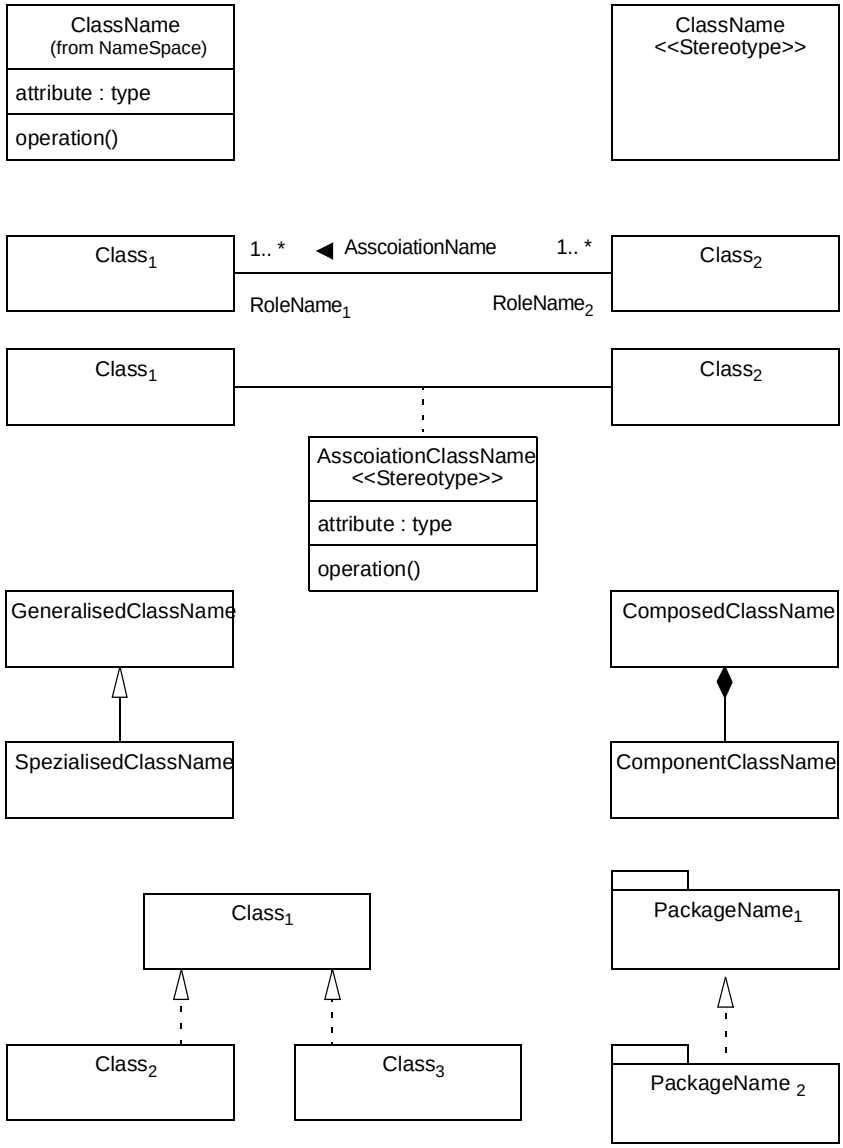
Requirements in Continuous Software Engineering

Stable and manageable requirements are the central basis for the successful development of industry-scale systems. But an existing infrastructure like the current air traffic management systems in Europe constantly faces new demands. In this context, systems have to be enhanced, they must be easily amalgamated with peer systems, and entire subsystems have to be replaced. Requirements engineering and problem analysis models describing demand play an important role in this context. However, especially the integration of legacy systems through reverse as well as reengineering processes marks an important challenge to requirements engineering. How can we map conceptual entities as derived from problem analysis models to components that already incorporate design decisions and represent the implemented architecture of operational systems? Both aspects, i.e. design decisions and the implemented architecture usually remain completely undocumented. The nature of the specific relationship between business concepts incorporated in a legacy system and future demand as expressed in a newly created problem analysis model marks another interesting research topic. Continuous enhancement of the existing infrastructure demands clear separation of business concepts on the one hand and applied design decisions in the implementation on the other. These questions mark starting points for future research in the context of continuous software engineering, as proposed by the research group 'Computation and Information Structures' directed by Prof. Dr. Herbert Weber at Technische Universität Berlin.

Appendix A

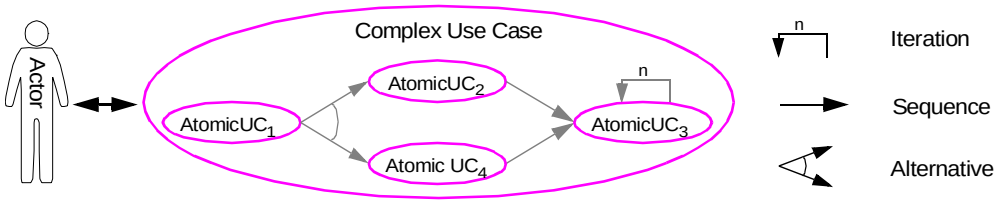
A.1 Notation for Static Structure in Component Schemas

Figure 111 Notation for Component Schema's Static Structure Diagrams



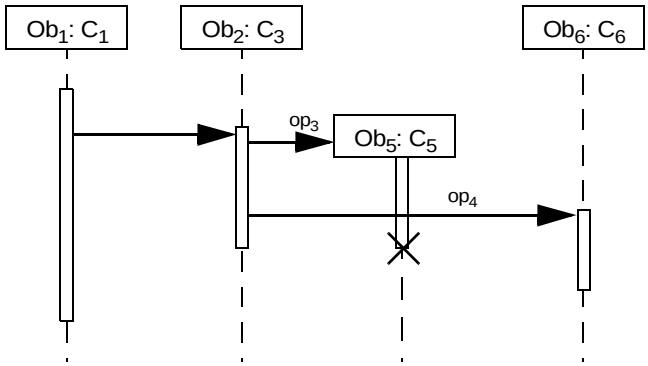
A.2 Notation for Use Cases in Component Schemas

Figure 112 Notation for Component Schema's Use Cases



A.3 Notation for Sequence Diagrams in Component Schemas

Figure 113 Notation for Component Schema's Sequence Diagrams



A.4 Predefined Stereotypes for Concurrent Requirements Engineering

The following list of stereotypes identifies model elements with a specific purpose within components schemas:

`«partition»`	This stereotype indicates packages representing initial partitions
`«HMI»`	This stereotype indicates class specifications encapsulating the future *humane machine interface*
`«interface»`	This stereotype indicates class specifications representing the interface to external systems

Appendix B

B.1 Sample Requirements Statements from EUROCONTROL Operational Requirements Documents

ATM Added Function

"Basic Functions provides MONA with current position data and actual performance characteristics of a flight in the form of a *State vector*. Basic Functions also provides a 4-D representation of the planned flight trajectory as the *System trajectory.* Additionally, Basic Functions also allows MONA access to more general *Flight data information*.

MONA can exercise control over trajectory recalculation via the *Recalculation request*.

Basic Functions supplies MONA with the necessary information relating to *Letters of agreement*, *Standard operating procedures*, *Airspace data* and *IAS limit data*.

MONA receives *Monitoring controls* from the controller through HMI. MONA provides the HMI with *Reminders* and *Non-conformance warnings* as appropriate for presentation to the controller.

Configuration data received from Basic Functions serves to inform MONA about the status of the overall system.
Arrival Management specifies the *Arrival management conformance monitoring criteria* that MONA is to take into account for flights subject to control by an automated Arrival Manager function.

Data for recording is sent to Basic Functions as MONA *recording data*." [Eur96a, p. 2-2]

FDPD

"*State Vector.* This is an elementary observation of an aircraft position ..." [Euro97, p. 3-3]

"FDPD-FDMD-23 Manual modifications of relevant SFPL data shall trigger trajectory prediction re-calculation" [Eur97, p. 9-5]

"FDPD-FDMD-31 The SFPL shall be updated upon receipt of ATFM messages" [Eur97, p. 9-6]

"FDPD-MESS-2 Semantic and syntactic checks shall be performed on all messages and inputs"

"flight level - A surface of constant atmospheric pressure which is related to a specific pressure datum, 1013.2 millibars, and is separated from other such surfaces by specific pressure intervals"

SDP

"Plots contain samples of the aircraft position at a certain moment in time ..." [Eur96b, p. 24]

B.2 Sample Requirements Statements from Use Cases

Advanced FDM OOA

"... the RECO proposes an alternate flight level using the *"CDN"* message ..." [Wor96, p. 41]

" ... the TRACO rejects the proposal using an *"RJC"* message ..." [Wor96, p. 53

" For departing aircraft the TRACO sends a *Preliminary Activate Message* ("*PAC*") to RECO, in order to allocate a new SSR Code for the departing aircraft" [Wor96, p. 62]

"In order to synchronise the participating systems TRACO sends a *REVision Message* ("*REV*") ..." [Wor96, p. 69]

Appendix C

C.1 Conflict Examples

Figure 114 Conflict Examples - Naming

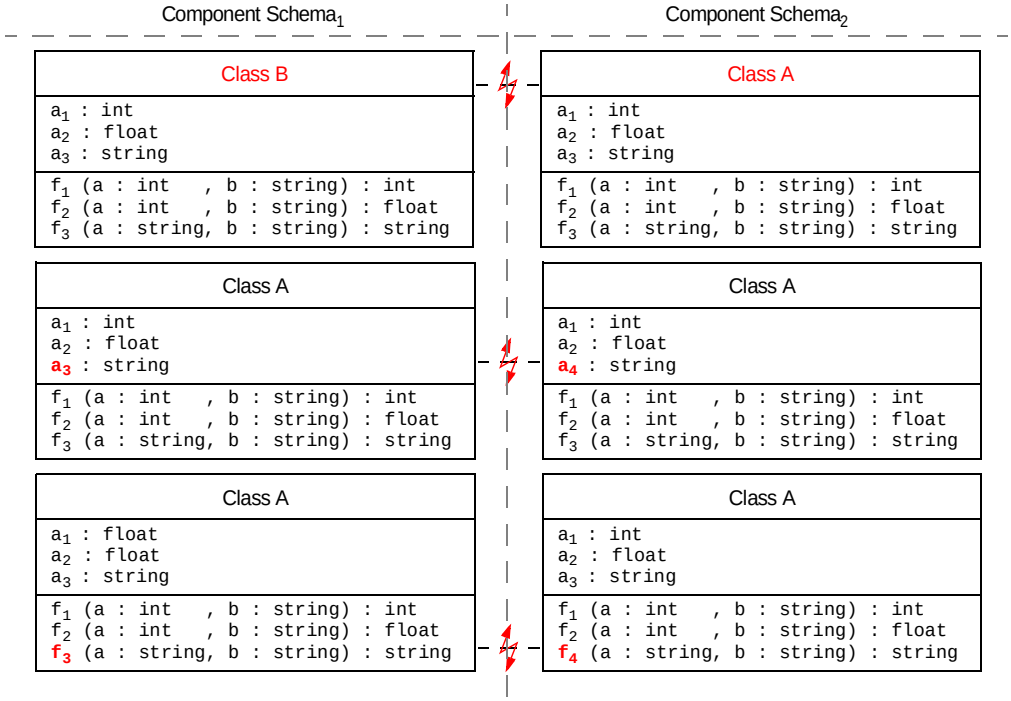
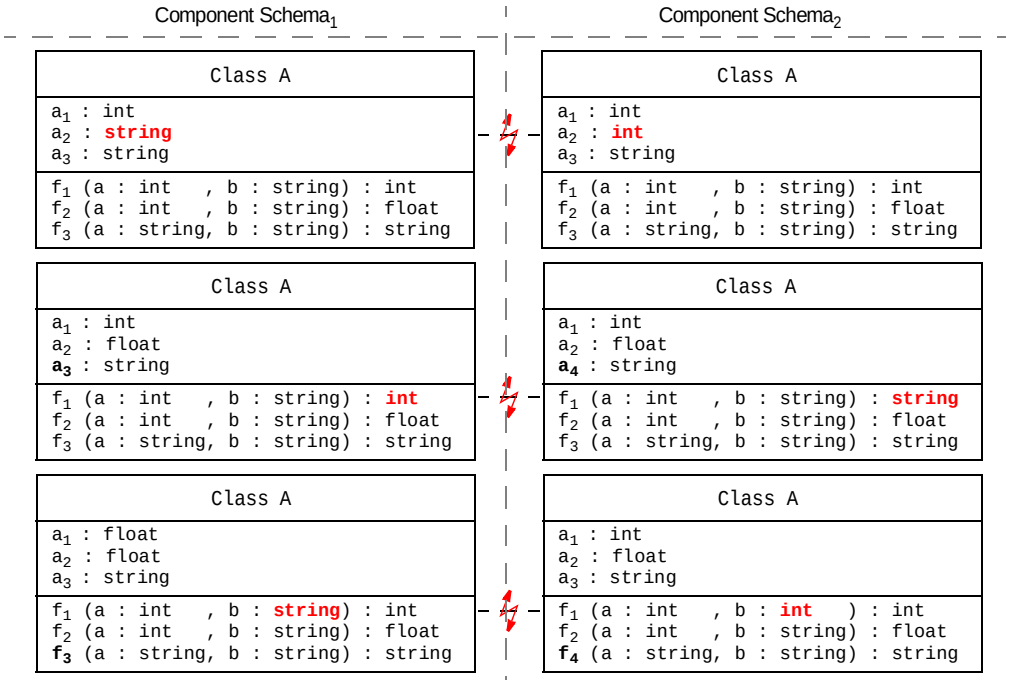


Figure 115 Construct Conflict Examples- Data Types



C.2 Surroundings Conflict Examples

Figure 116 Surroundings Conflicts - Cardinality

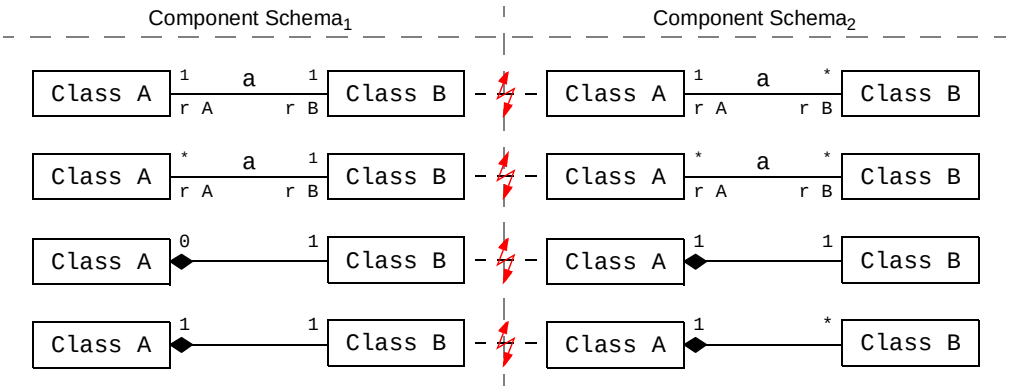


Figure 117 Surroundings Conflicts - Binary Associations

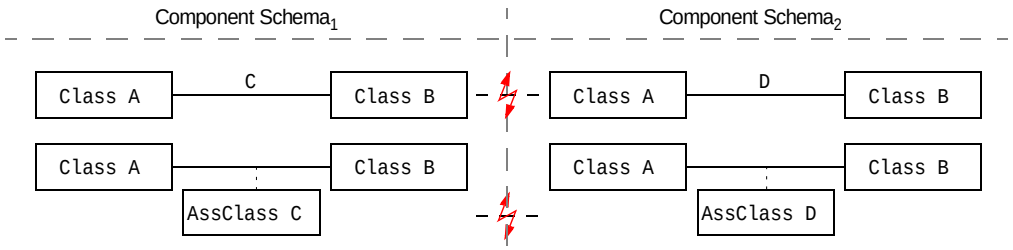


Figure 118 Surroundings Conflicts - Association-Type Conflict

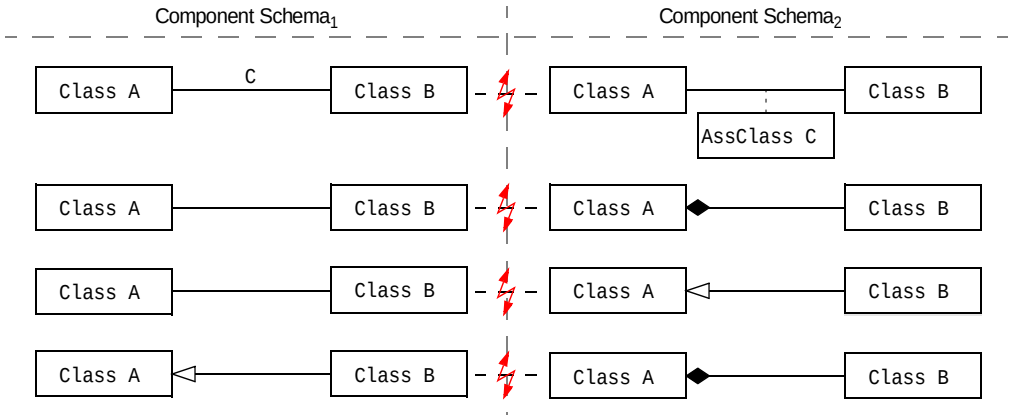
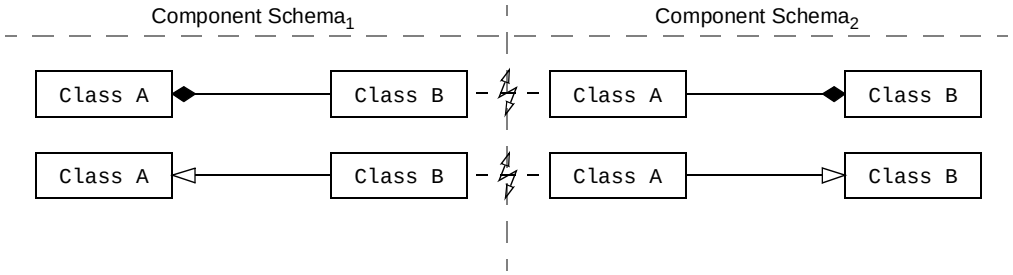
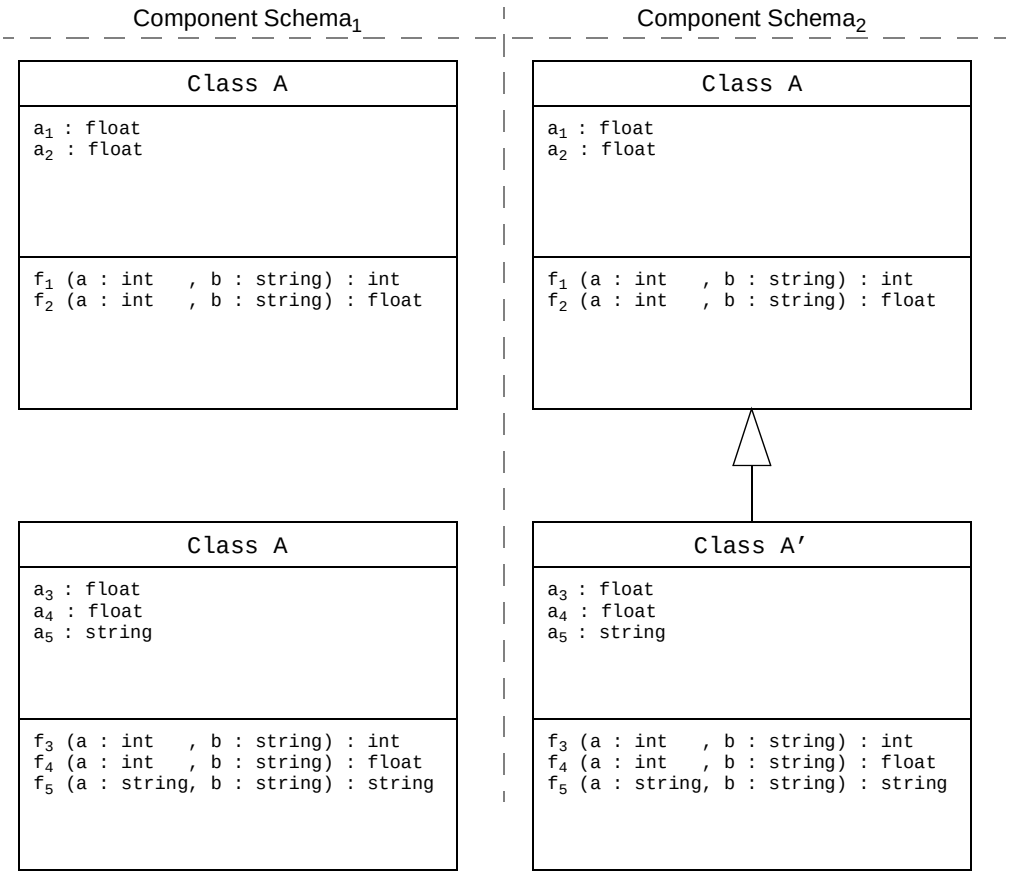


Figure 119 Surroundings Conflicts - Association-Direction Conflict



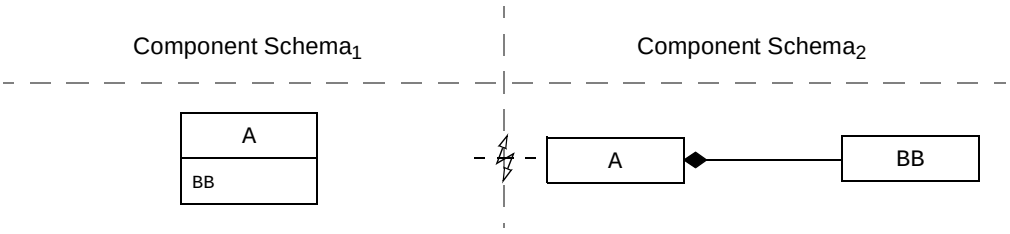
C.3 Examples of Semantic Conflicts

Figure 120 Examples of Internal Structural Similarity - Naming



C.4 Catalogue of Abstraction Conflicts

Figure 121 Examples of Abstraction Conflicts



Appendix D

D.1 List of Component Schema Correspondences and Conflicts

Table 11 List of Component Schema Correspondences and Conflicts with their corresponding Stereotypes

	Name	Correspondence	Conflict	Stereotype	Comment
intra-construct, intra-level	at-at	direct		«at-at correspondence»	
		synonym		«at-at correspondence»	
		terms of reference		«at-at correspondence»	
		scale		«at-at correspondence»	
		projection		«at-at correspondence»	
			attribute-domain	«at-at conflict»	
	op-op	direct		«op-op correspondence»	
		synonym		«op-op correspondence»	
		terms of reference		«op-op correspondence»	
		scale		«op-op correspondence»	
	cl-cl	direct		«cl-cl correspondence»	
		synonym		«cl-cl correspondence»	
		equipollence		«cl-cl correspondence»	
		specialisation		«cl-cl correspondence»	
	pc-pc	direct		«pc-pc correspondence»	
		synonym		«pc-pc correspondence»	
		equipollence		«pc-pc correspondence»	
inter-construct, intra-level	op-at	direct		«op-at correspondence»	
		synonym		«op-at correspondence»	
		terms of reference		«op-at correspondence»	
		scale		«op-at correspondence»	
		projection		«op-at correspondence»	
intra-construct, inter-level	cl-cl	direct		«cl-cl correspondence»	
		synonym		«cl-cl correspondence»	
		equipollence		«cl-cl correspondence»	
		specialisation		«cl-cl correspondence»	

	Name	Correspondence	Conflict	Stereotype	Comment
inter-construct, inter-level	at-cl	direct		«at-cl correspondence»	
		synonym		«at-cl correspondence»	
	op-cl	direct		«op-cl correspondence»	Incorporates name space
		synonym		«op-cl correspondence»	
	pc-cl	direct		«pc-cl correspondence»	Incorporates name space
		synonym		«pc-cl correspondence»	
named surroundings	as-as	direct		«as-as correspondence»	
		synonym		«as-as correspondence»	
	ac-as	direct		«as-ac correspondence»	
		synonym		«as-ac correspondence»	
surroundings type correspondence	co-as	composition-asscociation correspondence		«co-as correspondence»	
			concept-context	«co-as conflict»	
			holonym / meronym	«co-as conflict»	
	co-ac			«co-ac conflict»	composition properties are forbidden in component schemas
	co-ge	compositor-pattern correspondence		«co-ge correspondence»	
			composition-generalisation	«co-ge conflict»	
	ge-ac		generalisation-association class	«ge-ac conflict»	

D.2 Construct Matrix for Component Schemas' Static Structure Diagrams

Table 12 Intra-Viewpoint Construct Matrix for Component Schema's Static Structure Diagrams

	Class	Attribute	Operation	Association	Association Class	Composition	Generalization	Package	Dependency
Class	cl-cl p. 241								
Attribute	at-cl p. 243	at-at p. 250							
Operation	op-cl p. 245	op-at p. 251	op-op p. 255						
Association	as-cl p. 246	as-at p. 252	as-op p. 256	as-as p. 259					
Association Class	ac-cl p. 247	ac-at p. 252	ac-op p. 256	ac-as p. 261	ac-ac p. 266				
Composition	co-cl p. 248	co-at p. 253	co-op p. 257	co-as p. 262	co-ac p. 266	co-co p. 269			
Generalization	ge-cl p. 248	ge-at p. 253	ge-op p. 257	ge-as p. 263	ge-ac p. 267	ge-co p. 270	ge-ge p. 272		
Package	pc-cl p. 249	pc-at p. 254	pc-op p. 257	pc-as p. 264	pc-ac p. 267	pc-co p. 271	pc-ge p. 273	pc-pc p. 273	
Dependency	de-cl p. 249	de-at p. 255	de-op p. 258	de-as p. 265	de-ac p. 268	de-co p. 271	de-ge p. 273	de-pc p. 274	de-de p. 275

D.2.1 Class Correspondence

Class-Class Correspondence

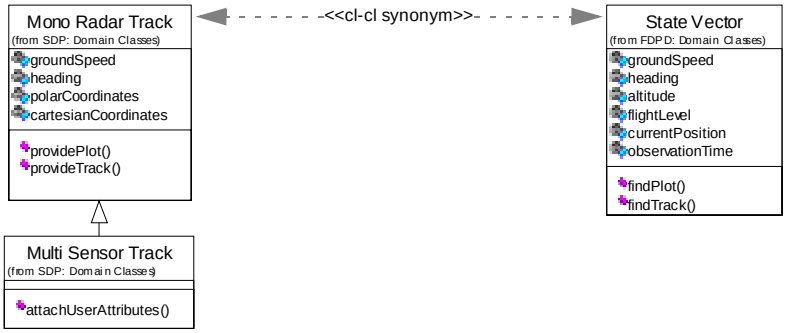
cl-cl Correspondence

Classification: intra-construct-kind correspondence
intra-construct-level correspondence

Categories: «direct», «synonym», «equipollence» & «upwardInheritance»

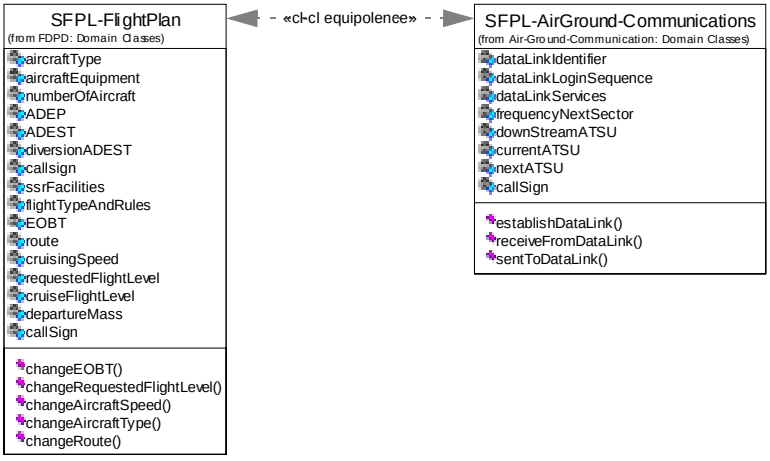
Within class-class correspondence a number of different forms can be distinguished. As artefacts, we first have to assess the similarity of the class names on the basis of a domain ontology, leading to a synonym class-class correspondence. In our first example, the terms »Mono Radar Track« and »State Vector« are synonyms of each other, according to the domain ontology. Still the assumed correspondence has to be reviewed by the involved domain leaders, since due to differing contexts and consequently distinct domain ontologies, the terms at hand may be at least polysems. This situation is detailed under association-composition correspondence (see Figure 153).

Figure 122 Class-Class Correspondence: Synonyms



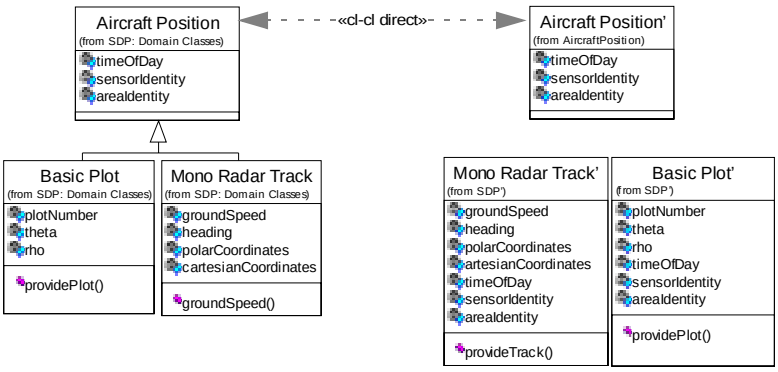
The next example introduces the equipollence class-class correspondence, which is directly related to the different perspectives incorporated in component schemas. Within this context, the same real-world concept leads to different artefacts, incorporating properties relevant in a given domain. Unlike the previous example, only few properties overlap, i.e. they are defined in the equipollent artefacts. Figure 124 depicts an example of equipollence.

Figure 123 Class-Class Correspondence: Equipollence



In this example, the two artefacts express the same concept, the »System Flight Plan«, but focusing on different aspects. While on the left hand side call sign, departure, destination aerodrome and the route in-between are represented, the right hand side concentrates on establishing and maintaining a digital datalink between controllers on the ground and the pilots. Only the property `callSign` is represented in both artefacts. This correspondence must not be confused with the concept-context conflict detailed in Figure 153, p. 262.

Figure 124 Class-Class Correspondence: Upward Inheritance



Unlike equipollence in the previous example, this one concentrates on the participation in generalisations hierarchies of corresponding classes. The domain ontology can automatically be extended whenever a term has been specialised in a component schema. So we can scan the other component schema for potential corresponding class names. However, this is exclusively related to direct correspondence and demands further minimisation of specialised class specification, i.e. the property `timeOfDay` for instance has to be removed.

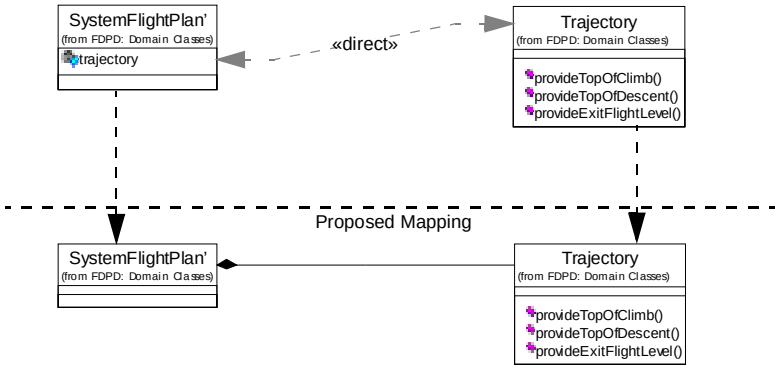
Attribute-Class Correspondence

at-cl Correspondence

Classification: inter-construct-kind correspondence
inter-construct-level correspondence
Categories: «direct» & «synonyms»

The recursive relationship between primary predicators denoting real-world things and secondary ones, which characterise the things expressed with a primary predicators lead to the following corresondence depicted in Figure 125:

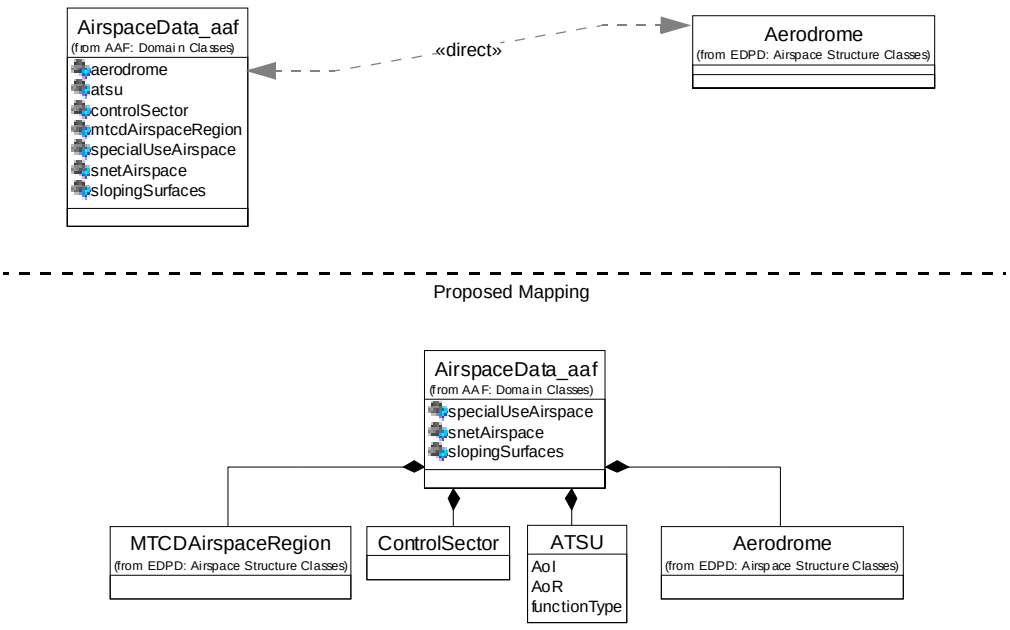
Figure 125 Attribute-Class Correspondence



Whenever two artefacts express the same concept, either through the use of the same or a synonymous terms, there is evidence for a attribute-class correspondence. Due to the restriction of attributes to simple data types in component schemas, complex properties always have to be modelled using two classes and a composition relationship. So when an assumed correspondence can be stated by the involved analysts, the property has to be removed and the artefacts have to be connected using a composition.

Our second example provides a slightly different situation. The component schema `AAF` contains a class called `AirspaceData_aaf` with a property `Aerodrome`. The term »data« in colloquial english is often used for an unspecified collection of information. In object-oriented problem analysis data is required from other partitions, without detailing the exact nature of this kind of information. In this light, this class specification can be regarded as a place holder for inter-partition requirements, that have to be settled. In this case the same actions may be applied, replacing the property by a composition. Consequently classes comprising the term data can be regarded as potential *to be resolved* requirements in the taxonomy of Oliver et al. [OKK97]. The bottom half of Figure 126 depicts a potential resolution of the assumed attribute-class correspondences:

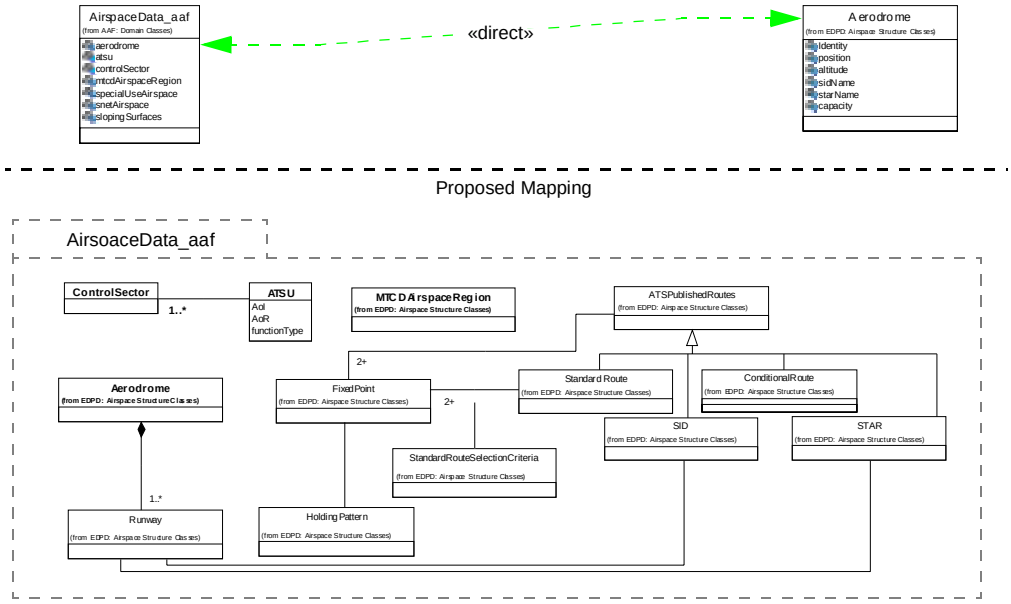
Figure 126 Resolving Attribute-Class Correspondence



Whenever the last property of class `AirspaceData_aaf` has been replaced by a composition, the entire class becomes obsolete, since the individual structural requirements modelled in this class have been detailed. Apart from classes that provide a

handle for complex structural problem aspects changing over time, such as the rather complex concept airspace configuration in air traffic management, the entire class `AirspaceData_aaf` can be replaced by a class category, in order to remove spurious class specifications. This situation is depicted in Figure 127.

Figure 127 Resolving Attribute-Class Correspondence Unspecified Data



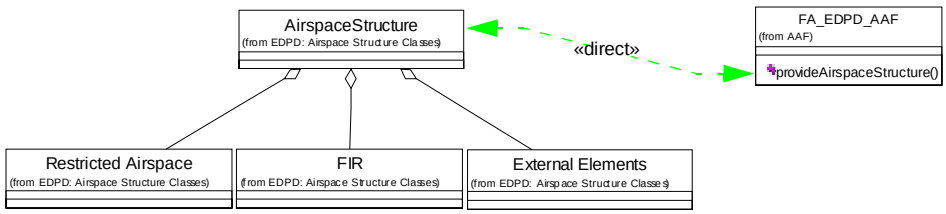
Operation-Class Correspondence

op-cl Correspondence

Classification: inter-construct-kind correspondence
inter-construct-level correspondence
Categories: «direct» & «synonyms»

Similar to our second example of attribute-class correspondence, course-grain inter-partition requirements may be collected in an individual class specification, representing a high-level interface to adjacent partitions. Unlike the previous example, the requirement is represented in terms of an operation. When component schemas have to be integrated an operation-class correspondence can be used to resolve coarse-grain inter-partition requirements. Unlike the attribute-class example, the operation must not be removed: instead the class name has to be inserted to the parameter section of the given operation. In order to improve traceability, a trace association may be inserted in order to connect the class with the operation. In our example the trace has to connect the classes `AirspaceStructure` and `FA_EDPD_AAF`.

Figure 128 Example of an Operation-Class Correspondence



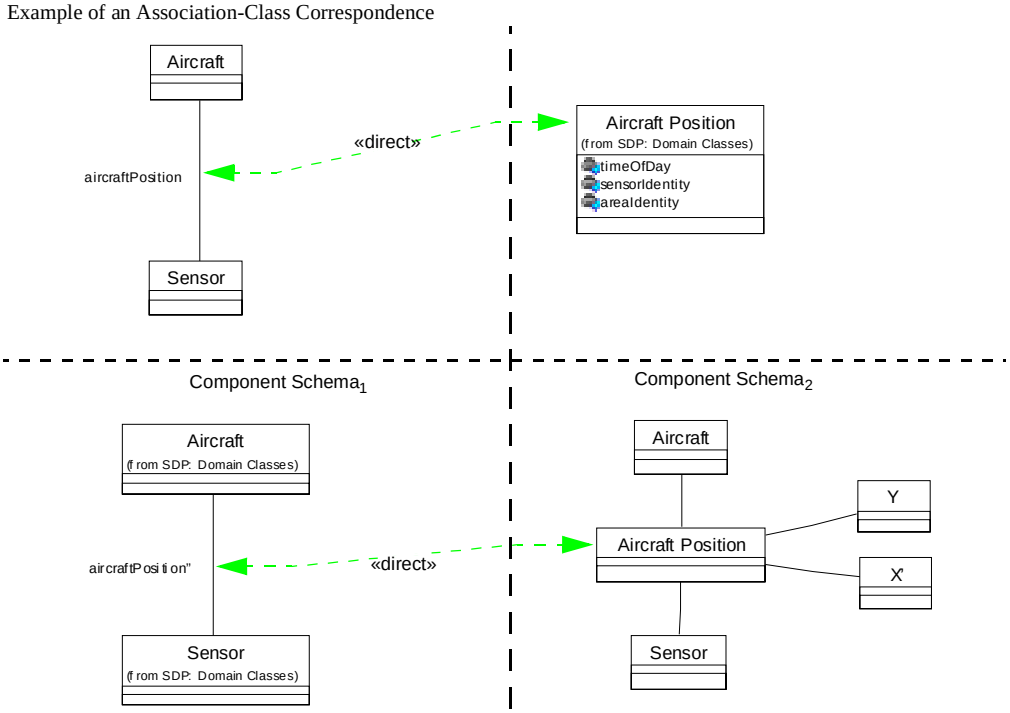
Association-Class Correspondence

as-cl Correspondence

Classification: inter-construct-kind correspondence
intra-construct-level correspondence
Categories: «direct» & «synonyms»

Association-class correspondences link relationships to properties, represented through association classes in component schemas. Such a correspondence only holds for named associations and entirely relies on the relationship name, as depicted in the upper half of Figure 129.

Figure 129 Example of an Association-Class Correspondence



However this correspondence gives rise to a number of questions, which lead to the bottom half of Figure 129. In the depicted models the class `Aircraft Position` has been modelled as a class in its own right, maintaining two independent relationships

to the classes Aircraft and Sensor. This situation basically illustrates modelling alternatives, with respect to the existence of instances, which can only be verified by domain experts. Choosing an association class makes the properties of class Aircraft Position to an inherent part of the relationship between a »sensor« and an »aircraft«. In the second representation instances of Aircraft Position may exist independently of a sensor and an aircraft. As a result an assumed correspondence among an association and a class specification involves an induced analysis of the class environment, namely the classes Sensor and Aircraft. The classes X′ and Y from component schema$_2$ do not influence the result, since there is no counterpart in component schema$_2$.

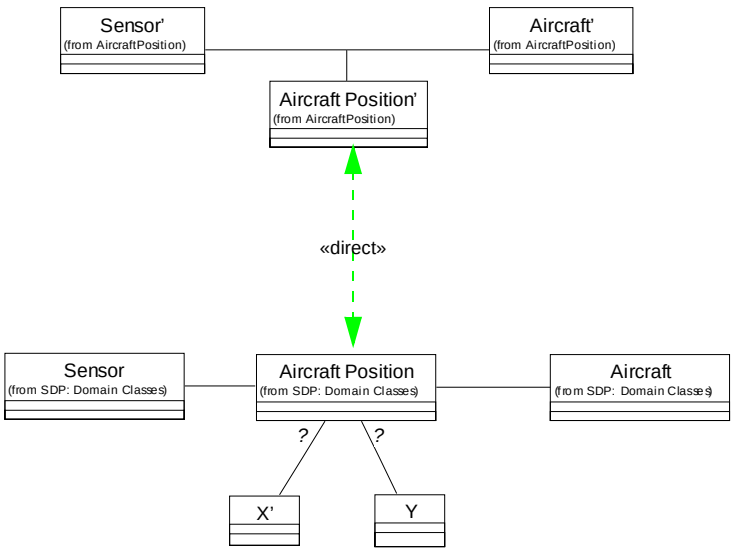
Association Class-Class Correspondence

ac-cl Correspondence

Classification: inter-construct-kind correspondence
intra-construct-level correspondence
Categories: «direct» & «synonyms»

Association class-class correspondences involve the same problems, as association-class correspondences, although in this case from the construct perspective the character of the association class has been stated. The assumed correspondence can then only be clarified through a comparison of the corresponding class' environment, namely the classes Sensor, Aircraft, X′ and Y.

Figure 130 Example of an Association Class-Class Correspondence



In the example of association class-class correspondence a concept similarity holds for the first two classes and no there is no further evidence for a correspondence of the latter ones. So although from a concept perspective we can state a correspond-

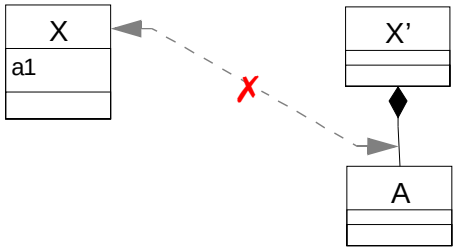
ence, from a modelling perspective we find a conflict, since the applied constructs differ and consequently apply a different set of integrity constraints. In the upper part, `Aircraft Position'` depends on the relationships, i.e. on the constituent classes, while in the bottom half the analysts have chosen a class in its own right.

Composition-Class Correspondence

co-cl Correspondence

Due to the anonymous character of compositions in component schemas, correspondences between classes and compositions can be excluded. This correspondence must not be confused with an attribute-class correspondence, as introduced in Figure 125.

Figure 131 Example of a Composition-Class Correspondence

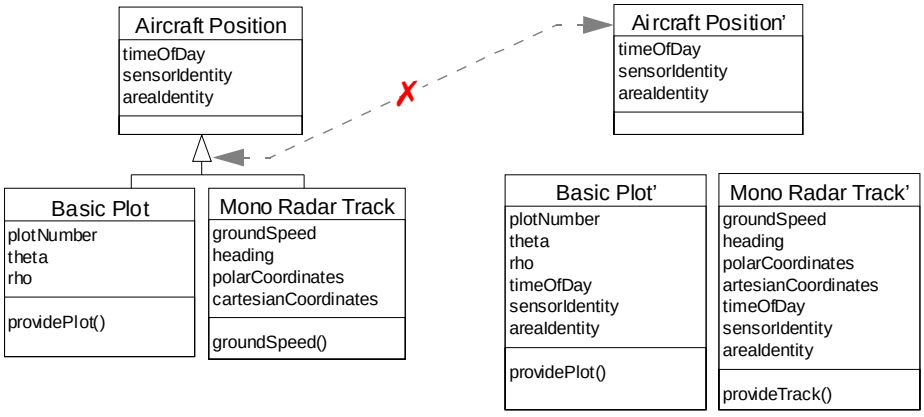


Generalisation-Class Correspondence

ge-cl Correspondence

Anonymous link artefacts cannot be correspond to artefacts, due to the relational character of UML relationships. However, this situation must not be confused with upward inheritance, as depicted in Figure 132 and has to be considered as a specialised form of class-class correspondence, which can be found in Figure 124.

Figure 132 Example of a Generalisation-Class for Correspondence



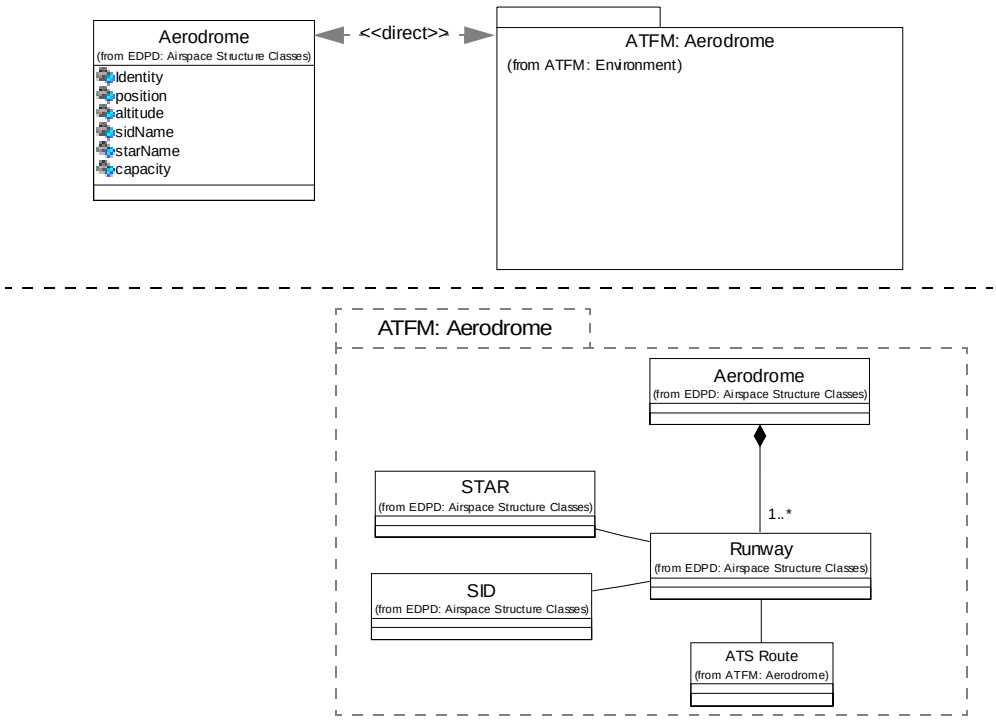
Package-Class Correspondence

pc-cl Correspondence

Classification:inter-construct-kind correspondence
inter-construct-level correspondence

Package-class correspondences hold, whenever their names are identical or are synonyms. A package then shall contain meronyms to the term or shall contain a class specification carrying the same name – `Aerodrome` in our example – and a set of meronyms detailing the concept referenced in the package name.

Figure 133 Example of a Package-Class Correspondence



Dependency-Class Correspondence

de-cl Correspondence

Since dependencies in component schemas are restricted to packages, a detailed analysis of this correspondence can be discarded and in order to avoid potential confusion, we do not provide an example. Moreover, anonymous link artefacts cannot be related to artefacts.

D.2.2 Attribute Correspondence

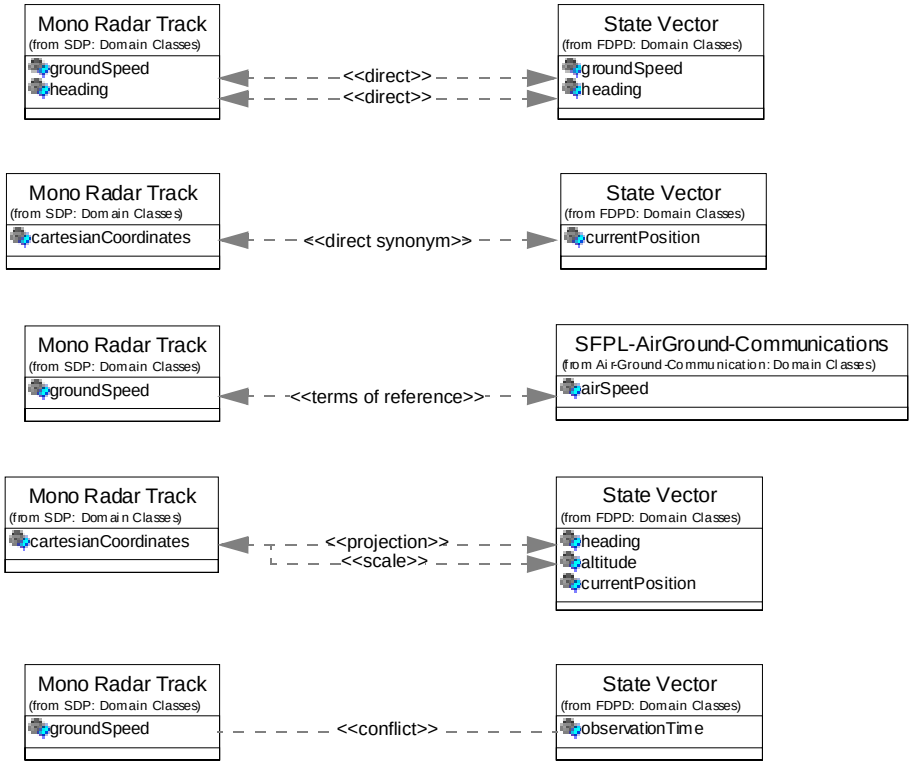
Attribute-Attribute Correspondence

at-at Correspondence

Classification: intra-construct-kind correspondence
intra-construct-level correspondence
Categories: «direct», «synonym», «terms of reference», «scale» & «projection»

Figure 134 depicts four different forms of attribute correspondences, each marked with a different stereotype. Attributes from the linguistic perspective are secondary predicatores characterising primary ones. Attribute correspondence presupposes that the artefacts use the same UML construct, namely instances of metaclass `Attribute`.

Figure 134 Different Forms of Attribute-Attribute Correspondence



In the uppermost example we find a direct correspondence, i.e. the same terms have been used to detail the concepts »State Vector« and »Mono Radar Track«. In the second example we find synonymous terms, since the term »currentPosition« incorporates the same scale and terms of reference with respect to »cartesianCoordinates«.

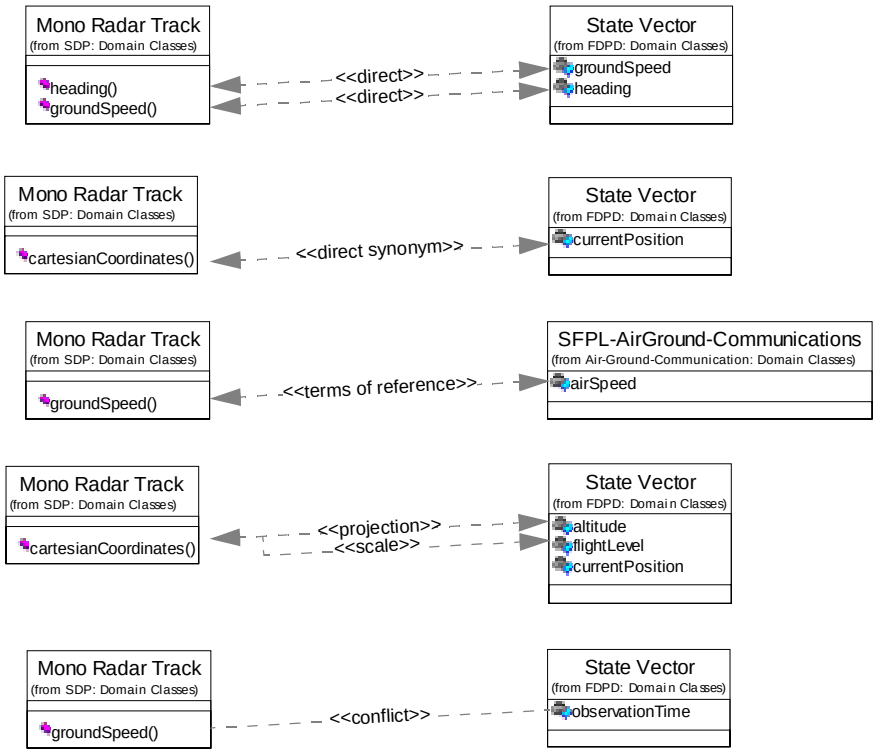
The third example introduces different terms of reference within synonymous terms, since »airSpeed« may vary from »groundSpeed« due to strong head winds. The terms of reference in this example denote where the speed has been measured. Finally, the »cartesianCoordinates«, »altitude« and »flight level« introduce a new notion. The »altitude« is a projection of the term cartesian coordinates. The term »flight level«, however, is a superimposition of *scale* and *terms of reference*, since flight levels are measured relative to the atmospheric pressure, depending on the local weather conditions. With respect to cartesian coordinates, »flight level« is a projected and scaled term, since the altitude is scaled to the local atmospheric pressure.

Operation-Attribute Correspondence

op-at Correspondence

Classification: inter-construct-kind correspondence
intra-construct-level correspondence
Categories: «direct», «synonym», «terms of reference», «scale», «projection» & «conflict»

Figure 135 Different Forms of Operation-Attribute Correspondence



Unlike operation-operation correspondence, the relationship between attributes and operations is more complicated. Following the naming conventions in Section 6.1.1.2 we assume that individual artefacts do not comprise so called *get and set operations*.

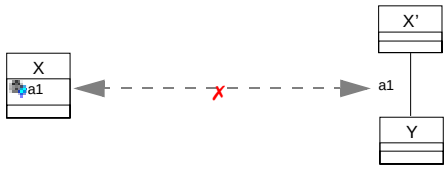
These operations enable access to the encapsulated internal structure of class specifications and usually carry the same name as the respective attribute. Therefore all operations, that exceed simply returning the values of the encapsulated attribute, are deemed to provide services. In concurrent problem analysis, due to equipollence, attributes in a class specification may be synonymous with an operation in another component schema. Therefore operation-attribute correspondence is comparable to attribute-attribute correspondence. Two constructs correspond when they have a similar linguistic meaning. Within the linguistic meaning we may find synonyms in the simplest case or variations such as projections, differing scale or differing terms of reference. In summary, operation-attribute correspondence stress the mathematical character of attributes as one-place predicates.

Association-Attribute Correspondence

as-at Correspondence

Due to the tuple semantics of UML relationships, association-attribute correspondence can be discarded. However, this correspondence must not be confused with the relationship concepts in languages with referential semantics such as CLASSIC [BBM+89] or programming languages such as C++.

Figure 136 Example of an Association-Attribute Correspondence

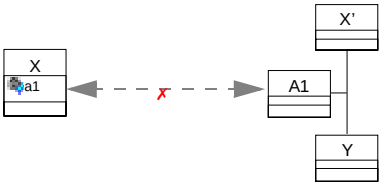


Association Class-Attribute Correspondence

ac-at Correspondence

Association class-attribute correspondence can be discarded due to the tuple semantics in UML. However this situation must not be mixed up with attribute-class correspondence

Figure 137 Example of an Association Class-Attribute Correspondence

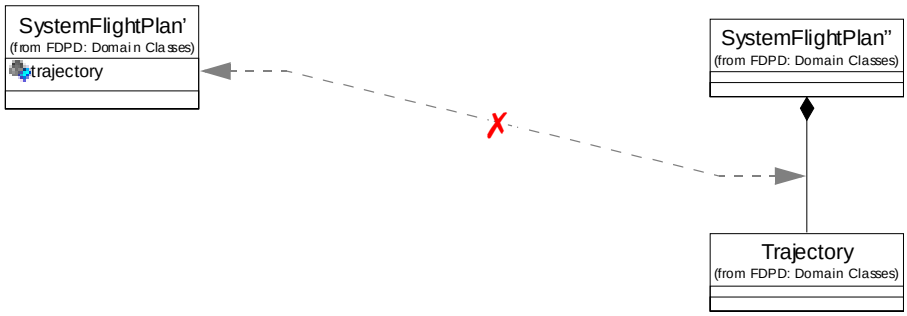


Composition-Attribute Correspondence

co-at Correspondence

Anonymous link artefacts cannot be compared to artefacts. As a result, there is no reasonable interpretation for a composition-attribute correspondence. However this correspondence is not to be confused with the attribute-class correspondence in Figure 127.

Figure 138 Illegal Composition-Attribute Correspondence

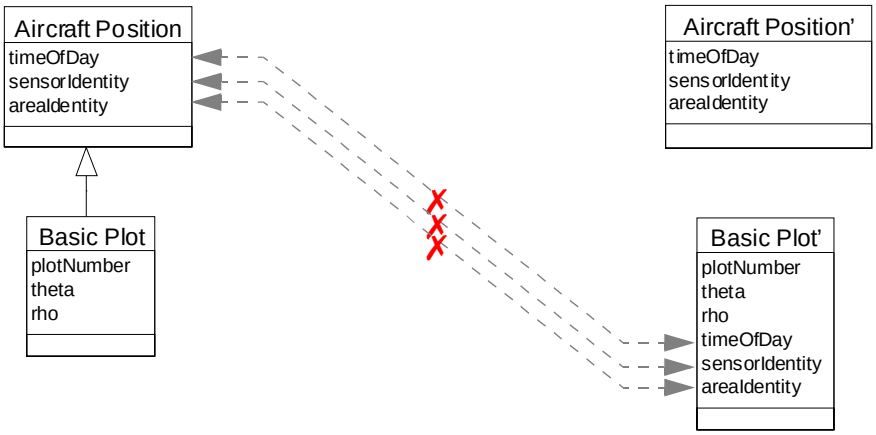


Generalisation-Attribute Correspondence

ge-at Correspondence

Like all anonymous link artefacts, generalisation relationships cannot be related to artefacts and are therefore discarded in this work. Moreover, this situation is not to be confused with upward inheritance.

Figure 139 Illegal Generalization-Attribute Correspondence

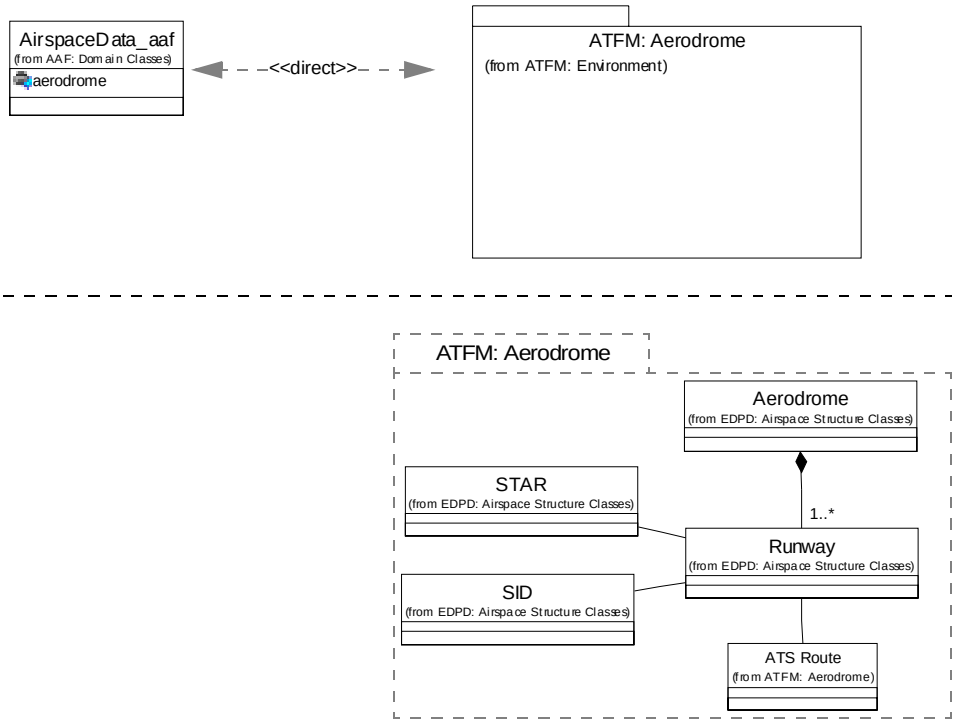


Package-Attribute Correspondence

pc-at Correspondence

Classification: inter-construct-kind correspondence
inter-construct-level correspondence

Figure 140 Example of Package-Attribute Correspondence



Package-attribute correspondences are part of the advanced correspondence relationships in component schemas. Following the naming conventions, the class specification `AirspaceData_AAF` comprises the property `aerodrome`. This term can also be found in the package name in the component schema `ATFM`. WordNet [WN98] provides the following definition for this term:

> "airport, airdrome, aerodrome -- (an airfield equipped with control tower and hangers as well as accommodations for passengers and cargo)"

The bottom half of Figure 140 depicts the contents of the package `ATFM: Aerodrome`, i.e. the classes describing the decomposition of the abstract concept airspace. A correspondence between an attribute and a package holds, when the terms within the different artefacts are synonymous with each other and the class specifications contained in the package are meronyms of the package name.

In this case, the assertion holds, since the package even contains a class carrying the same name (class `Aerodrome`) and depicts a set of classes further detailing the structure of an aerodrome. The class `Runway` is listed as an inherited meronym of class `Aerodrome` in WordNet [WN98]:

> “airport, airdrome, aerordrome [...]
> HAS PART: control tower [...]
> HAS PART: apron [...]
> HAS PART: runway [...]
> HAS PART: taxiway, taxi strip [...]

Dependency-Attribute Correspondence

de-at Correspondence

Besides their behavioural character, dependencies in component schemas have been restricted to packages. Therefore a dependency-attribute correspondence can be discarded.

D.2.3 Operation Correspondence

Operation-Operation Correspondences

op-op Correspondence

Classification: intra-construct-kind correspondence
intra-construct-level correspondence
Categories: «direct», «synonym», «terms of reference», «scale», «projection» & «conflict»

From a linguistic point of view, operation correspondences follow the same distinction as attributes. The central difference, however, is the behavioural character of these properties describing required system functionality.

Figure 141 Different Forms of Operation-Operation Correspondence

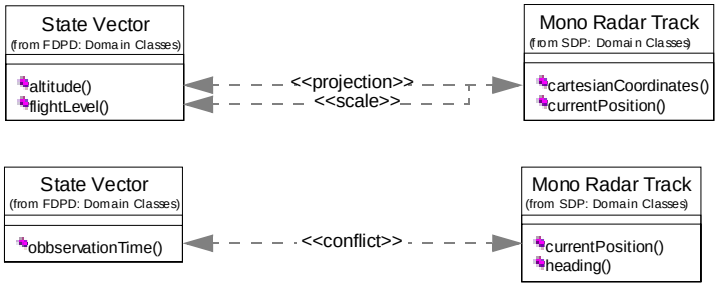
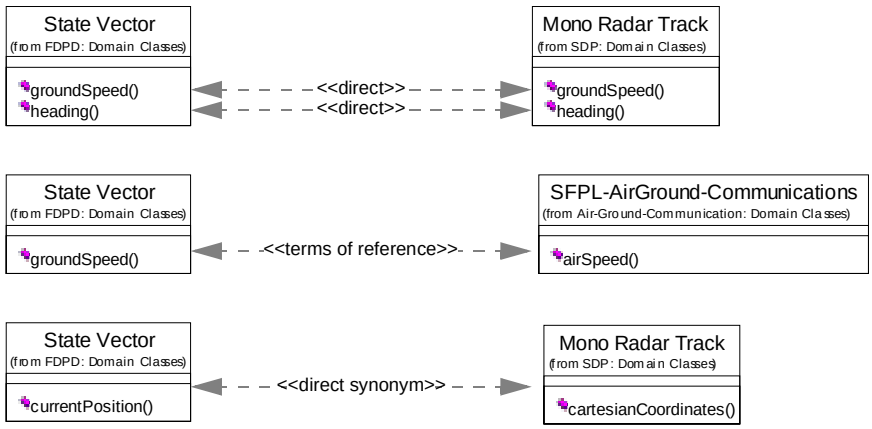


Figure 142 Different Forms of Operation-Operation Correspondence

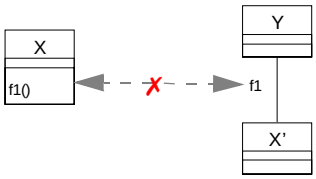


Association-Operation Correspondence

as-op Correspondence

A correspondence between an association and an operation does not have a reasonable interpretation in component schemas. On the one hand, the mixing of behavioural and structural properties is responsible for this observation. On the other hand, this situation is not to be confused with the operation-attribute correspondence, where *get and set operations* are compared to attributes.

Figure 143 Illegal Association-Operation Correspondence

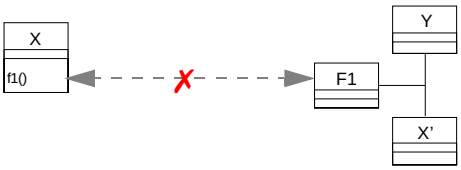


Association Class-Operation Correspondence

ac-op Correspondence

In practical terms, the same argument against an association-operation correspondence holds for this correspondence. We therefore discard this correspondence.

Figure 144 Example of Association Class-Operation Correspondence

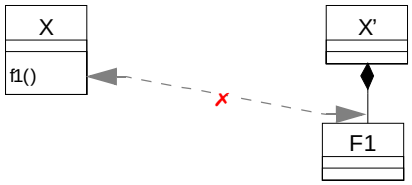


Composition-Operation Correspondence

co-op Correspondence

Besides the behavioural character of operations, anonymous link artefacts cannot correspond to artefacts. Therefore we discard this correspondence.

Figure 145 Example of a Composition-Operation Correspondence

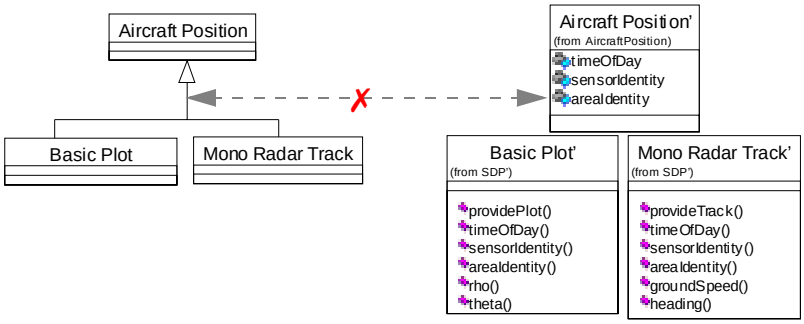


Generalisation-Operation Correspondence

ge-op Correspondence

Although similar properties, of both behavioural and structural nature, are the basis for the correspondence between generalizable classes where a common generalised class has to be built for a set of classes sharing the same properties, a correspondence between an anonymous link artefact and a artefact does not find a reasonable interpretation. However, this situation is not to be confused with the situation Figure 146, p. 257

Figure 146 Illegal Generalization-Operation Correspondence



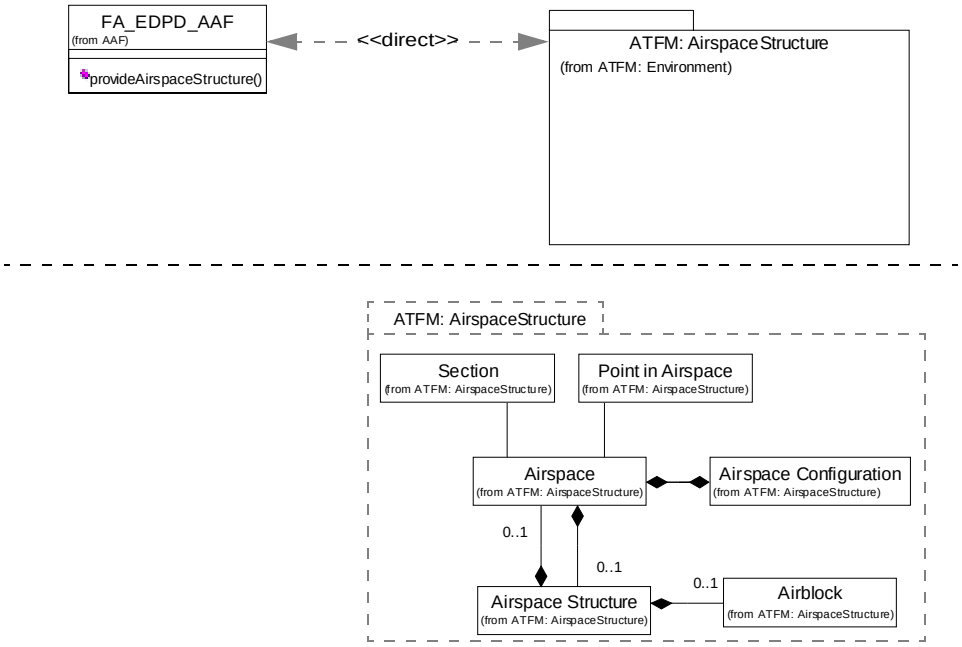
Package-Operation Correspondence

pc-op Correspondence

Classification: inter-construct-kind correspondence
inter-construct-level correspondence

Package-operation correspondences are part of the advanced correspondence relationships in component schemas, that remain uncovered in the schema inetgration community based on the relational data model.

Figure 147 Example of Package-Operation Correspondence



Following the naming conventions, the class specification FA_EDPD_AAF offers a rather complex service to its client, the provision of an airspace structure. WordNet [WN98] provides the following definition for this term:

> "Airspace: -- (the atmosphere above a nation and deemed to be under its jurisdiction)"

The bottom half of Figure 147 depicts the contents of the package ATFM: AirspaceStructure, i.e. the classes describing the decomposition of the abstract concept airspace. A correspondence among an operation and package holds, when the terms within the different artefacts are synonymous to each other and the class specifications contained in the package are meronyms of the package name. In this case the assertion holds, since the package even contains a class carrying the same name (class Airspace Structure). Class AirspaceStructure encapsulates the airspace configuration at a given moment in time and therefore justifies the asserted package-operation correspondence.

Dependency-Operation Correspondence

de-op Correspondence

A dependency-operation correspondence can be excluded for component schemas, since dependencies in static structure diagrams have been restricted to packages in order to avoid behavioural elements in structural specification. Moreover, anonymous link artefacts cannot correspond to artefacts.

D.2.4 Association Correspondence

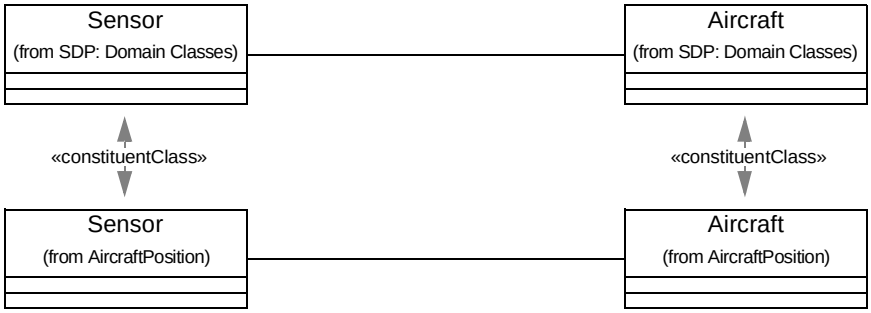
Association-Association Correspondence

as-as Correspondence

Classification: inter-construct-kind correspondence
intra-construct-level correspondence
Categories: «constituentClass», «direct» & «synonym»

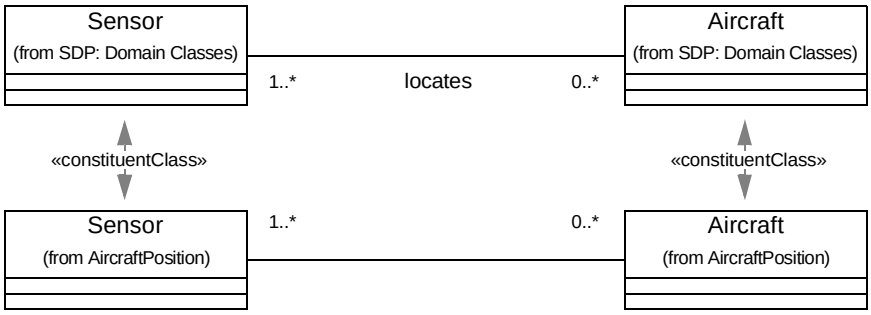
According to Whitmire's [Whi97] categorisation of external correspondences, association-association correspondences form the most important part. Actually there are three distinct forms of correspondences with respect to relationships in component schemas, originated by the fact that relationships may or may not carry a name.

Figure 148 Association-Association Correspondence: Constituent Class Correspondence



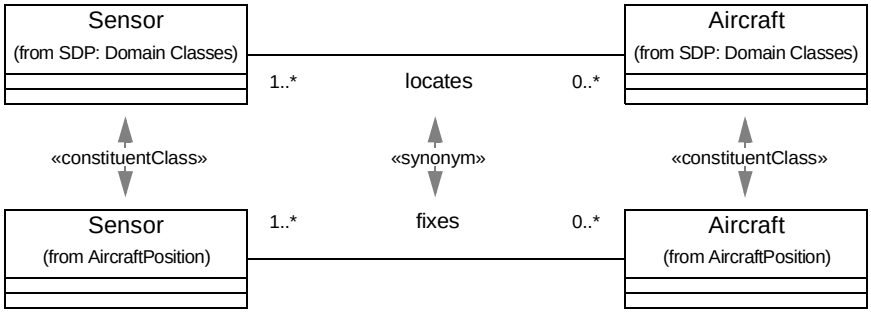
A constituent class correspondence can be assumed, whenever the relationships are anonymous and their constituent classes correspond on the basis of a class-class correspondence (see p. 241). Thus the names of the constituent classes either shall directly correspond or shall be synonyms to each other. The same assumption holds, when only one of the two relationships carries a name.

Figure 149 Association-Association Correspondence: Constituent Class Correspondence



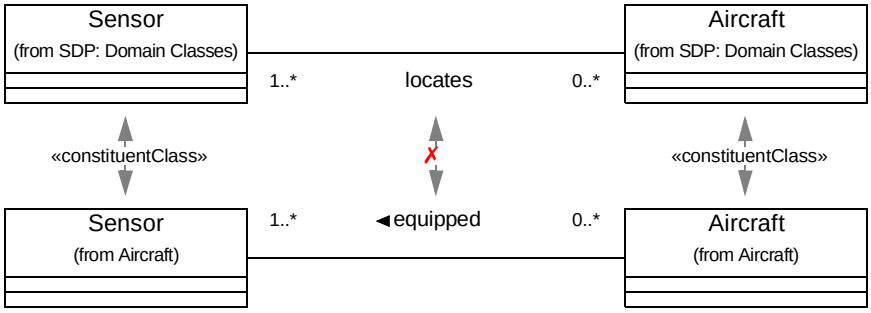
The applied cardinalities provide the second criteria for the formulation of correspondence assertion, which holds in the bottom half of Figure 149, since the same cardinality constraint has been applied. Differing cardinality constraints trigger a negotiation among the involved domain leaders, in order to state or reject the assumed correspondence. In cases of conceptual dissimilarity, a unique name is assigned to the previously anonymous instance of metaclass `Association`, in order to exclude the two relationships from future correspondence assumptions (see Figure 151).

Figure 150 Association-Association Correspondence: Association Name Correspondence



An association name correspondence can be assumed, whenever two relationship names correspond, directly as well as synonymously, and their constituent classes also correspond, i.e. a constituent class correspondence holds. Similar to the previous correspondence differing cardinality constraints trigger a negotiation between the involved domain leaders. Relationship names consequently distinguish between correspondences, as we will show in Figure 151.

Figure 151 Association-Association Correspondence: Class Name Conflict



Although the constituent classes correspond, the relationships may be dissimilar, as expressed in the previous figure. While in the upper part models the detection of an aircraft by a radar sensor, the lower part represents different forms of aircraft equip-

ment, such as collision avoidance systems and weather radar. This dissimilarity can be concluded from the conceptual difference of the involved terms »locates« and »equipped«, where the second one denotes a property of an aircraft.

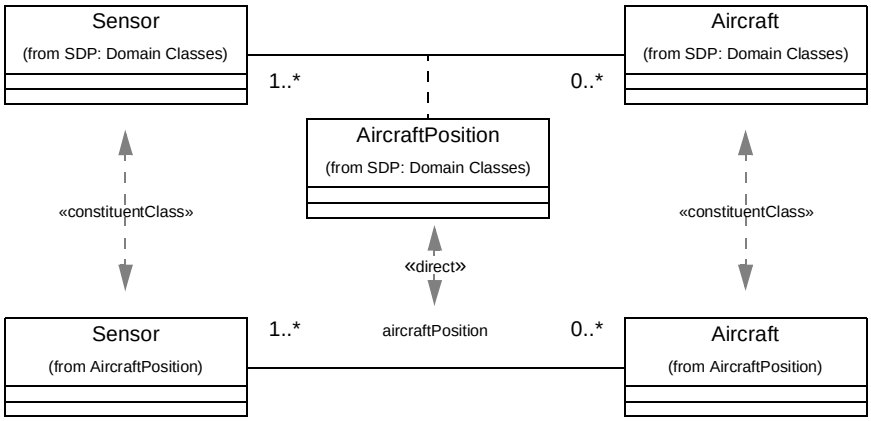
Association Class-Association Correspondence

ac-as Correspondence

Classification: inter-construct-kind correspondence
intra-construct-level correspondence
Categories: «constituentClass», «direct» & «synonym»

Within the external conflicts, association class-association correspondences are quite complex. Figure 152 depicts in the most simple case:

Figure 152 Example of an Association Class-Association Correspondence



However, from a linguistic point of view this correspondence is more complex. Following the naming conventions in Section 6.1.1.2, relationship names should be verbs. However, in the example, a compound noun has been used, stressing the character of class `AircraftPosition` as a concept in its own right, with special consistency constraints, namely that instances depend on the existence of their consistuent classes, in this case `Aircraft` and `Sensor`. Thus the assessment of the correspondence entirely depends on the structure of the underlying ontology. In the worst case, i.e. when a distinct *task* and *domain ontologies* have been used, there is no possible way of detecting this correspondence on an automated basis, since no relationship between the involved concepts can be found in the ontology. As a matter of consequence, association-class association corresondences call for additional attention in the correspondence detection cycle.

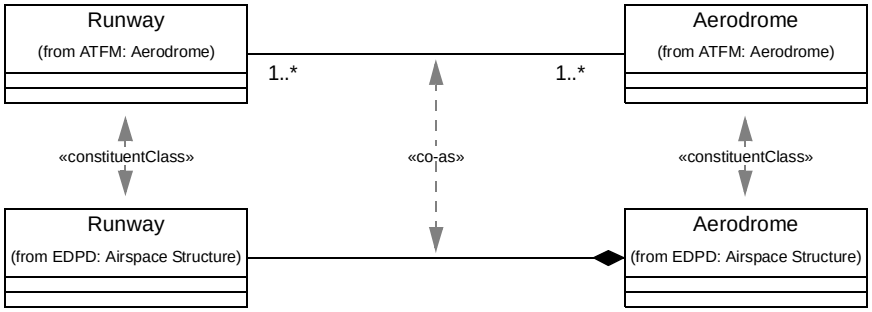
Composition-Association Correspondence

co-as Correspondence

Classification: inter-construct-kind correspondence
intra-construct-level correspondence

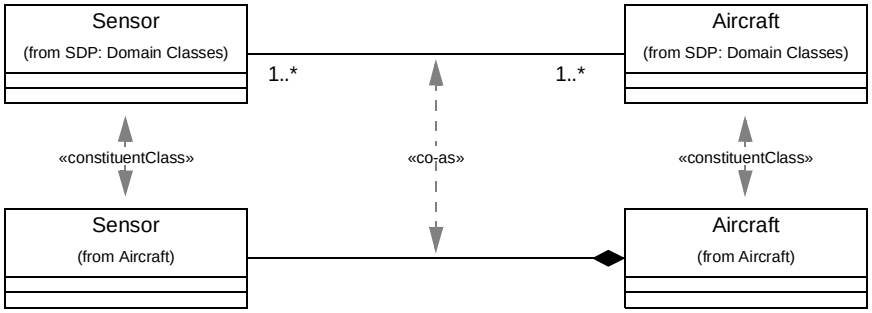
Compared to ordinary relationships, UML compositions express additional constraints affecting the life-cycle of the involved class specifications connected by the composition. This correspondence is therefore based on the association-association correspondence (see p. 259) and depends on corresponding constituent classes.

Figure 153 Example of a Composition-Association Correspondence



From a linguistic point of view, the one concept shall be meronym to the other. With respect to the depicted example, for each aerodrome there must exist at least one runway. Otherwise, we would not consider it as an aerodrome. In the bottom half of Figure 153, however, an aerodrome is part of a runway, contradicting the real-world situation. So our domain ontology states the similarity of the classes `Runway` and `Aerodrome`, but also indicates, that the composition's direction has been mistaken. This observation leads to a new category of conflicts, which we call *conceptual conflicts*. This category of conflicts arises whenever the applied construct contradicts concepts in the domain ontology. In this case the direction of the composition has been misplaced.

Figure 154 Example of a Composition-Association Conflict



The second example leads to a more complicated situation. The assumed correspondence in this case is based on a constituent class-class correspondence and may be asserted at first glance. The example, however, involves two completely different contexts, in which similar terms may play an important role. The classes from the package `Domain Classes` in the component schema `SDP` depict the observation of aircraft in the surveillance domain. The classes from the package `Aircraft`, however, depict a set of sensors which can be found in aircraft. So the assumed composition-association correspondence cannot be stated. Surveillance sensors exist without the presence of aircraft, while the on-board sensors are crucial for proper conduct of a flight. This example demonstrates another conflict category, which we call *concept context conflict*. *Concept context conflicts* arise whenever the meaning of terms varies over given domains. So an air traffic management domain ontology may not incorporate a detailed description of aircraft equipment and therefore will mistake the polysem »sensor«.

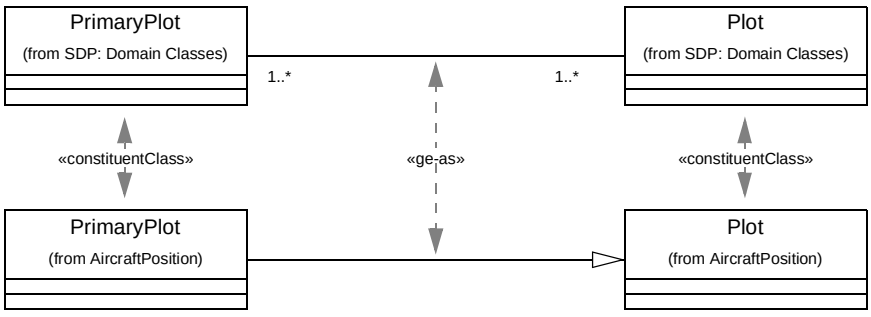
Generalisation-Association Correspondence

ge-as Correspondence

Classification: inter-construct-kind correspondence
intra-construct-level correspondence
Categories: «constituentClass», «direct» & «synonym»

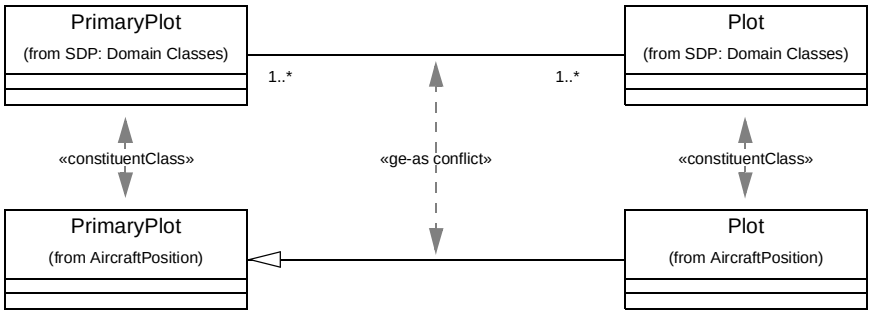
Being an association correspondence, generalisation-association correspondences are again based on a constituent class correspondence, as described in Figure 149. Unlike simple association-association correspondence, a semantic relationship is implied in this correspondence which leads to two different forms. In the first example, a constituent class correspondence holds and the concept »plot« is listed as a generalisation of concept »primary plot« in the domain ontology. As a result, the representation in the component schema `SDP` does not regard set inclusion semantics of the ontology and therefore proposes wrong cardinality constraints. In brief, the first representation cannot be deemed minimal, since due to the known correspondence from the ontology, a set of synonymous attributes and operation correspondences must exist.

Figure 155 Example of a Generalisation-Association Correspondence



Due to the erroneous direction of the generalisation in our second example, a generalisation-association conflict can be assumed. Since the term »plot« is listed as a hypernym of term »primary plot«, a conceptual conflict arises in the package `AircraftPosition`. As a result, the minimality of the model is not assured and it may also contradict domain experts' experience.

Figure 156 Example of a Generalisation-Association Conflict



Finally, the conceptual generalisation/specialisation relationship between terms which might be mistaken as ge-as correspondence is discussed in the class-class correspondence at the beginning of this section.

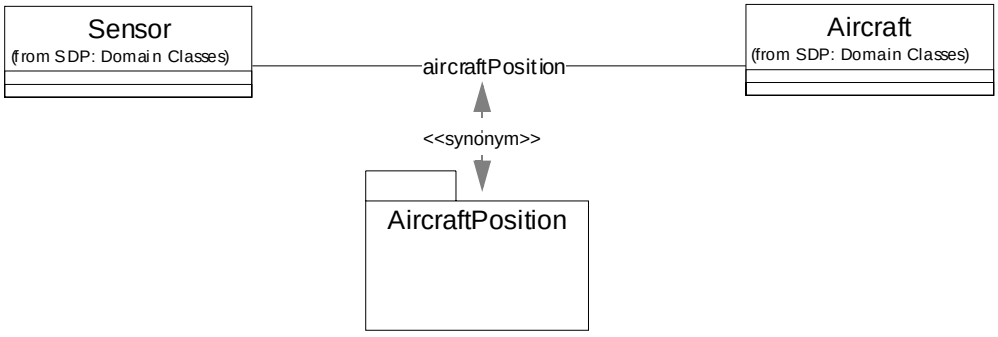
Package-Association Correspondence

pc-as Correspondence

Classification: inter-construct-kind correspondence
inter-construct-level correspondence

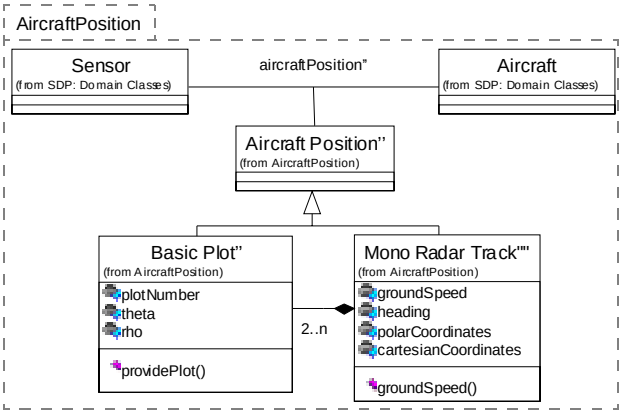
A package-association correspondence only holds within a number of restrictions. First and most important, the association must be named, otherwise there is no relevant evidence for a correspondence assertion at all.

Figure 157 Example of a Package-Association Correspondence



Packages in UML provide a grouping mechanism for classes and associations. Consequently the asserted correspondence between a package and an association is related to a refinement of the association, i.e. properties have to be added to the association within the package leading to an association class. The specialised classes `Basic Plot''` and `Mono Radar Track''` in the package `Aircraft Position` in Figure 158 present an example of this kind of refinement. On the basis of the association correspondence constituated by the classes `Sensor` and `Aircraft`, the package association correspondence holds. But the concept similarity established by the term »Aircraft Position« alone only holds when the domain ontology states the relational character of the term. So a package-association class correspondence has a higher relevance then a simple package-association correspondence.

Figure 158 Example of a Package-Association Correspondence

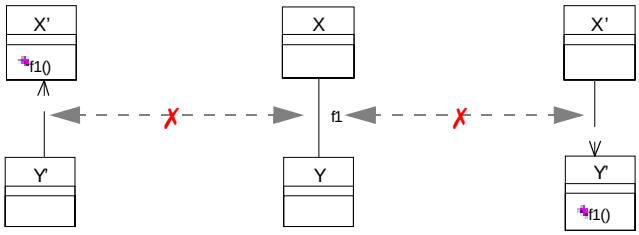


Dependency-Association Correspondence

de-as Correspondence

Although dependencies between class specifications are not part of component schemas, there is still an interesting aspect behind this correspondence. First of all, behavioural aspects have been mixed with structural ones, since dependencies are reserved for service provision of classes, i.e. method invocations. The other argument against this relationship is the anonymous character of a dependency. Therefore this situation must not be confused with the model in Figure 159 and can therefore be discarded.

Figure 159 Example of a Dependency-Association Correspondence



D.2.5 Association Class Correspondence

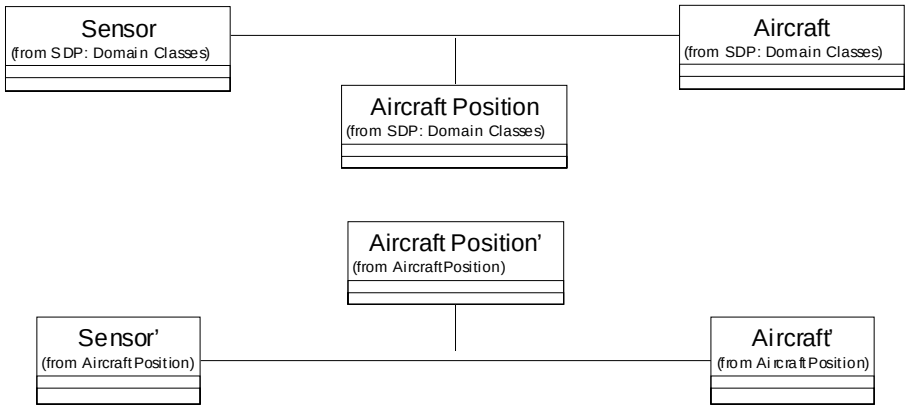
Association Class-Association Class Correspondence

ac-ac Correspondence

Classification:intra-construct-kind correspondence
intra-construct-level correspondence

An association class - association class correpondence holds in the case that the artefacts in question express the same concept.

Figure 160 Example of an Association Class-Association Class Correspondence

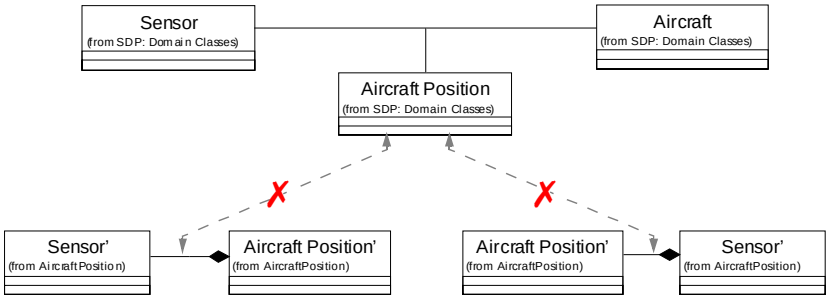


Composition-Association Class Correspondence

co-ac Correspondence

Correspondences between association classes and compositions cannot exist in component schemas, due to the anonymous connecting character of compositions. However, this situation is not to be confused with Figure 161, which depicts a class-association class correspondence.

Figure 161 Example of a Composition-Association Class Correspondence

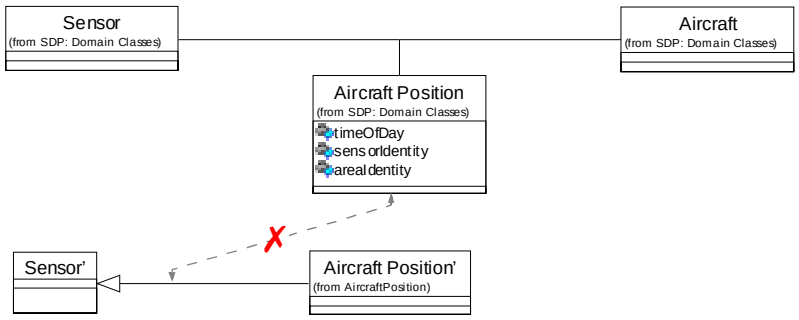


Generalisation-Association Class Correspondence

ge-ac Correspondence

In component schemas, association classes do not have any correspondences to generalisations, due to the anonymous character of generalisations. This situation is not to be confused with an association class-class correspondence depicted in Figure 162. Therefore this correspondence can be discarded.

Figure 162 Illegal Generalization-Association Class Correspondence



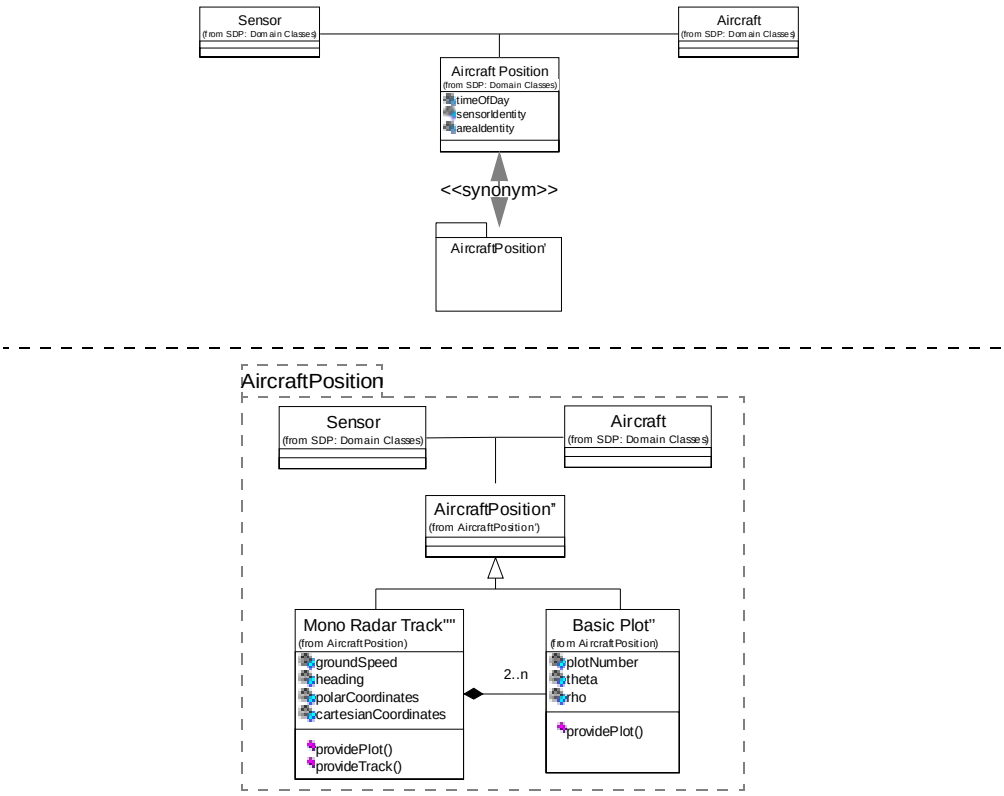
Package-Association Class Correspondence

pc-ac Correspondence

Classification:inter-construct-kind correspondence
inter-construct-level correspondence

Unlike the package association correspondence, relationships in this case always carry a name, through the attached association class. As a result, synonymous concept names in the artefacts provide strong evidence for a correspondence assertion. The evidence is further increased when the package directly references the same or corresponding classes within the package. In our example, the classes `Sensor` and `Aircraft` are defined within the package `SDP: Domain Classes` and have been imported to the package `AircraftPosition'`. Similar to the package-class correspondence, the package shall contain a class with a synonymous name (`AircraftPosition''`). Within the package this class has to be refined, depicting meronyms or specialised as depicted in our example (see Figure 163, p. 268).

Figure 163 Example of a Package-Association Class Correspondence

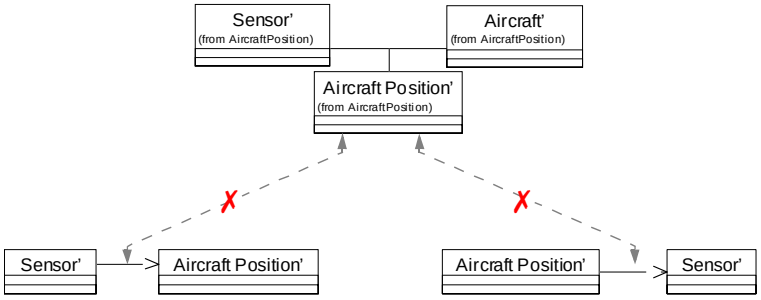


Dependency-Association Class Correspondence

de-ac Correspondence

Dependency-association class correspondences in component schemas are invalid, since dependencies are uniquely reserved for representing dependencies among packages. Besides this restriction, behavioural and structural aspects are confused in Figure 164, where the only valid correspondence might be asserted among association classes and classes.

Figure 164 Example of a Dependency-Association Class Correspondence



D.2.6 Composition-Correspondences

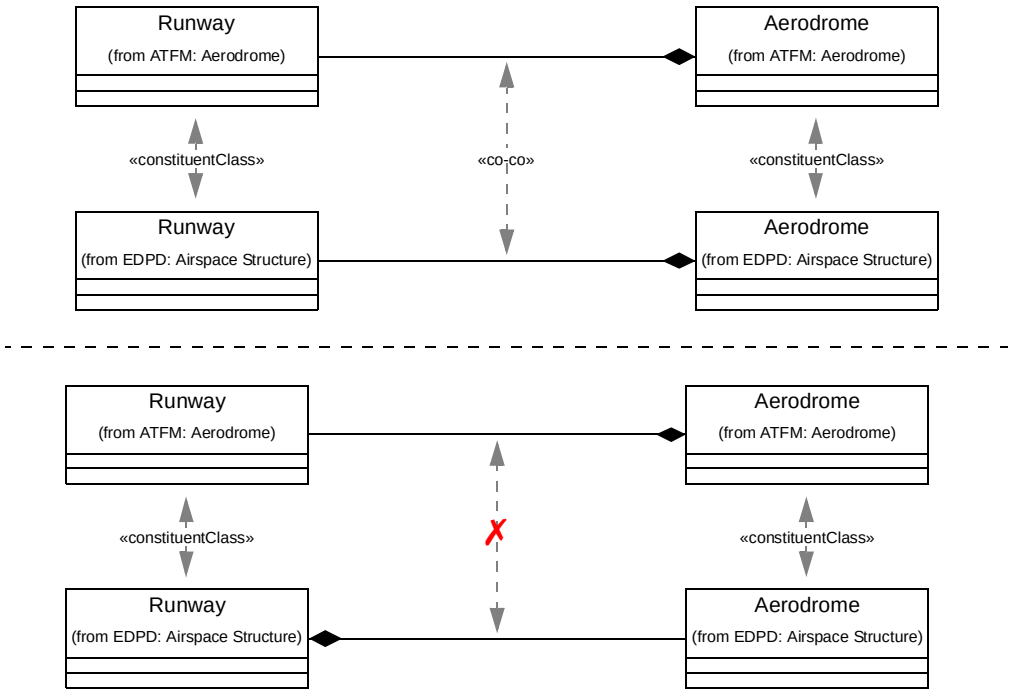
Composition-Composition Correspondence

co-co Correspondence

Classification: intra-construct-kind correspondence
intra-construct-level correspondence

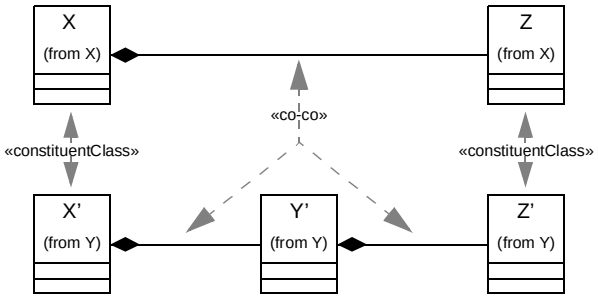
The composition-composition correspondence is once again based on a constituent class correspondence, where the classes participating in the relationship correspond on a conceptual level. Unlike the constituent class correspondence, additional integrity constraints apply to compositions.

Figure 165 Example of Composition-Composition Correspondence and Conflict



Aggregates can only exist with their components, i.e. their life-times coincide. However, this is only the structural perspective, i.e. with respect to notation and semantics of component schema's static structure diagrams. On a conceptual level with respect to the domain ontology, aggregates are holonyms, while the components are meronyms. This distinction leads to the bottom half of Figure 165. The concept `Aerodrome` is represented as a component of `Runway` contradicting human perception. Consequently, we may detect a composition-composition conflict on the basis of the domain ontology.

Figure 166 Composition-Composition Correspondence



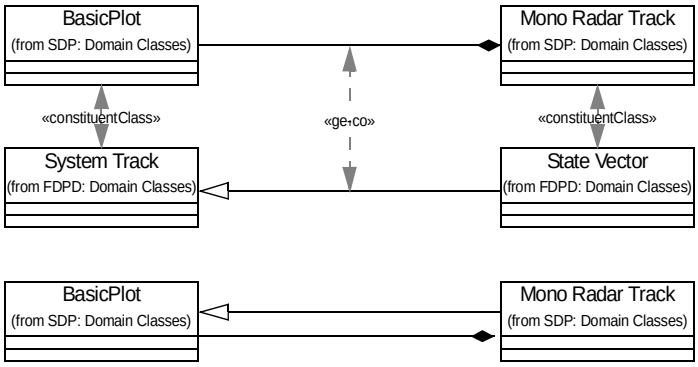
Generalisation-Composition Correspondence

ge-co Correspondence

Classification: inter-construct-kind correspondence
intra-construct-level correspondence

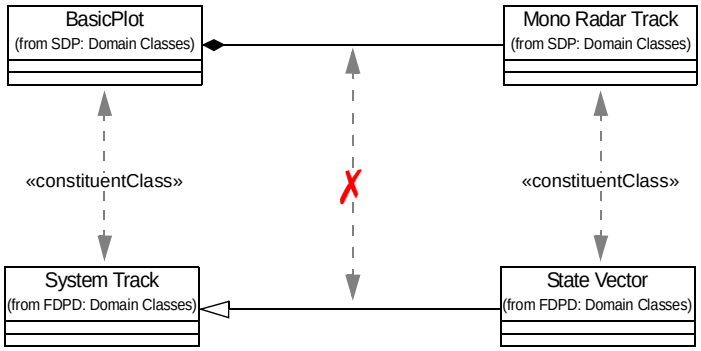
The following correspondence demonstrates another interesting extension of the constituent-class correspondence. In this case however, the classes are connected via a composition and a generalisation relationship at the same time. Due to the directional character of the two associations, we have to take their actual orientation into account. In the first case, their direction points in the inverse direction leading to a compositor-pattern correspondence. This correspondence is related to the application of the compositor pattern according to Gamma et al. [GHJ+95]. According to Gamma, this pattern can be used for uniform treatment of composed and simple objects and has also been used in the metamodel for complex use cases in this work (see Figure 41, p. 88). From a linguistic point of view, this distinction is quite complicated, since in most cases an abstract concept is introduced for the generalised class specification. A hypernym/hyponym relationship between the classes provides a minimal criteria for this correspondence. Alternatively, the ontology might state a meronym / holonym relationship. Generally speaking this kind of correspondence calls for careful review by the domain leaders and experts.

Figure 167 Generalisation-Composition Correspondence



Although the example in Figure 168 depicts a clear generalisation-composition conflict, two class specifications can never be parts and generalisations of each other at the same time.

Figure 168 Generalisation-Composition Conflict

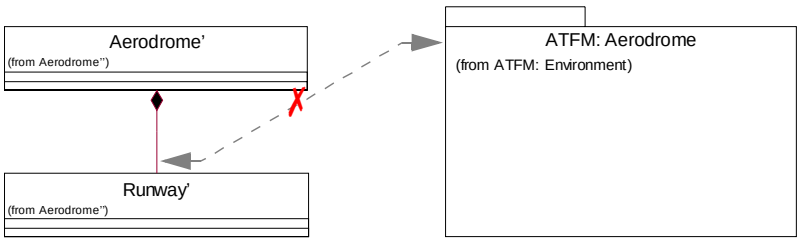


Package-Composition Correspondence

pc-co Correspondence

Correspondence between anonymous link artefacts and artefacts such as packages can generally be discarded for the evaluation. However, it is not to be confused with package-class and an attribute class correspondence as depicted in Figure 169.

Figure 169 Example of a Package-Composition Correspondence



Dependency-Composition Correspondence

de-co Correspondence

A dependency-composition correspondence can be discarded from our analysis, since behavioural and structural aspects have to be mixed in this relationship. Furthermore, both constructs are anonymous, i.e. their correspondence can only be clarified via their constituent classes, and will consequently be discarded in this work. This situation in particular must not be confused with an association-composition correspondence, depicted in Figure 170, p. 272.

Figure 170 Illegal Dependency-Composition Correspondence

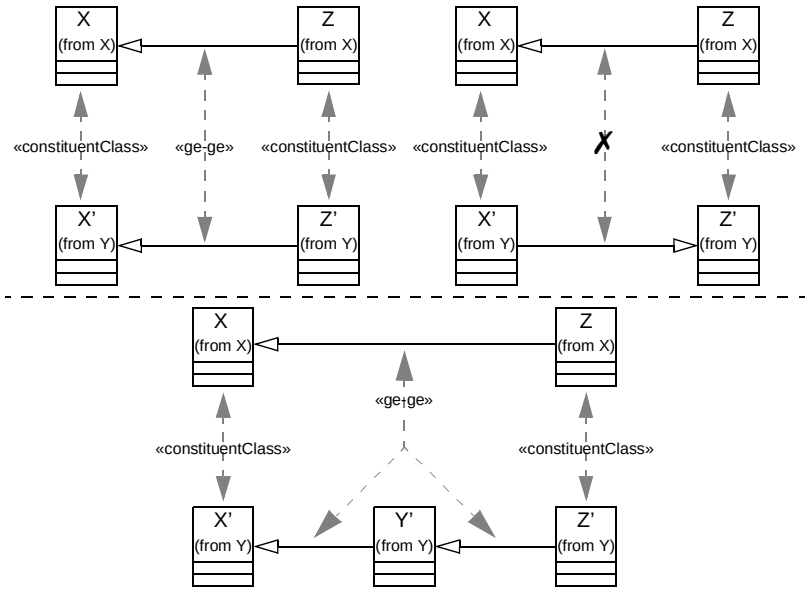
D.2.7 Generalisation Correspondence

Generalisations -Generalisations Correspondence

ge-ge Correspondence Classification: intra-construct-kind correspondence
intra-construct-level correspondence

Like all link artefact correspondences, the generalisation-generalisation correspondence is based on a constituent class correspondence. Similar to compositions, generalisation/specialisation is a directional relationship. Whenever the corresponding class specifications have connected with generalisation, but with different direction, we have a generalisation-generalisation conflict. More complicated is the second situation, which aims for the transitivity property of generalisation.

Figure 171 Example of a Generalisation-Generalisation Correspondence and Conflict

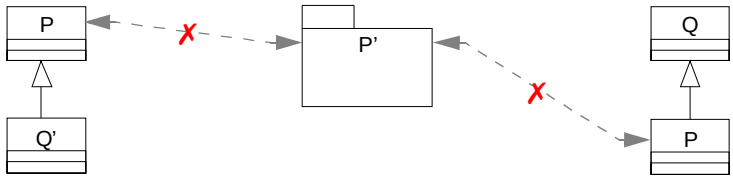


Package-Generalisation Correspondence

pc-ge Correspondence

Correspondences between anonymous link artefacts and grouping artefacts can generally be discarded. However, they must not be confused with a package class correspondence as depicted in Figure 172.

Figure 172 Example of a Package-Generalization Correspondence

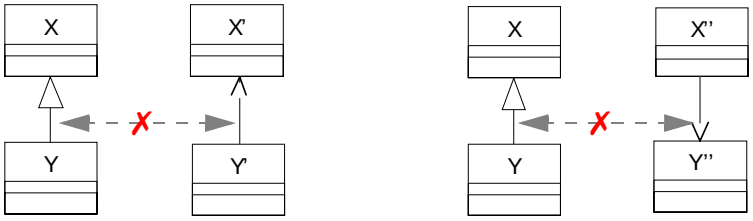


Dependency-Generalization Correspondence

de-ge Correspondence

Mixing structural and behavioural aspects automatically prohibits this kind of correspondence. Restricting dependencies to packages further stresses this claim. However, this situation is not to be confused with a generalisation-association correspondence.

Figure 173 Illegal Dependency-Generalization Correspondence



D.2.8 Package Correspondence

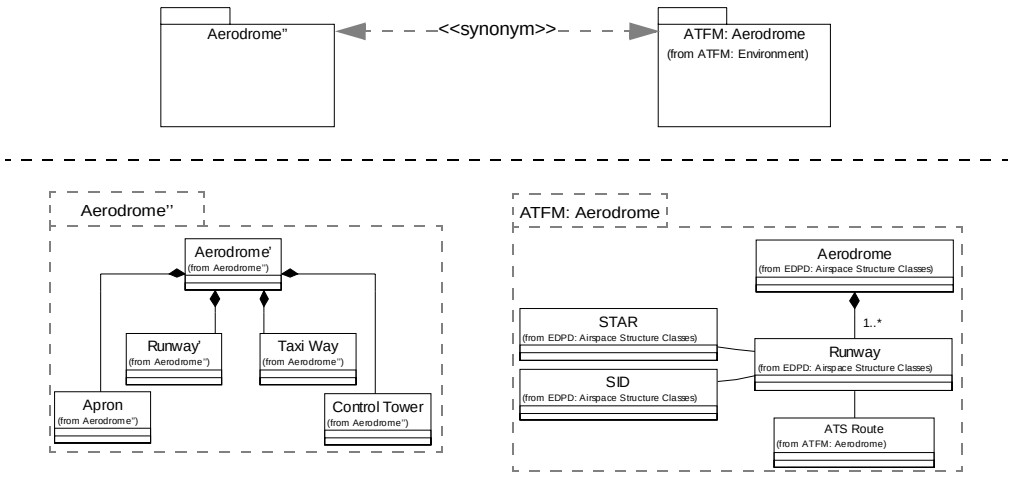
Package-Package Correspondence

pc-pc Correspondence

Classification: intra-construct-kind correspondence
intra-construct-level correspondence

Package correspondence are expressed by synonymous or identical names. The individual package contents shall contain meronyms of the package name. Similarity assessment, a class-class correspondence, has to be performed for the package contents in order to identify similar concepts, such as the synonymous classes `Aerodrome` depicted in Figure 174, p. 274.

Figure 174 Example of a Package-Package Correspondence

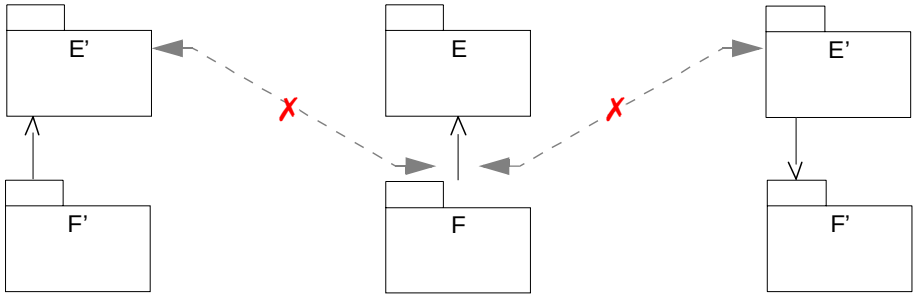


Dependency-Package Correspondence

de-pc Correspondence

Packages cannot correspond to dependencies, due to the anonymous connecting character of dependencies. However, this situation is not to be confused with a dependency-dependency correspondence based on the constituent packages.

Figure 175 Example of a Dependency-Package Correspondence



D.2.9 Dependency Correspondence

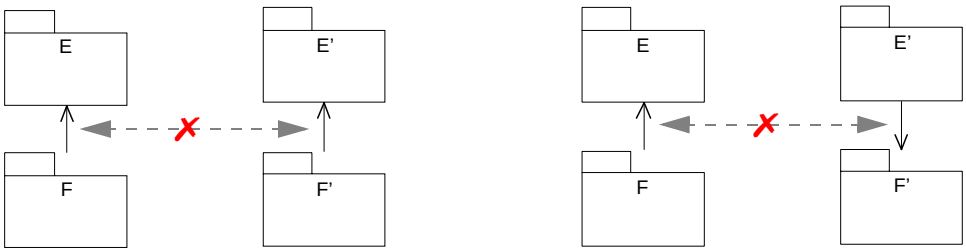
Dependency-Dependency Correspondence

de-de Correspondence

Classification: intra-construct-kind correspondence
intra-construct-level correspondence

Similar to association-association correspondences UML package dependences depend on a constituent element correspondence, classes in this case. As we pointed out in all correspondences related to packages, the similarity of packages can only be expressed through the meaning of their names. Otherwise complex similarity metrics for package contents have to be designed, which is not the scope of this work.

Figure 176 Example of a Dependency-Dependency Correspondence



References

[AEQ98] J. Arlow, W. Emmerich and J. Quinn
Literate Modelling -
Capturing Business Knowledge with the UML
In P.-A. Muller and J. Bézivin (ed.)
Proceedings of UML'98 International Workshop Mulhouse, France, June 3 - 4, 1998

[AHM+93] E. Aslaksen, I. Hooks, R. Mengot and K. Ptack
What is a Requirement?
in R. Thayer and M. Dorfmann (ed.)
Software Requirements Engineering 2nd Edition
IEEE Computer Society Press
1997

[AR92] E. Andersen and T. Reenskaug
System Design by Composing Structures of Interacting Objects
Proceedings of the European Conference on Object-Oriented Programming, ECOOP'92
Springer-Verlag
1992
pp. 133-152

[Bal95] Heide Balzert
Methoden der objektorientierten Systemanalyse
BI Wissenschaftsverlag
Mannheim 1995

[BBD+95] E. Boiten, H. Bowman, J. Derrick, and M. Steen
Cross viewpoint consistency in open distributed processing (intra language consistency)
Technical Report 8-95
University of Kent, Canterbury, UK
June 1995

[BBM+89] A. Borgida, R. J. Brachman, D. L. McGuiness and L. A. Resnick
CLASSIC: A Structural Data Model for Objects
Proceeding of 1989 SIGMOD Conference on the
Management of Data
ACM Press
1989
pp. 58-67

[BCN92] C. Batini, S. Ceri and S. Navathe
Conceptual Database Design - An Entity-Relationship Approach
Benjamin/Cummings
1992

[Ber93] E. Berard (ed.)
Essays on Object-Oriented Software Engineering Volume I
Prentice Hall
1993

[BJR98] G. Booch, I. Jacobson and J. Rumbaugh
The Unified Modeling Language User Guide
Addison-Wesley
1998

[BK91] C. Ball and R. Kim
An Object Oriented Analysis Of Air Traffic Control
WP 90W00542
MITRE Corporation, McLean, Virginia
1991
http://www.caasd.org/Papers/WP/90W542/index.html

[BLN86] C. Batini, M. Lenzerini and S. Navathe
A Comparative Analysis of Methodologies for Database Schema Integration
ACM Computing Survey, Vol. 8 No. 4, December 1986

[Boe87] B. Boehm
A Spiral Model of Software Development and Enhancement
in R. Thayer and M. Dorfmann (ed.)
Software Requirements Engineering 2nd Edition
IEEE Computer Society Press
1997
pp. 416 - 430

[Boo94] Grady Booch
Object-Oriented Analysis and Design with Applications -
second Edition
Benjamin/Cummings
1994

[Bun77] M. Bunge
Treatise on Basic Philosophy I: The Furniture of the World
Riedel
Dordrecht 1977

[CB97] M. Jahrke et al.
ConceptBase - A deductive object manager for meta databases
http://www-i5.informatik.rwth-aachen.de/CBdoc/

[Chen76] P. P. Chen
The Entity Relationship Model: Towards a Unified View of Data
ACM Transaction on Database Systems, 1, 1976

[Cho65] N. Chomsky
Aspects of a Theory of Syntax
Harvard University Press
1965

[CH74] R. H. Campbell and A. N. Habermann in
The Specification of Process Synchronization by Path Expressions
E. Gelembe and C. Kaiser (Ed.)
Operating Systems
Lecture Notes in Computer Science 16
Springer Berlin, Heidelberg, New York
1974

[CLN98] D. Calvanse, M. Lenzerini and D. Nardi
Desrciption Logics for Conceptual Data Modelling
in J. Chomicki and G. Saake (ed.)
Logics for Databases and Information Systems
Kluwer Academic Publishers
1998

[Con97] S. Conrad
Föderierte Datembanksysteme -
Konzepte der Datenintegration
Springer Verlag Berlin
1997

[CY91] P. Coad and E. Yourdon
OOA — Object-Oriented Analysis
Prentice-Hall
1991

[Dar90] S. Dart
Spectrum of Functionality in Configuration Management
Technical Report CMU/SEI-90-TR-11
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
1990

[Dav93] A. M. Davis
Software Requirements - Object, Functions & States
Prentice Hall
1993

[DeM79] T. DeMarco
StructuredAnalysis and System Specification
Prentice Hall
1979

[DeM82] T. DeMarco
Controlling Software Projects
Yourdon Press
1982

[DLF93] A. Dardenne, A. van Lamsweerde and S. Fickas
Goal-Directed Requirements Acquisition
Science of Computer Programming Vol. 20
North Holland
1993
pp. 3-5

[ElN94] R. Elmasri and S. Navathe
Fundamentals of Database Systems 2nd Edition
Benjamin /Cummings
1994

[Eur96a] European Organisation for the Safety of Air Navigation
Operational Reqiuirements Document for EATCHIP Phase III
ATM Added Functions Volume 0 - General
OPR.ET1.ST04.DEL01.01
Brussels, October 1st1996.

[Eur96b] European Organisation for the Safety of Air Navigation
Surveillance Reference Model
SUR.ET1.ST05.1110-TYP-00-01
Brussels, October 22nd1996.

[Eur97] European Organisation for the Safety of Air Navigation
Operational Requirements for Flight Data Processing and Distribution Core Functions (Area Control)
OPR.ET.ST03.1000-ORD-01-00
Brussels, February 11th1997.

[Eur98a] European Organisation for the Safety of Air Navigation
Overall CNS/ATM Architecture for EATCHIP Volume 3: Object Model
ASE.ETI.ST02-ADD-01-02
Brussels, March 3rd1998.

[Eur98b] European Organisation for the Safety of Air Navigation
Overall CNS/ATM Architecture for EATCHIP Volume 3: Object Model
Annex A: Flight Data Processing and Distribution Core Functions
ASE.ETI.ST02-ADD-01-02
Brussels, March 3rd1998.

[Eur98c] European Organisation for the Safety of Air Navigation
Overall CNS/ATM Architecture for EATCHIP Volume 3: Object Model
Annex C: Environment Data Processing and Distribution
ASE.ETI.ST02-ADD-01-02
Brussels, March 3rd1998.

[Eur98d] European Organisation for the Safety of Air Navigation
Overall CNS/ATM Architecture for EATCHIP Volume 3: Object Model
Annex H: Human Machine Interface
ASE.ETI.ST02-ADD-01-02
Brussels, March 3rd1998.

[FFR96] Farquhar, A., R. Fikes, and J. Rice
The Ontolingua Server: a Tool for Collaborative Ontology Construction
In KAW96. November 1996.
Also available as KSL-TR-96-26
1996

[Fir93] D. G. Firesmith
Object-Oriented Requirements Analysis and Logical Design
John Wiley
1993

[FM96] K. Forsberg and H. Mooz
System Engineering Overview
in R. Thayer and M. Dorfmann (ed.)
Software Requirements Engineering 2nd Edition
IEEE Computer Society Press
1997
pp. 44 - 72

[GG95] N. Guarino and P. Giaretta
Ontologies and Knowledge Bases: Towards a Terminological Clarification
In N. J. I. Mars (ed.)
Towards Very Large Knowledge Bases
IOS Press 1995

[GHJ+95] E. Gamma, R. Helm, R. Johnson and J. Vlissides
Design Patterns - Elements of Reusable Object-Oriented Software
Addison-Wesley 1995

[GHS96] J. Galler, J. Hagemeyer and A.W. Scheer
Asynchronous Cooperation Support for Distributed and Collaborative Information Modelling
in K. Sandkuhl and H. Weber
Telekooperations-Systeme in dezentralen Organisationen
ISST Bericht 31/96
February 1996

[GL93] J. Goguen and C. Linde
Techniques for Requirements Elicitation
in R. Thayer and M. Dorfmann (ed.)
Software Requirements Engineering 2nd Edition
IEEE Computer Society Press
1997

[Gla97] R. Glass
Software Runaways
Prentice Hall
1997

[Glaß97] Uta Glaß
Von Anwendungsszenarien zu dynamischen Objekten
Diplomarbeit am Fachbereich Informatik
Technische Universität Berlin
1997

[God97] Dörte Godulla
Erstellung von Software Requirements Spezifikationen aus Use Cases
Diplomarbeit am Fachbereich Informatik
Technische Universität Berlin
1997

[GR83] A. Goldberg and D. Robson
Smalltalk-80 The Language
Addison-Wesley
1983

[Gua98] N. Guarino
Formal Ontology and Information Systems
In N. Guarino (ed.)
Proc. of the 1st International Conference Formal Ontology in Information Systems
Trento, Italy, 6-8 June 1998
IOS Press
1998

[Har87] D. Harel
Statecharts: A Visual Formalism for Complex Systems
Science of Computer Programming, vol. 8
1987

[Hars97] Alexander Hars
Natural language-based data modeling: Improving validation and integration
http://www-rcf.usc.edu/~hars/pub/1997/nldb/nldb97r1.html
1st revision, submitted to
Journal of Database Management
March 1997

[HFW97] K. Hennig, P. Florath and J. Wortmann
CNS/ATM Data Dictionary - User Manual
1997

[HL96] R. Holowczak and W. Li
A Survey on Attribute Correspondance and Heterogeneity Metadata Representation
Proceedings of the First IEEE Metadata Conference
April 16-18, Silver Spring Maryland,1996

[HM93] J. Hammer and D. McLeod
An Approach to Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems
International Journal of Intelligent & Cooperative Information Systems, World Scientific
Vol. 2, Num. 1, 1993
pp. 51-83

[HM97] A. Hars and J.T. Marchewka
The Application of Natural Language Processing for Requirements Analysis
http://www-rcf.usc.edu/~hars/pub/1997/nlra
submitted to Journal of Management Information Systems
March 1997

[IBM95] IBM Deutschland Entwicklungs GmbH
IBM FlowMark: Modelling Workflow
Document Number SH19-8241-02
Version 2 Release 3
1995

[IEEE830] Institute of Electrical and Electronics Engineers
IEEE Recommended Practice for Software Requirements Specifications
IEEE Std. 830-1993
IEEE New York 1993

[ISO8807] ISO/IS8807
LOTOS: Language for the Temporal Ordering Specification of Observational Behaviour
1987

[ISO10027] ISO/IEC 10027
Information Technology - Information Resource Dictionary System (IRDS) Framework
1990

[ITU95] ITU: *Algebraic Semantics to Message Sequence Charts*
1995

[Jack90] M. Jackson
Some Complexities in Computer-Based Systems and Their Implications for System Development
Proceedings of International Conference on Computer Systems and Software Engineering (CompEuro ’90)
Tel-Aviv, Israel, 8- 10th May 1990
IEEE Computer Society Press
1990
pp. 344-351

[JBR99] I. Jacobson, G. Booch and J. Rumbaugh
Unified Software Development Process
Addison Wesley
1999

[JCJ+94] I. Jacobson, M. Christerson, P. Jonsson and G. Övergaard
Object-Oriented Software Engineering - A Use Case Driven Approach
Addison Wesley, 1994

[Jeu92] M. A. Jeusfeld
Änderungskontrolle in deduktiven Objektbanken
Infix-Verlag, St. Augustin, Germany
1992

[KASP96] E.U. Kriegel, J. Altenhein, M. Schlösser-Fassbender and P. Thierse
Eine Softwareproduktionsumgebung für kleine und mittelständische Unternehmen
ISST-Bericht 36/96
Dezember1996

[KKA+97] E.U. Kriegel, D. Kurzmann, J. Altenhein, M. Löw and P. Thierse
Eine Software Produktionsumbegung für kleine und mittelständische Unternehmen - 2. Teil
ISSN 0943-1624
November 1997

[Klar99] Markus Klar
A Semantical Framework for the Integration of Object-Oriented Modeling Languages
Dissertation am Fachbereich Informatik
Technische Universität Berlin
1999

[Klu98] M. Klusch
Kooperative Informationsagenten im Internet
PhD Thesis, Computer Science Dept., University of Kiel
Kovac Verlag
1998

[KaS98] V. Kashyap and A. Sheth
Semantic Heterogeneity in Global Information Systems: the Role of Metadata, Context and Ontologies
in M. Papazoglou and G. Schlageter
Cooperatice Information Systems
Academic Press
1998

[KoS98] G. Kotonya and I. Sommerville
Requirements Engineering Process and Techniques
John Wiley
1998

[Kur98] D. Kurzmann
Werkzeuge und Techniken für die internetgestützte parallele Anforderungsanalyse
Diplomarbeit am Fachbereich Informatik
Technische Universität Berlin
1998

[LDL98] A. van Lamsweerde, R. Darimont and E. Letier
Managing Conflicts in Goal-Driven Requirements Engineering
IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development
IEEE November 1998

[LGP+90] D. Lenat, R. Guha, K. Pittman, D. Pratt and M. Shepherd
Cyc: toward programs with common sense
Communications of the ACM
Vol.33, No. 8, August 1990
pp. 30-49

[Lor87] P. Lorenzen
Lehrbuch der konstruktiven Wissenschaftstheorie
BI-Wissenschaftsverlag
1987

[LNE89] J. Larson, S. Navathe and R. Elmasri
A Theory of Attribute Equivalence in Databases with Application to Schema Integration
IEEE Transactions on Software Engineering, Vol. 15, no. 4 April 1989
pp. 449-463

[LRO96] A. Levy, A. Rajaraman and J. Ordille
Querying Heterogeneous Information Sources Using Source Descriptions
Proceedings of the 1996 International
Conference on Very Large Databases
Morgan Kaufmann
1996
pp. 251-262

[MAC+95] N. Maiden, P. Assenova, P. Constantopoulos, M. Jarke, P. Johanneson, H. Nissen, G. Spanoudakis and A. Sutcliffe
Computational Mechanisms for Distributed Requirements Engineering, Proceedings of the 7th International Conference on Software Engineering & Knowledge Engineering (SEKE '95), Pitsburg, Maryland, USA June 1995
pp. 8-16

[MBF+93] G.A. Miller, R. Beckwith, C. Fellbaum, D. Gross and K. Miller
Introduction to WordNet: An On-Lexical Database
ftp://ftp.cogsci.princeton.edu/pub/wordnet/5papers.pdf
Cognitive Science Laboratory
Princeton University
221 Nassau St
Princeton, NJ 08542
1993

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke and M. Koubarakis
Telos: A Language for Representing Knowledge about Information Systems
ACM Trans. Information Systems Vol. 8, no 4
1990

[MIB96] E. Mena, A. Illarramendi and J.M. Blanco
Discovering Relationships among Ontologies Describing Data Repositories Contents
Nagib Callaos (ed.)
Proceedings of the International Conference on Information Systems, Analysis and Synthesis (ISAS'96)
Orlando (Florida), USA, July 1996
pp. 474-480

[Mil95] George A. Miller
WordNet: A Lexical Database for English
In Communications of the ACM
Vol. 38, No. 11, 1995
pp. 39 - 41

[Mic97] Microsoft Corporation
Microsoft Access for Windows
1997

[Moo96] D. Moody
Integrating Stakeholder Views in Data Modelling: A Participative Approach
Proceedings of the Entity Relationship Approach ER'96
Springer Verlag Berlin
1996

[MML93] E. Métais, J.N. Meunier and G. Levreau
Database Schema Design: A Perspective from Natural Language Techniques to Validation and View Integration
Proceedings of the Entity-Relationship Approach - ER'93
Springer Verlag Berlin
1993

[MW93] Merriam-Webster
Merriam-Webster's Collegiate Dictionary
Merriam-Webster
1993

[NKF94] B. Nuseibeh, J. Kramer and A. Finkelstein
A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications
IEEE Transactions on Software Engineering 20, October 1994
pp. 760-773

[ODP P1] ISO/IEC DIS 10746-1:
Basic Reference Model of Open Distributed Processing - Part 1: Overview
1995

[ODP P2] ISO/IEC DIS 10746-2:
Basic Reference Model of Open Distributed Processing - Part 2: Foundations
1995

[OKK97] D. Oliver, T. Kelliher and J. Keegan
Engineering Complex Systems with Models and Objects
McGraw-Hill
1997

[OMG97a] Object Management Group
UML Notation Semantics
Version 1.1
OMG Document ad/97-08-04
1997

[OMG97b] Object Management Group
UML Notation Guide
Version 1.1
OMG Document ad/97-08-05
1997

[OMG97c] Object Management Group
Object Constraint Language Specification
Version 1.1
OMG Document ad/97-08-08
1997

[OMG97d] Object Management Group
OA&D CORBAfacility Interface Definiton
Version 1.1
OMG Document ad/97-08-09
1997

[OMG97e] Object Management Group
Meta Object Facility (MOF) Specificaton
Version 1.1
OMG Document ad/97-08-14
1997

[OMG97f] Object Management Group
Event Management
OMG Document ad/97-12-08
1997

[OMG98] Object Management Group
The Object Request Broker: Architecture and Specification
Version 2.2
OMG Document ad/98-02-33
1998

[Ont95] R. Patil and W. Swartout
OntoSaurus Web Browser
Information Science Institute of the University of Southern California
http://mozart.isi.edu:8003/sensus/sensus_frame.html
1995

[OR97] J. Odell and G. Ramackers
Towards a Formilization of OO Analysis
Journal of Object-Oriented Programming July /August 1997
pp. 64 - 68

[Orr81] K. Orr
Structured Requirements Definition
Topeka Kans.: Ken Orr and Associates
1981

[Ort97] E. Ortner
Methodenneutraler Fachentwurf
B. G. Teubner Verlagsgesellschaft
Stuttgart Leipzig
1997

[OS96] E. Ortner and B. Schienmann
Normative Language Approach A Framework for Understanding
Proceedings of the Entity Relationship Approach ER'96
Springer Verlag Berlin
1996

[PWC95] M.C. Paulk, C.V. Weber and B. Curtis
The Capability Maturity Model:
Guidelines for Improving the Software Process
Software Engineering Iinstitute - Series in Software Engineering
Carnegie Mellon University
Addison-Wesley
1995

[RAB96] B. Regnell, M. Andersson and J. Bergstrand
A Hierachical Use Case Model with Graphical Representation
Proceedings of ECBS '96
IEEE International Syposium on Engineering of Computer-Based Systems
March 1996

[Rat98] Rational Software Corporation
Rational Rose 98
1998

[Rat99] Rational Software Corporation
UML Notation Semantics - Version 1.3
http://www.rational.com/uml/resources/documentation/media/OMG-UML-1_3-Alpha5-PDF.zip
1999

[Ree96] T. Reenskaug
Working With Objects - The OORAM Software Engineering Method
Prentice-Hall
1996

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen
Object-Oriented Modelling & Design
Prentice Hall
1991

[RJB99] J. Rumbaugh, I. Jacobson and G. Booch
The Unified Modelling Language Reference Manual
Addison-Wesley
1999

[RK70] H. Rittel and W. Kunz
Issues as Elements of Information Systems
Arbeitspapier 131
Institut für Grundlagen der Planung I.A.
Universität Stuttgart
1970

[RP92] C. Rolland and C. Proix
A Natural Language Approach for Requirements Engineering
Proceedings of the 4th International Conference on Advanced Information Systems Engineering - CAiSE '92
Springer-Verlag
1992
pp. 257-277

[Ross77] D. T. Ross
Structured Analysis: A Language for Communicating Ideas
IEEE Transactions on Software Engineering 3,1 (January 1977)
pp.16 -34

[Rum94] James Rumbaugh
Getting Started - Using use cases to capture requirements
Journal of Object-Oriented Programming
September 94

[Sch97] B. Schienmann
Objektorientierter Fachentwurf
Teubner 1997

[Sche94] A.-W. Scheer
ARIS Toolset: A Software Product is Born
Information Systems Vol. 19, No. 8 1994
pp. 607 - 624

[Schi98] Ute Schirmer
Entwicklung von ViewPoint-Templates für ausgewählte Teilmodelle der Sprache UML
Diplomarbeit am Fachbereich Informatik
Technische Universität Berlin
1998

[Schm98] Ingo Schmitt
Schemaintegration für den Entwurf Föderierter Datenbanken
DISBIS 43
Infix
1998

[SC96] G. Spanoudakis and P. Constantopoulos
Elaborating Analogies from Conceptual Models
International Journal of Intelligent Systems, Vol. 11, No 11
R. Yager (ed.)
J.Wiley and Sons
1996
pp. 917-974

[Sel97] SELECT Software Tools plc
SELECT Enterprise 5.1
Westmoreland House
80-86 Bath Road
Cheltenham
Gloustershire, GL53 7JT, UK
1997

[Sen95] K. Knight, E. Hovy and R. Whitney
SENSUS Ontology
Information Science Institute of the University of Southern California
http://mozart.isi.edu:8003/sensus/sensus_frame.html
1995

[SG96] M. Shaw and D. Garlan
Software Architecture: Pespectives on an Emerging Discipline
Prentice Hall
1996

[SNi96] TakeFive Software GmbH
SNiFF+ User's Guide and Reference
SNiFF-URG-002
1996
http://www.takefive.com

[SPK+96] B. Swartout, R. Patil, K. Knight and T. Russ
Toward Distributed Use of Large-Scale Ontologies
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
September 27, 1996

[SoDa98] Rational Software Corporation
Rational SoDa
Cupertino 1998

[Sowa84] John Sowa
Conceptual Structures
Addison-Wesley
1984

[SPD92] S. Spaccapietra, C. Parent and Y. Dupont
Model-Independent Assertions for Integration of Heterogeneous Schemas
Very Large Databases Journal, 1(1), July 1992.

[StP99] Aonix Software Ltd.
Software through Pictures UML
Release 7.1
Aonixs
595 Market Street
San Francisco, CA 94105, USA
1999

[SS97] I. Sommerville and P. Sawyer
Requirements Engineering
John Wiley 1997

[Str93] B. Stroustrup
The C++ Programming Language
Addison Wesley
1993

[Tai99] Stefan Tai
Architectural Abstractions
Dissertation am Fachbereich Informatik
Technische Universität Berlin
1999

[TD97] R. Thayer and M. Dorfmann (Ed.)
Software Requirements Engineering -
Second Edition
IEEE Computer Society Press
1997

[Tic85] W.F. Tichy
RCS - A System for Version Control
Software - Practice and Experience, Vol. 15, No. 7
1985
pp. 637-654

[Tog99] Object International Software
Together/C++
Version 2.21
Schulze-Delitzsch-Str. 16
D-70565 Stuttgart, Germany
http://www.oisoft.com
1999

[WCK97] J. Wortmann, I. Claßen and G. Klawitter
Object-Oriented Analysis for Advanced Flight Data Management
Proceedings of the Eighth International Workshop on Software Technology and Engineering Practice
London1997

[WCW97] J. Wortmann, I. Claßen and J. Watson
Object-Oriented Requirements Analysis in the Air Traffic Control Domain
Proceedings of the Second CAiSE/IFIP8.1 International Workshop on Evaluation of Modelling Methods in System Analysis and Design
Barcelona 1997

[Web82] H. Weber
On the Ambiguous Use of Names in Database Design
Proceedings of the Second International Conference on Databases: Improving Database Usability and Responsiveness
Jerusalem, June1982

[WF97] J. Wortmann and P. Florath
Modelling Guidelines for Object-Oriented Analysis
EUROCONTROL Report April 14th 1997
EUROCONTROL, 96, Rue de la Fusée, B-1130 Bruxelles

[WFW97] J. Wortmann, P. Florath and J. Watson
Towards an Overall CNS/ATM Object Model
EUROCONTROL Report June 11th 1997
EUROCONTROL, 96, Rue de la Fusée, B-1130 Bruxelles

[Whi97] S. Whitmire
Object-Oriented Design Measurement
John Wiley
1997.

[Wie94a] G. Wiederhold
An Algebra for Ontology Composition
Proceedings of 1994 Monterey Workshop on Formal Methods
U.S. Naval Postgraduate School
Monterey CA, Sept 1994
pages 56-61

[Wie94b] G. Wiederhold
Interoperation, Mediation, and Ontologies
Proceedings International Symposium on Fifth Generation Computer Systems (FGCS94)
Workshop on Heterogeneous Cooperative Knowledge-Bases
Vol.W3,
ICOT, Tokyo, Japan, Dec. 1994
pp. 33-48

[WK99] J. Warmer and A. Kleppe
The Object Constraint Language -
Precise Modelling with UML
Addison-Wesley
1999

[WN98] Cognitive Science Laboratory, Princeton University
WordNet - a Lexical Database for English
http://www.cogsci.princeton.edu/~wn/
Cognitive Science Laboratory
Princeton University
221 Nassau St.
Princeton, NJ 08542
1998

[Wor96] J. Wortmann
Object-Oriented Analysis for Advanced Flight Data Management
Technischer Bericht 96-43
Technische Universität Berlin, Fachbereich Informatik, November1996.

[WW93] Y. Wand & R. Weber
On the ontological expressiveness of information system analysis and design Grammars
Journal of Information Systems, vol. 3, no. 4
1993
pp. 203 - 223

[WW95] Y. Wand & R. Weber
On the Deep Structure of Informstion Systems
Information Systems Journal, vol.5
1995
pp. 203 -223

Index

A

B

C

D - F

G -J

K - M

N

O

P

R

S

T

U - Z