A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools

Asmus Pandikow

Linköping 2002

ABSTRACT

In the 1980s, the evolution of engineering methods and techniques yielded the object-oriented approaches. Specifically, object orientation was established in software engineering, gradually relieving structured approaches. In other domains, e.g. systems engineering, object orientation is not well established. As a result, different domains employ different methods and techniques. This makes it difficult to exchange information between the domains, e.g. passing systems engineering information for further refinement to software engineering. This thesis presents a generic principle for bridging the gap between structured and object-oriented specification techniques. The principle enables interoperability of structured and object-oriented analysis and design tools through mutual information exchanges. Therefore, the concepts and elements of representative structured and object-oriented specification techniques are identified and analyzed. Then, a meta-model for each specification technique is created. From the meta-models, a common meta-model is synthesized. Finally, mappings between the meta-models and the common meta-model are created. Used in conjunction, the meta-models, the common meta-model and the mappings enable tool interoperability through transforming specification information under one meta-model via the common meta-model into a representation under another meta-model. Example transformations that illustrate the proposed principle using fragments of an aircraft's landing gear specification are provided. The work presented in this thesis is based on the achievements of the SEDRES (ESPRIT 20496), SEDEX (NUTEK IPII-98-06292) and SEDRES-2 (IST 11953) projects. The projects strove for integrating different systems engineering tools in the forthcoming ISO-10303-233 (AP-233) standard for systems engineering design data. This thesis is an extension to the SEDRES / SEDEX and AP-233 achievements. It specifically focuses on integrating structured and modern UML based object-oriented specification techniques which was only performed schematically in the SEDRES / SEDEX and AP-233 work.

ACKNOWLEDGEMENTS

My decision to return to academia in order to learn more about scientific ways of assessing, approaching and solving problems has turned out to be a good decision. During my time at the Real-Time Systems Laboratory at Linköpings universitet I had many opportunities to meet knowlegable people with interesting perspectives, to participate in a number of inspiring graduate courses and to improve my personal skills in many ways.

I am deeply indebted to my supervisor Dr. Anders Törne for his continuous encouragement, advice, mentoring and support. His technical and editorial advice was essential for carrying out the research work and for completing this dissertation. I am also grateful to my colleague and friend Erik Herzog for being an excellent and patient source of feedback and for introducing me into our common research area. Also, my thanks go to the former and present members of the Real-Time Systems Laboratory as well as to the many people from other laboratories for the vivid discussions we had and for creating such a nice working environment. I would like to thank Simin Nadjm-Tehrani for her feedback and support as leader of the Real-Time Systems Laboratory. Furthermore, I would also like to express my gratitude to our secretary Anne Moe for being such an attentive, foresighted and kind person. I gratefully acknowledge the hard work of the SEDRES project partners. In particular, I would like to thank Dr. Julian Johnson and Michael Giblin (both BAE Systems UK) for providing me with example data from the SEDRES projects and for their support.

Last, but not least, I would like to thank my wife Kartinka for her love, patience and encouragement during the past years. Her understanding and support provided the foundation for my work. I dedicate this work to her and our wonderful daughters Malin Muriel and Annika My.

This work has been supported in part by the European Commission through the SEDRES (ESPRIT 20496) and SEDRES-2 (IST 11953) projects as well as by the Swedish National Board for Industry and Technology Development (NUTEK) through the SEDEX (IPII-98-06292) project. Their support is gratefully acknowledged.

CONTENTS

1. Introduction 1

1.1 Background and Motivation 1

1.2 Research Questions and Main Contributions 3

1.3 Related Work 4

1.3.1 Adjacent Areas 4

1.3.2 Past Efforts on Integrating Structured and Object-Oriented Approaches 5

1.3.3 SEDRES, SEDEX, SEDRES-2 and AP-233 7

1.3.4 OMG SE DSIG 8

1.3.5 Thesis Delimitation 9

1.4 Approach and Thesis Overview 11

1.4.1 Approach Overview 11

1.4.2 Alternatives 14

1.4.3 Thesis Structure 14

1.4.4 Guideline 16

2. Comparing Structured and Object-Oriented Development 17

2.1 Historical Development 17

2.2 Process Level Comparison 23

2.2.1 Structured Perspective 23

2.2.2 Object-Oriented Perspective 25
2.2.3 Comparison 26
2.3 Analysis and Design - The Focus of Interest 29
2.4 Technique and Concept Level Comparison 30
2.4.1 Structured Perspective 30
2.4.2 Object-Oriented Perspective 32
2.4.3 Comparison 34
2.5 Comparison Conclusions 35

3. Extracting Concepts from Structured and Object-Oriented Techniques 37
3.1 Extracting Concepts from Structured Techniques 37
3.1.1 Selecting Representative Structured Techniques 37
3.1.2 Data-Flow Diagram (Yourdon and Constantine) . . . 39
3.1.3 Data Dictionary (Pressman) 42
3.1.4 Entity-Relationship Diagram (Martin) 44
3.1.5 State-Transition Diagram (Harel) 47
3.2 Extracting Concepts from Object-Oriented Techniques 50
3.2.1 Selecting Representative Object-Oriented Techniques . 50
3.2.2 Common Concepts 52
3.2.3 Static Structure Diagram 54
3.2.4 Use Case Diagram 57
3.2.5 Collaboration Diagram 59
3.2.6 Sequence Diagram 61
3.2.7 Statechart Diagram 63

4. Creating Meta-Models of Structured and Object-Oriented Techniques . . . 65

4.1 Meta-Modeling Framework . . . 65

4.1.1 STEP . . . 66

4.1.2 EXPRESS and EXPRESS-G (ISO 10303-11) Overview 67

4.1.3 EXPRESS-X (ISO 10303-14) Overview . . . 72

4.2 Common Types . . . 73

4.3 Meta-Models of Structured Techniques . . . 75

4.3.1 Data-Flow Diagram . . . 75

4.3.2 Data Dictionary . . . 77

4.3.3 Entity-Relationship Diagram . . . 79

4.3.4 State-Transition Diagram . . . 80

4.4 Meta-Models of Object-Oriented Techniques . . . 82

4.4.1 Common Object-Oriented Types . . . 82

4.4.2 Classifier . . . 84

4.4.3 Relationships . . . 86

4.4.4 Static Structure Diagram . . . 88

4.4.5 Use Case Diagram . . . 90

4.4.6 Collaboration and Sequence Diagram . . . 92

4.4.7 Statechart Diagram . . . 95

5. Creating the Common Meta-Model . . . 97

5.1 Integration Principles . . . 97

5.2 General Concepts of the Common Meta-Model . . . 102

5.2.1 Types . . . 102

5.2.2 Views . . . 104

5.3 Data Aspect . . . 107

5.3.1 Classifications 107

5.3.2 Relationships 109

5.4 Functional Aspect 113

5.4.1 Functional Extension to the Classification Concept . . 113

5.4.2 Functional Extension to the Relationship Concepts . . 115

5.5 Behavioral Aspect 116

5.5.1 Behavioral Extension to the Classification Concept . . 116

5.5.2 Behavioral Extension to the Relationship Concepts . . 118

6. Integrating the Specification Techniques through the Common Meta-Model 121

6.1 Integration Principle 121

6.2 EXPRESS-X (ISO 10303-14) 123

6.3 EXPRESS-X Visualization 125

6.4 Common Mappings 129

6.5 Mappings from Structured Techniques 131

6.5.1 From Data-Flow Diagrams 131

6.5.2 From Data Dictionaries 133

6.5.3 From Entity-Relationship Diagrams 135

6.5.4 From State-Transition Diagrams 137

6.5.5 Summary 139

6.6 Mappings to Structured Techniques 140

6.6.1 To Data-Flow Diagrams 140

6.6.2 To Data Dictionaries 142

6.6.3 To Entity-Relationship Diagrams 145

6.6.4 To State-Transition Diagrams 147

6.6.5 Summary 149

6.7 Mappings from Object-Oriented Techniques 151
6.7.1 From Common Object-Oriented Elements 151
6.7.2 From Static Structure Diagrams 153
6.7.3 From Use Case Diagrams 154
6.7.4 From Collaboration and Sequence Diagrams 155
6.7.5 From Statechart Diagrams 157
6.7.6 Summary 160
6.8 Mappings to Object-Oriented Techniques 162
6.8.1 To Common Object-Oriented Elements 162
6.8.2 To Static Structure Diagrams 164
6.8.3 To Use Case Diagrams 165
6.8.4 To Collaboration and Sequence Diagrams 168
6.8.5 To Statechart Diagrams 169
6.8.6 Summary 172
6.9 Discussion 174

7. Application Example 175
7.1 Scenario 175
7.2 Example Outline 178
7.3 Example Specification Data Exchanges 180
7.3.1 Landing Gear Control System 180
7.3.2 Nose Gear Manager 190
7.4 Summary and Evaluation 194

8. Conclusions 195
8.1 Summary and Conclusions 195
8.2 Future Directions 197

Appendix 199

A. Terminology 201

B. Meta-Model Mappings 205

B.1 Data-Flow Diagram to Common Meta-Model 206

B.2 Common Meta-Model to Static Structure Diagram 211

B.3 State-Transition Diagram to Common Meta-Model 215

B.4 Common Meta-Model to Statechart Diagram 218

LIST OF FIGURES

1.1 Overview of the approach 12

1.2 Use Scenario 13

1.3 Tool integration options 15

2.1 Historical development of structured and object-oriented methods and techniques 19

3.1 Elements of a data-flow diagram 39

3.2 Example of a data-flow diagram 41

3.3 Example of a data dictionary 43

3.4 Elements of an entity-relationship diagram 44

3.5 Example of an entity-relationship diagram 46

3.6 Elements of a state-transition diagram 47

3.7 Example of a state-transition diagram 49

3.8 Elements of a static structure diagram 54

3.9 Example of a static structure diagram 56

3.10 Elements of a use case diagram 57

3.11 Example of a use case diagram 58

3.12 Elements of a collaboration diagram 59

3.13 Example of a collaboration diagram 60

3.14 Elements of a sequence diagram 61

3.15 Example of a sequence diagram 62
3.16 Example of a statechart diagram (adapted from [Gro01]) . . . 64

4.1 Important elements of the EXPRESS-G notation 68
4.2 Extensions to the EXPRESS-G notation 69
4.3 EXPRESS-G example 70
4.4 Example of an EXPRESS schema 71
4.5 EXPRESS-G example instantiation 72
4.6 Meta-model of common basic type 73
4.7 Meta-model of the common multiplicity type 74
4.8 Meta-model of data-flow diagrams 76
4.9 Meta-model of data dictionaries 78
4.10 Meta-model of entity-relationship diagrams 80
4.11 Meta-model of state-transition diagrams 81
4.12 Meta-model of common object-oriented types 83
4.13 Meta-model of classifiers 85
4.14 Meta-model of associations 87
4.15 Meta-model of generalizations 87
4.16 Meta-model of static structure diagrams 89
4.17 Meta-model of use case diagrams 91
4.18 Meta-model of collaboration and sequence diagrams 94
4.19 Meta-model of statechart diagrams 96

5.1 Cases of semantic matching 98
5.2 Common meta-model of basic types 102
5.3 Common meta-model of specific types 103
5.4 Common meta-model of cardinality 104
5.5 Common meta-model of views 105

5.6 Common meta-model of classifications (data aspect only) . . 108
5.7 Common meta-model of relationships (data aspect only) . . . 110
5.8 Common meta-model of association classifications 112
5.9 Common meta-model of classifications (data and functional aspect only) 114
5.10 Common meta-model of classifications 117
5.11 Common meta-model of relationships 119
5.12 Common meta-model of events 120

6.1 Integration Principle 122
6.2 Example of an EXPRESS-X mapping 124
6.3 EXPRESS-X visualization elements 125
6.4 EXPRESS-X visualization examples 127
6.5 Mapping overview: Data-flow diagram meta-model to common meta-model 132
6.6 Mapping overview: Data dictionary meta-model to common meta-model 134
6.7 Mapping overview: Entity-relationship diagram meta-model to common meta-model 136
6.8 Mapping overview: State-transition diagram meta-model to common meta-model 137
6.9 Mapping overview: Common meta-model to data-flow diagram meta-model 141
6.10 Mapping overview: Common meta-model to data dictionary meta-model 143
6.11 Mapping overview: Common meta-model to entity-relationship diagram meta-model 146
6.12 Mapping overview: Common meta-model to state-transition diagram meta-model 148

6.13 Mapping overview: Object-oriented classifier to common meta-model . . . 151

6.14 Mapping overview: Object-oriented relationships to common meta-model . . . 152

6.15 Mapping overview: Static structure diagram meta-model to common meta-model . . . 153

6.16 Mapping overview: Use case diagram meta-model to common meta-model . . . 154

6.17 Mapping overview: Collaboration and sequence diagram meta-model to common meta-model . . . 156

6.18 Mapping overview: Statechart diagram meta-model to common meta-model . . . 158

6.19 Mapping overview: Common meta-model to object-oriented classifier meta-model . . . 162

6.20 Mapping overview: Common meta-model to object-oriented relationships meta-model . . . 163

6.21 Mapping overview: Common meta-model to static structure diagram meta-model . . . 164

6.22 Mapping overview: Common meta-model to use case diagram meta-model . . . 166

6.23 Mapping overview: Common meta-model to collaboration and sequence diagram meta-model . . . 168

6.24 Mapping overview: Common meta-model to statechart diagram meta-model . . . 170

7.1 Data exchange scenario . . . 176

7.2 Schematic overview of the landing gear system (adopted from [NTS99]) . . . 178

7.3 Landing gear control system: Original Statemate representation 181

7.4 Landing gear control system: Fragment of the Statemate native file representation . . . 183

7.5 Landing gear control system: Fragment of the data-flow diagram meta-model instantiation . . . 184
7.6 Landing gear control system: Fragment of the common meta-model instantiation . . . 185
7.7 Landing gear control system: Fragment of the static structure diagram meta-model instantiation . . . 187
7.8 Landing gear control system: Fragment of the Rose native file representation . . . 188
7.9 Landing gear control system: Fragment of the Rose representation . . . 189
7.10 Nose gear manager: Statemate representation . . . 190
7.11 Nose gear manager control: Original Statemate representation 191
7.12 Nose gear manager control: Rose representation, top-level statechart . . . 192
7.13 Nose gear manager control: Registration statechart, Rose representation . . . 193
7.14 Nose gear manager control: Moding statechart, Rose representation . . . 193

LIST OF TABLES

2.1 Activities in structured and object-oriented life cycle phases . 28
2.2 Analysis and design in different life cycle models 30
2.3 Structured specification techniques 31
2.4 Object-oriented (UML) specification techniques 33

3.1 Selected structured specification techniques 38
3.2 Elements of a data dictionary 42
3.3 Selected UML specification techniques 51

5.1 Specification techniques and their modeling aspects 100

1. INTRODUCTION

This chapter presents the background and motivation for the thesis, posts the tackled research questions and describes the delimitation of this thesis with respect to related work. Furthermore, it outlines the selected approach as well as the structure of the thesis.

1.1 Background and Motivation

The advent of programmable computers has unmistakably revolutionized system development. Programmable components of a system allowed for flexible and reusable designs, and software allowed for implementing solutions that previously were unimaginable with purely mechanical and electrical means. Throughout recent decades, software has become a substantial part of systems, and system functionality is in general increasingly realized through software. In many cases, software even conquers previously purely mechanical or electrical domains, as for example in the fly-by-wire principle of modern aircrafts. The development of system components in different engineering disciplines has not always been carried out in an integrated fashion, which led to the situation that different specification methods and techniques have emerged and have been established in different engineering disciplines.

A characteristic example are the object-oriented concepts, methods and techniques for engineering software that have emerged in the 1980s. In software engineering, object orientation has established during the 1990s, gradually relieving the previous structured approaches. However, object orientation is not yet well established in other engineering disciplines. The shift from structured to object-oriented approaches in software engineering has caused some concern about how to integrate structured and object-oriented approaches and how to exchange specifications between both. However, in software engineering alone, this problem has become less important over

time as object orientation solidified during the 1990s as the major technology for specifying software. Reusing previous structured specifications was difficult in software engineering, due to the rapid evolution of software specification methods and techniques. Reuse was anyhow in practice not a major issue, due to the relatively short life-cycles of software.

In the systems engineering domain, however, engineers are often engaged in long-life products with strong focus on reuse, e.g. in the aircraft and space industry. Systems engineers often continue to employ structured specification techniques due to a number of reasons. First, object orientation is relatively young, compared to structured techniques, and was in the 1990s (and by some engineers also still today) not considered to be sufficiently mature for engineering systems. This opinion was fed by the growing plethora of differing and incompatible methods and notations, lacking standards, and divergent semantics for otherwise homonymous concepts. Second, accessing and reusing previous specifications is of prime importance for industries in the systems engineering domain, e.g in the aircraft industry. However, it is still unclear how the structured legacy specifications can be integrated with or turned into object-oriented ones. Third, engineers in these industries have a broad background, skill and practice in structured approaches and often prefer those over the newer and different object-oriented ones.

In summary, due to the increasing importance of software in systems, and due to increasingly heterogeneous intra- and interorganizational projects that require integrated analysis and design of several engineering disciplines, it is necessary also to integrate structured and object-oriented methods and techniques.

In 1996, the SEDRES project (presented in Section 1.3) commenced, aiming at solving a similar problem, namely integrating different system specification tools used in the systems engineering domain. The SEDRES project was primarily focused on tools that implement structured engineering approaches, the integration of object-oriented ones was first included in the follow-up project SEDRES-2. Amongst many other achievements in the SEDRES-2 project, object-oriented concepts have been integrated with the (structured) concepts of the SEDRES information model. However, the integration has only been performed at a high level that primarily allows for tracing object-oriented elements in a specification history and to include them in a common configuration management. Data exchanges at the level of single concepts were not supported, i.e. meaningful data exchanges between structured and object-oriented specification tools were still not pos-

sible with the SEDRES-2 information model. Also, the forthcoming ISO 10303-233 standard (short: AP-233), will probably not support such data exchanges, as it is based on the SEDRES-2 information model.

The work presented in this thesis can be seen as extension of the work performed in both SEDRES projects, regarding the integration of structured and object-oriented specification tools and techniques. It aims at integrating structured and object-oriented concepts such that it is possible to exchange specification data between structured and object-oriented tools that are to be used collaboratively. Such a link between structured and object-oriented specification techniques would enable organizations to transform their structured specifications into object-oriented representations. In turn, this would allow for reusing structured specifications in an object-oriented environment, e.g. if an organization intends to switch from using structured approaches to using object-oriented ones.

1.2 Research Questions and Main Contributions

The work in this thesis tackles the problems presented above of finding mappings from structured approaches to modern object-oriented ones. The goal of this work is to provide a principle that allows for meaningful collaboration of structured and object-oriented specification tools, e.g. by supporting the transformation of systems engineering specifications to software engineering representations, or by allowing for the use of structured legacy specification data in object-oriented specification tools. The major research questions derived from this can be formulated as follows.

- Do structured and object-oriented development approaches have enough semantic overlap so that meaningful mappings between both can be created?
- How and at which level do these mappings need to be described so that the major structured and object-oriented concepts can be mutually mapped?
- How can these mappings be implemented so that the implementation serves as a means for specification data exchanges between structured and object-oriented specification tools?
- What are the problems in creating the mappings, what is easy and what is difficult to map?

In anticipation of discussing and answering these questions in the remainder of this thesis, the main contributions of the thesis can be summarized as follows.

- Description of a generic principle of integrating structured and object-oriented specification techniques through a common meta-model, using standardized mechanisms.
- A common meta-model of representative structured and object-oriented specification techniques that serves as a data model for a central data repository between structured and object-oriented tools.
- An illustration of how the presented principle is applied in a real-world scenario.

As a by-product, a visual notation for graphically sketching mappings under the framework used in this thesis (ISO 10303, Standard for the Exchange of Product Model Data, STEP) is introduced that also can be applied to other work within the same framework.

In summary, the goal of this thesis is to present an approach that enables interoperability through specification data exchanges between structured or object-oriented tools.

1.3 Related Work

This section describes work that is related to this thesis from different levels of abstraction. First, the areas of research are presented that are either touched on in this thesis or closely related to it. Second, the approaches of the past that can be found in the literature and that specifically tackle structured and object-oriented specification techniques are analyzed. Third, the work is described that directly serves as a foundation for this thesis. Finally, a delimitation of the thesis with respect to related work is provided.

1.3.1 Adjacent Areas

From a broader perspective, the work presented in this thesis addresses a number of different research areas. Various aspects of systems engineering, tool integration, meta-modeling, software and system specification techniques, and object orientation influence the thesis as follows. The need

within the systems engineering community to integrate their tools in order to be able to conduct joint projects has yielded the SEDRES projects and the AP-233 standardization work (see Subsection 1.3.3 below). These activities were also the starting points for this thesis. The insights from meta-modeling in different domains have been incorporated in the ISO 10303 (STEP) standard that has been employed for the SEDRES and AP-233 information models, and also for this thesis. The area of software and system specification techniques is the focus of this thesis, including object-oriented approaches.

Similar problems to the problem tackled in this thesis can be found in the emerging research area around the Semantic Web [WWW02]. Berners-Lee outlines the Semantic Web as "an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." ([BLHL01]). Possible connections between the work presented in this thesis and the Semantic Web are discussed under "future directions" in Section 8.2.

Another closely related area is the area of ontology integration, which also is within the central scope of the Semantic Web. Most of the work in this area originates from database schema integration and discusses different aspects of mappings between two ontologies. Examples of publications in this area are the general analysis of ontology mismatches in [VJBCS97] or the discussion about high costs for the creation of mappings in [Hei95].

1.3.2 Past Efforts on Integrating Structured and Object-Oriented Approaches

Numerous efforts have been undertaken in examining and integrating structured and object-oriented methods and techniques. The bulk of this work has been performed during the late 1980s, when object orientation just emerged. At this time, little experience of object-oriented approaches was available, nor was it clear how object orientation would develop. It was also unclear which methods and techniques would prevail and which ones would fall into oblivion. The related work of this period can be roughly classified into the following categories.

- Concurrently using structured and object-oriented methods and techniques.
- Combining structured and object-oriented methodologies.

- Extending structured methods to be able to integrate object-oriented principles.
- Comparison of structured and object-oriented methods and techniques.

The first category, using structured and object-oriented techniques concurrently, represents an integration on a high level and from a life cycle and phase perspective. The publications proposing such approaches, such as [Ala88], [BD89], [Gra87], [Gra88], [Kha89], or [Luk91], disregard lower level integration. They do not describe the integration at the level of single items, which would allow for information exchanges between structured and object-oriented specification tools. However, the approaches in this category merely bind object-oriented techniques to existing structured processes.

Proposals of the second category combine structured and object-oriented techniques, e.g. on programming level, as proposed in [BBM90]. They create a new technique, weakening some of the aspects of both structured and object-oriented techniques. For example, the approach taken by Pendley in [Pen89] is based on the anyway close relationship between information engineering and object-oriented characteristics; however, without explicitly tackling the integration of structured and object-oriented techniques.

The proposals of the third category, e.g. [War89], [HS91], [HSC91], or [Li91], extend the traditional structured approaches and make room for object-oriented principles, such as encapsulation or inheritance. However, an integration at the level of single items in order to allow actual information exchanges between tools is not provided. The integration is merely performed at the level of abstract concepts.

Furthermore, in the fourth category, a number of comparisons of different quality have been performed. Comparisons between structured and object-oriented techniques include [BDR84], [Kel87], [FK92], and [Sha94]. Comparisons of only object-oriented techniques can be found in [Gra91], [Shl92], or [SC93]. The comparisons in this category do not explicitly aim at integrating structured and object-oriented analysis and design techniques, but allow for identifying important techniques and concepts of both domains and provide a good basis for conducting further research on this subject.

It is noticeable that the research activity in the area of integrating structured and object-oriented software engineering methodologies has declined since the early 1990s. This is mainly based on the fact that, at that time, object orientation was considered not to be mature enough. Additionally, it was unclear how it would develop. In the meantime, object orientation

has become the prevailing approach to software engineering and within the software domain the need for integration with and access to "outdated" structured approaches is small, as explained above. However, in other domains, e.g. systems engineering, the access to structured specifications, and therewith the integration of structured and object-oriented approaches, is still an important issue. Comparing the conditions for such an integration with the situation in the 1990s, where a plethora of different object-oriented notations and techniques were in use, the picture is a lot clearer today. With the adoption of the Unified Modeling Language (UML) as standard by the OMG, the UML gained broad acceptance and became the prevailing notation for object-oriented software specifications. The UML can today be considered to be the main reference in the area of object-oriented notations.

1.3.3 SEDRES, SEDEX, SEDRES-2 and AP-233

In 1996, the SEDRES project (Systems Engineering Design Representation and Exchange Standardization, ESPRIT project 20496, 1996 – 1999, see also [SED99]) commenced. SEDRES arose from the need of several European companies from the aerospace industry to integrate their specification tools and to exchange design specifications across different companies and tools to enable collaboration in joint projects. SEDRES was aiming at creating an international standard for the exchange of design data and used the STEP standardization framework (Standard for the Exchange of Product Model Data [ISO94]) by ISO (International Standardization Organization [ISO02]) for describing its information models. The SEDRES project ended in March 1999, having created an information model that accommodates and integrates the semantics of the concepts of a number of different specification tools, such as CoRE (BAE Systems internal tool), LABSYS (Aerospatiale Matra internal tool), Statemate (by i-Logix), MATRIXx (by ISI), StP (Software through Pictures, by Aonix), TeamWork (by Cayenne), and others (see [SED99] for more details). The SEDRES project also produced example implementations using the information model, showing actual data exchanges between different tools.

During 1999, the SEDEX project (project number IPII-98-06292, funded by the Swedish National Board for Industry and Technology Development, NUTEK) continued the SEDRES work. It acted as intermediary project between SEDRES and SEDRES-2 and was mainly focused on improving details of the SEDRES information model and evaluating integration opportunities

for object-oriented concepts.

In 2000, the follow-up project on SEDRES, SEDRES-2 (IST project 11953, 2000 – 2001, see also [SED02] and [JH01]), commenced. Amongst other goals, SEDRES-2 also aimed at evaluating and including object-oriented concepts in its information model, as described in [PHT00] and [PT01c]. As briefly mentioned in Section 1.1, the final SEDRES-2 information model incorporates object-oriented concepts; however, at a level that primarily allows the tracing of object-oriented concepts from a version and configuration management perspective. The support of the SEDRES-2 information model for data exchanges between object-oriented and structured specification tools at the level of single concepts, i.e. with mutual "understanding" of the respective other concepts, is very limited.

Already at the outset, the SEDRES projects strove for standardizing the information model as an international standard within the STEP standardization framework ISO 10303. In STEP, such information models are integrated as protocols between applications, and the SEDRES information models served as the foundation for application protocol 233 of STEP (ISO 10303-233 or short: AP-233). During the SEDRES projects, and especially during SEDRES-2, different versions of the SEDRES information model have been made available to contributors other than the SEDRES participants, e.g. INCOSE and NASA, in order to conduct additional reviews and obtain further feedback on the proposed information model. The different versions of the information model have been published as working drafts of AP-233. The final version was working draft 5, which was also the basis for the Publicly Available Specification (PAS) 20542, published by ISO under [ISO01] (a PAS can be viewed as intermediary step towards a standard in the form of an AP).

Since the ending of SEDRES-2 in October 2001, the direction of the further development of the AP-233 working draft towards an actual part of ISO 10303 has been taken over by the ISO AP-233 working group. The future shape of AP-233 is currently unclear, particularly with regard to the degree and level of integrating object-oriented and structured concepts.

1.3.4 OMG SE DSIG

Also initiated from within the systems engineering community, the Systems Engineering Domain Special Interest Group (SE DSIG [DSI02]) was set up

in 2001 at the Object Management Group (OMG [Gro02]). The OMG also hosts the development of the object-oriented Unified Modeling Language notation (UML, see for example version 1.4 at [Gro01]). The goal of the SE DSIG is to develop proposals for future extensions and modifications of the UML that allow for a specialized use of the UML for systems engineering, i.e. outside the original domain of the UML. The SE DSIG tries to identify issues and constructs that are currently not supported by the UML but necessary to be able to employ the UML for specifying other than pure software systems. Therefore, the SE DSIG also examines a number of previously published (mainly proprietary) UML extensions and adaptations for being integrated in the SE DSIG work, e.g. the application of the UML to systems engineering described by Holt [Hol01], the UML real-time extensions by Axelsson [Axe00], or the more general work on applying object orientation to systems rather than only to software by Dori [Dor02].

Compared with the approach taken in SEDRES-2, the approach by the SE DSIG can be seen as approaching the problem from the opposite side. The SE DSIG tailors an object-oriented specification standard (the UML) for the use in systems engineering, whereas SEDRES-2 tried to integrate object-oriented concepts (based on the UML) in a systems engineering standard (the forthcoming AP-233). Another approach is to create the two standards for their domains, i.e. the systems engineering standard AP-233 within ISO and the software engineering UML profile within the OMG, and then interlink both through interfaces, as suggested in [PT01b]. However, although one focus of the SE DSIG is to "promote rigor in the transfer of information between disciplines and tools for developing systems" [DSI02], it is currently unclear at which level and to what extent the support of the SE DSIG work will prosper for actual data exchanges between structured and object-oriented specification tools.

1.3.5 Thesis Delimitation

The approaches presented above each contribute in one way or the other to the integration of structured and object-oriented concepts, at different levels of abstraction and to different extents. However, none of the approaches presented provides a generic solution to integrating structured and object-oriented specification tools.

The work in this thesis was mainly initiated by the work performed in SEDRES-2, regarding the integration of structured and object-oriented con-

cepts (see Subsection 1.3.3). However, it aims to integrate at a finer-grained level, namely at the level of single concepts, in order to allow for direct data exchanges between structured and object-oriented specification tools. The work is partly based on the early experience described in Subsection 1.3.2. However, it proposes a more general integration approach and, in contrast to the early approaches, bases its efforts on a single homogeneous and widely accepted object-oriented standard, the UML. Also, the results from adjacent areas such as schema mapping or meta-modeling can be applied, but are not sufficient alone, because they do not aim at integrating structured approaches and object orientation. Instead, they focus on either of them (e.g. the Meta-Object Facility MOF by the OMG [Gro00]) or only on a single aspect of system specification (e.g. schema mapping concentrating on the structural aspect only).

In summary, this thesis was initiated by the tool integration need in the SEDRES projects and was inspired by a number of contributions of related work. However, it specifically focuses on the integration of structured and (modern) object-oriented (UML based) specification tools, as suggested in [PT01a]. It may serve as input to the ongoing activities of creating the AP-233 systems engineering design data exchange standard (by the ISO AP-233 working group), the systems engineering extensions to the UML (by the OMG SE DSIG working group), or a combination of both as proposed in [PT01b]. Furthermore, it may also be taken as a basis for implementing the proposed approach with Semantic Web techniques, as proposed for the use of AP-233 together with models from the theory of domains in [AP02].

1.4 Approach and Thesis Overview

1.4.1 Approach Overview

As outlined in Section 1.2, one of the main goals of this thesis is to provide a generic principle for integrating structured and object-oriented specification techniques. The approach taken for this follows and extends the approach presented in [PT01a], which in turn is based on the approach of the SEDRES projects, as this turned out to be a sensible way of integrating different tools. This thesis suggests enabling tool interoperability through data exchanges based on a common meta-model of the specification techniques underlying the involved tools. The necessary elements for this approach are the meta-models of the tools (or of the specification techniques they implement), the common meta-model and mappings between the meta-models and the common meta-model. The individual steps towards creating the elements of the approach are illustrated in Figure 1.1 and can be summarized as follows.

1. Concept identification
2. Meta-modeling the specification techniques of the tools
3. Common meta-model synthesis
4. Mapping synthesis

First, representative structured and object-oriented specification techniques are selected and their constituting concepts are identified. For this thesis, generic specification techniques have been selected from the literature rather than actual implementations of specification techniques in tools, in order to achieve more generic results and to omit tool specific details that yield no additional insights.

Second, for each of the selected specification techniques, a meta-model is created that comprises all previously identified concepts of the specification technique. Hence, each meta-model should reflect and support the data that may accrue during modeling with the respective specification technique.

Third, a common meta-model is synthesized from the single meta-models of the selected specification techniques. The common meta-model comprises all concepts of the constituting meta-models of the involved specification techniques. Semantically similar or equivalent concepts are unified in integrated concepts of the common meta-model. The common meta-model can

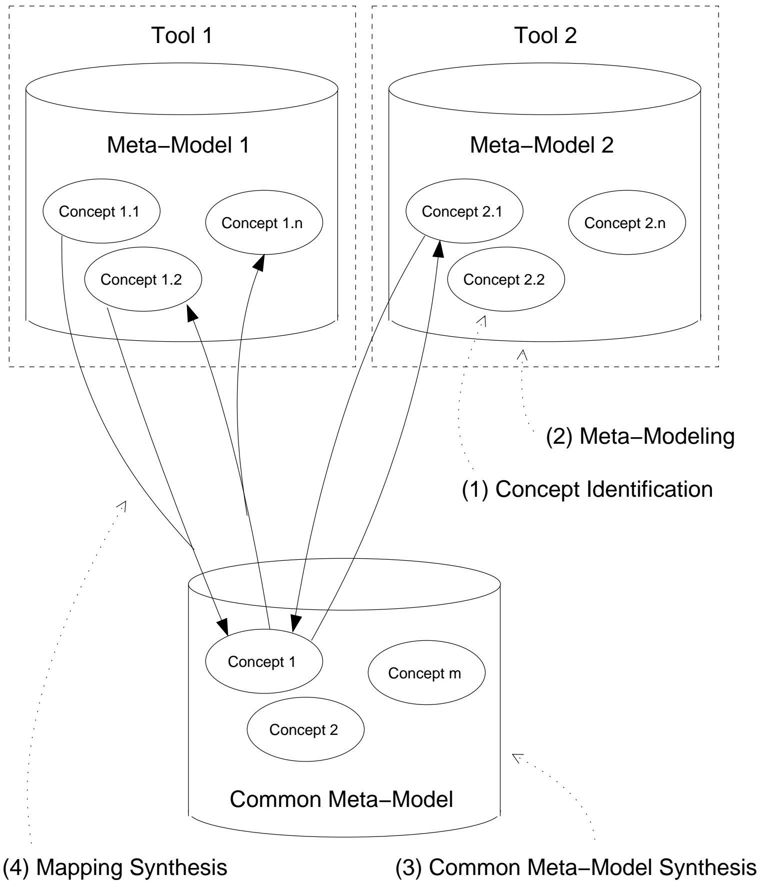

Fig. 1.1: Overview of the approach

be viewed as a template for a common repository, to be used as a central point of specification data storage and exchange between tools.

Finally, for each specification technique, a pair of mappings from and to the common meta-model is created, describing how data is transformed into the respective other representation. In more detail, mappings describe how one element of a meta-model of one of the specification techniques is mapped to a representation in the common meta-model and vice-versa. However, note that a transformation and a following re-transformation may not result in the same representation as the source representation due to the differences in semantical "richness" of concepts in the single meta-models compared to the common meta-model (see Figure 1.1).

With the meta-models, the common meta-model and the respective transformations available, these artifacts are used for enabling tool interoperability through data exchanges as illustrated in the use scenario in Figure 1.2.

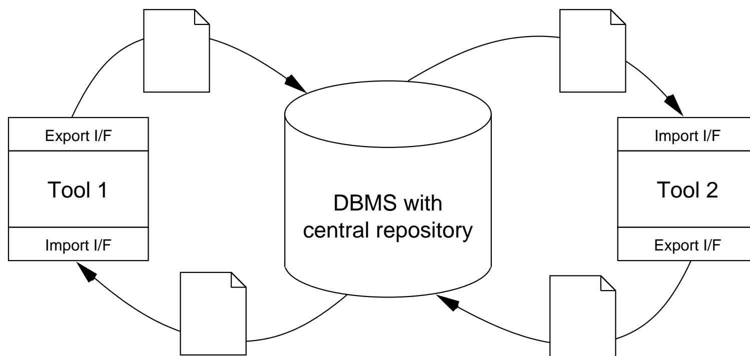

Fig. 1.2: Use Scenario

In this use scenario, the source tool (*Tool 1*) exports a specification through its export interface into the central repository. In the database management system (DBMS) that hosts the repository, the specification is transformed from its representation in *Tool 1* to a representation under the common meta-model. Then, the specification is transformed into a representation under the meta-model underlying *Tool 2* and imported by *Tool 2* through its import interface. For a reverse transformation of the specification, these steps are taken analogously in the reverse direction.

1.4.2 Alternatives

An alternative to the approach presented above would be to propose a new single specification technique on the basis of previously gained experience, combining concepts from the structured and object-oriented domain, as proposed by some of the related work discussed in Subsection 1.3.2. Advantages of this approach would be to obtain a common ontology with common semantics for specification artifacts, and a homogenous and linear basis for data exchange among specification tools. However, this is an unattainable solution due to the heterogeneous nature of systems engineering, where different people employ different dynamically evolving methods and taxonomies. Another weighty disadvantage of this approach is the fact that it disregards past specification efforts. Hence, it does not allow for the adoption of legacy specifications for further refinement, which is crucial as viewed against the background of this thesis and hence not a viable solution here. Additional transformation rules from legacy specification to representations according to the new technique would be a positive improvement of this alternative approach. However, it is still doubtable whether users and tool vendors would accept the endeavors of changing the specification methods and habits they are familiar with.

Using direct transformations between specification tools to exchange data is another alternative to the proposed architecture. However, using direct transformations between tools makes it more difficult to keep a global specification (that spans several tools) consistent, compared to keeping the specification in a central repository. Furthermore, the number of necessary mappings, and therefore the cost for creating these mappings, is exponential in the direct tool-to-tool approach ($n^2 - n$, n: number of involved tools, 2 transformation descriptions per tool, one for import and one for export). Using a central repository, this number is linear ($2 * n$), as illustrated in Figure 1.3.

1.4.3 Thesis Structure

Chapter 1 (this chapter) explains the background, states the motivation, and summarizes the main research issues behind this thesis. Furthermore, it provides an outline of related work and states and delimits the contributions of the thesis. Also, an outline of the chosen approach as well as a brief discussion on alternative approaches are presented.

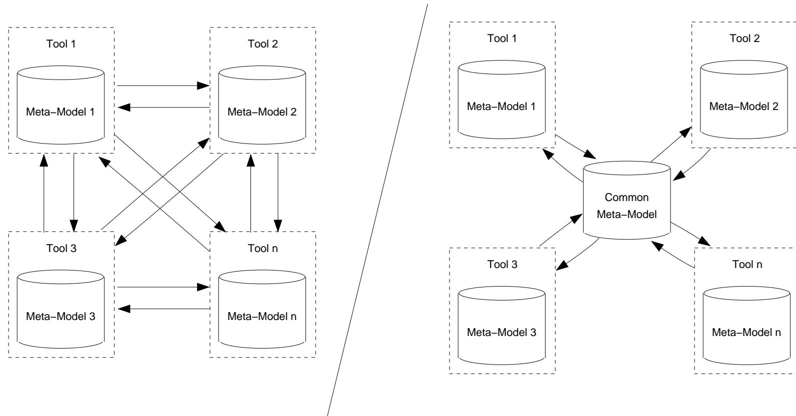

Fig. 1.3: Tool integration options

Chapter 2 reports on the historical evolvement of structured and object-oriented methods and techniques and contrasts both in order to illustrate their relationships. This forms the basis for integrating specification techniques from both domains.

Chapters 3 to 6 form the technical core of the thesis and are aligned with the proposed approach, described in Subsection 1.4.1, as follows.

Chapter 3 motivates the selection of structured and object-oriented specification techniques for integration purposes. Furthermore, the constituting concepts of the selected techniques are made explicit and the semantics of each concept is described.

Chapter 4 begins with a brief description of the meta-modeling framework that is used in the remainder of the chapter for meta-modeling the specification techniques described in Chapter 3.

Chapter 5 describes the common meta-model of the specification techniques presented in Chapter 3, synthesized from their meta-models as presented in Chapter 4. The description of the common meta-model is preceded by the principles that have been applied to integrate concepts of the single meta-models in the common meta-model.

Chapter 6 describes the generic principle of integrating the meta-models (Chapter 4) with the common meta-model (Chapter 5) through transformations between the models, as well as the actual mappings from and to the common meta-model.

Chapter 7 provides an example that shows how the proposed integration principle is implemented in a real-world scenario, based on specification data taken from the SEDRES projects.

Chapter 8 summarizes the thesis and draws conclusions from the work presented. Furthermore, possible extensions of the work and future directions are discussed.

The appendix provides supplementary information on the terminology used and the complete formal EXPRESS-X mappings (described in Chapter 6) that are used in the examples in Chapter 7.

1.4.4 Guideline

Due to differing semantics of terms used in the area of software and systems engineering, it is recommendable to first go through the terminology defined for this thesis in Appendix A. Furthermore, a knowledge of the background for the thesis, i.e. the origination of the SEDRES projects (see Subsection 1.3.3) in systems engineering tool integration problems, is necessary to understand the still existing need to access legacy specification efforts.

Readers who are familiar with basic structured specification techniques such as entity-relationship modeling, data dictionaries or data-flow diagramming, and the concepts of the Unified Modeling Language (UML) need only to skim Chapter 3. Readers who are familiar with the ISO 10303 (STEP) standardization framework, and especially the EXPRESS language family, can skip the introductory part of Chapter 4 and proceed directly to the meta-models of the specification techniques. The same applies to the mappings described in Chapter 6. However, the visualization notation for sketching EXPRESS-X mappings (see Subsection 6.3) should be understood to interpret the mapping overviews provided in Chapter 6. Chapter 5 should be read as a whole, as it presents and motivates the decisions made during the creation of the common meta-model. Readers who are interested in how the principle presented can actually be implemented should read Chapter 7 and the formal EXPRESS-X mappings from and to the common meta-model in Appendix B.

2. COMPARING STRUCTURED AND OBJECT-ORIENTED DEVELOPMENT

This chapter compares structured and object-oriented development. Therefore, the historical development of structured and object-oriented methods and techniques is presented. This is followed by a comparison of both approaches at different levels of abstraction, namely from a process and from a technique and concept perspective. The goal of this chapter is to show the grounding of object orientation in structured approaches as well as the commonalities of both as a basis for integration.

2.1 Historical Development

This section presents the historical evolution of structured and object-oriented specification methods and techniques from the 1970s until today. It emphasizes the shift from structured to object-oriented approaches.

In the past 30 years, new specification methods and techniques have emerged whenever the current practice did not suffice to handle the evolving requirements on software and its intrinsic complexity. Before the 1970s, software has mainly been developed individually and in ad hoc manner without following an underlying universal technique. At that time, approximately two thirds of the overall costs of the phases of a software's life cycle were spent on maintenance efforts, according to [Sch99]. Additionally, the constantly growing capabilities of computer hardware allowed for larger and more complex implementations of software. It became clear that the unstructured approaches were ineffective in producing fault-free software and also unsuitable for overcoming the enormous maintenance costs. In an effort towards

a more structured approach to software development, Dijkstra stated in his article [Dij69] that a major improvement would be to prevent errors instead of curing them, coining the term "structured programming". Later, in 1971, Wirth introduced "Program Development by Stepwise Refinement" in [Wir71], a method for structured programming that was based on the previous work by Dijkstra, Böhm and Jacopini [BJ66]. However, as structured programming still did not suffice in producing higher quality software at more controllable maintenance costs, the focus was put on finding errors before implementing the software. Parnas described in [Par72] principles of the work preceding the implementation. Stevens, Myers and Constantine developed an approach of "composite design", later published in [SMC74] as "structured design". In the following years, a number of techniques for structured design and structured programming emerged, e.g. in [War74], [Jac75], [Che76], or Yourdon's "Techniques of Program Structure and Design", published in [You75]. At the same time, attention was also directed at even earlier phases of software development, namely on the analysis preceding design. The term "structured analysis" was introduced by Ross [Ros77] and extended by DeMarco [DeM78], Gane and Sarson [GS79], and others.

As shown in Figure 2.1, the evolution of structured methods and techniques started at the late phases of the software life cycle, i.e. with implementation. Later the focus was centered on more abstract and earlier phases, i.e. design and analysis. In the late 1970s it became clear that structured methods could not live up to expectations. The size, and thus the intrinsic complexity and cost of software systems, grew exponentially, but the structured methods often did not scale up well and thus could not handle the demands on maintainability sufficiently well. The main reason for this was considered to be the one-sided focus of either data or the processing of data, granting the respective other aspect less attention. At that time, in the late 1970s, the first object-oriented approaches were developed, giving both data and data processing the same attention. The course that the evolution of object-oriented software engineering methods took was analogous to the structured methods. Again, it started at implementation level with the definition of programming languages such as Smalltalk80 in 1980 [GR83], followed by C++ in 1983, Eiffel in 1986, and others. In the 1990s, C++ became the dominating programming language, and since 1996 Java is increasingly used, whereas other object-oriented programming languages such as Smalltalk are losing importance. Besides object-oriented programming, also considerations about object-oriented design have emerged since the mid-1980s, such as in [Mey87], [WBW89], [CY91] or [Boo91]. With a

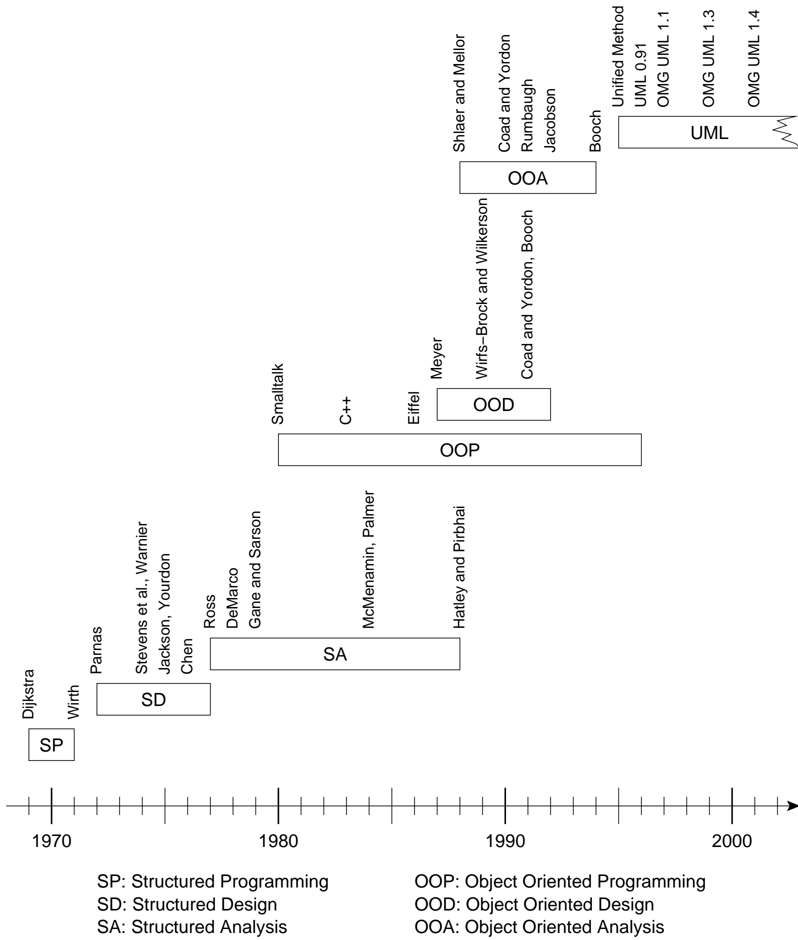

Fig. 2.1: Historical development of structured and object-oriented methods and techniques

short delay to object-oriented design, methods and techniques for object-oriented analysis (and a combination of both) were also published, such as [SM88], [CY90], the "Object Modeling Technique (OMT)" [RBP+91], "Object-Oriented Software Engineering (OOSE)" [JCJO92], or the "Booch Methodology" [Boo94b].

By 1993, about 40 different object-oriented methods and techniques had been published, as described in [Boo94a], but only a small number found broader acceptance and were used more widely. Among the most important (i.e.: widely used) were Booch's and Rumbaugh's methods that Booch and Rumbaugh unified at Rational Software Corporation in a single method, published as "Unified Method" [BR95]. Later, since 1995, Jacobson also joined Booch and Rumbaugh at Rational and features of Jacobson's OOSE were included in the work. Due to the fact that the unification of the three methods resulted in a new notation rather than in a new method, it has been published since then as the "Unified Modeling Language" (UML), starting with version 0.91 [BRJ96].

During the development of object-oriented methods and techniques, the Object Management Group (OMG) was set up in 1989. The mission of the OMG was defined as "setting vendor-neutral software standards, and enabling distributed, enterprise-wide interoperability" [Gro02]. The main focus of the OMG was put on modern software engineering, i.e. object-oriented and component-oriented software engineering. In 1997, the OMG adopted the UML and published it in its version 1.1 as standard. Since then, the OMG UML has been widely adopted and has become the most widely used notation for specifying object-oriented software systems.

The shift from structured to object-oriented techniques seemingly allowed engineers to tackle several of the problems that software engineering was facing. First, the immensely growing complexity of software products was challenging the traditional structured approaches. object-oriented principles allowed for a better handling of complexity. Furthermore, the focus on encapsulation facilitated distributed development, which is a common requirement for the developing complex systems. Second, encapsulation and abstraction through interfaces improved re-usability, and hence provided the means for reducing the development time and costs of future. Third, as opposed to traditional structured approaches that provide either a data-oriented or a processing-oriented view, object orientation allowed for specifying both views at the same time without giving preference to either of them.

However, despite the seemingly positive impact of object orientation on software engineering, it had not solved all problems that structured approaches could not solve. Instead, it introduced a number of new problems, as discussed in the following paragraphs.

The original main goal of creating more generic development approaches, namely the reduction of the cost of the maintenance phase, has not been achieved; neither by the structured approaches, nor by object orientation. In the mid 1970s, i.e. while the creation of the structured methodology, the average maintenance costs of software came to approximately two thirds of the overall life cycle costs, according to [Sch99]. After the introduction and establishment of object-oriented software engineering, i.e. in the mid 1990s, the contribution of the maintenance phase to the overall life cycle costs of software was still at approximately the same level, according to [Gra94]. Some authors even claimed the share of maintenance costs to be up to 80 percent of the overall life cycle costs, as for example described in [You96]. This fact, and the strong dependence of the maintenance efforts on the preceding analysis and design work, motivates further research on analysis and design, as, for example, provided by this thesis.

At its emergence, object orientation has been propagated as a new paradigm of software engineering, leading to a separation of methodologists into two camps, as described by Yourdon [You89]: Revolutionaries and synthesists. Revolutionaries, such as Booch, Coad and Yourdon, considered the shift from the traditional structured methodology to object orientation as a real paradigm shift, thus overruling the structured paradigm. Fichman and Kemerer summarize object orientation in [FK92] as "a radical change that renders conventional methodologies and ways of thinking about design obsolete". Garceau, Jancura and Kneiss [GJK93] even consider the structured approach with its separation between data and data processing inappropriate. The opposite camp, the synthesists, consider object orientation as a natural evolution in software engineering methodology, summarizing sound software engineering practices, compatible with traditional methods and techniques. These contradicting definitions can still be found today. However, in software engineering, the sharp distinction between both camps is not as apparent as in other domains. In systems engineering, for example, object orientation is not yet considered to be best practice. Publications in the systems engineering community on object-oriented systems engineering or even object-oriented system designs are still the exception. A common and widely accepted denominator for both positions of the camps has not yet been found, preventing the efforts of both domains from being joined.

In the last 15 years, object-oriented methods and techniques have emerged and become established, gradually relieving the traditional structured approaches used in software engineering. However, many of the software artifacts developed with the help of structured techniques are still in use. Also, many engineers still work today using structured techniques, mainly by necessity in order to seamlessly perform corrections and extensions of the legacy systems and partly because of their many years of experience and competence. However, a lack of harmonization efforts during the emergence of object orientation with the traditional structured approaches has established the differences between both in their respective methods, techniques and tools. Subsequently, the exchange of information and specifications between the tools of the different approaches is at least difficult. In most cases, data exchange is not automated, it can only be performed manually, inducing high costs and often also loss of information during the transformations because of mismatching concepts. This also implies that in an organization the transition from structured approaches to object-oriented ones is directly associated with difficulties in reusing previous specification work.

In summary, it can be seen from the historical development of specification methods and techniques that object-oriented and structured approaches have a number of similarities. Both have been introduced to handle the increasing complexity and intrinsic cost of software, and the evolution of object-oriented methods and techniques is analogous to the structured ones. The following sections illustrate this relatedness further. The strong foundation of object orientation in structured approaches is shown by unfolding and comparing the structured and object-oriented perspectives in terms of their intrinsic activities during product development.

2.2 Process Level Comparison

This section compares structured and object-oriented development at the level of development processes. Therefore, the major activities and sequential steps of structured and object-oriented development are described and compared. The goal of this section is to show that there are no major differences between structured and object-oriented development processes.

2.2.1 Structured Perspective

As described in Section 2.1, a plethora of approaches were created during the 1970s and 1980s proposing how structured development should be carried out. However, none of the approaches could establish itself as the one "universal structured development approach". During the rise and establishment of structured methods and techniques, the only widely applied life cycle model for structured development was the "waterfall model", first formulated by Royce [Roy89]. During this period, a number of variations of the waterfall model were published, but basically sharing the same core phases. The following paragraphs describe a more generic structured development process, following the development steps described by DeMarco in [DeM78], a widely employed variation of the waterfall model.

Problem Definition. The development starts with the agreement of users, customers and developers on the existence and the definition of the problem to be solved. The problem is usually stated in textual form, including the specifications of objectives and scope. Together with the subsequent feasibility study, this phase of the development process is also referred to as the requirements phase.

Feasibility Study. In this step, the feasibility of a problem solution is determined, resulting in a document describing the feasibility and difficulties of a possible solution, including an outline of cost and schedule.

Structured Analysis. In the analysis step, the artifacts and activities that are required to solve the identified problem are determined. The employed methods differ depending on whether the problem is approached from the processing-oriented or from the data-oriented perspective. Processing-oriented methods, such as Gane and Sarson's [GS79], start with the functional specification of the problem, employing techniques like data-flow diagrams, data dictionaries, or minispecs.

Data-oriented methods, such as entity-relationship modeling by Chen [Che76] focus on the modeling of data first; functions processing the data are considered in a second step. The result of the structured analysis step is a coarse specification of "what" is necessary and has to be done to solve the problem.

Preliminary Design. In this step, often integrated with the subsequent detailed design, alternative solutions are determined and contrasted, resulting in a "system specification" that describes the selected solution in general terms of "how" the problem is going to be solved.

Detailed Design. The detailed design elaborates and refines the input from the previous step, determining "how in detail" the selected solution will be implemented. As for the analysis, the activities in the design step are strongly dependent on the selected perspective, either processing-oriented or data-oriented, and the respective selected method. However, the design results in implementation specifications that describe architecture, function, data, and behavior of the planned solution in detail. Among the most widely used techniques for this task are detailed structure charts, detailed data-flow diagrams, detailed data dictionaries, minispecs, entity-relationship diagrams, finite-state machines (in the form of state-transition diagrams or decision matrices), Petri nets, and pseudo-code.

Implementation. Implementing means to code the designed program specifications in a programming language and to build an executable software system from it. Additional results of the implementation step are the implementation documentation and suggestions for the subsequent testing.

Test. In this step, the output from the implementation step is correlated with the specification in order to identify departures from original intentions, yielding test reports that describe the testing and its results.

Integration. The integration comprises acceptance testing by the stakeholders and subsequent installation and putting the system into operation.

Maintenance. Maintenance includes all reparation, improvement and updating activities that a system experiences after being handed over to the customer and before being removed from service. The major goal of maintenance is to guarantee system availability. The maintenance activities are usually textually documented in order to provide evolution traceability throughout the operational phase of the system.

Retirement. In this step, the system is removed from operation.

2.2.2 Object-Oriented Perspective

During the rise and establishment of object-oriented methods and techniques, the view on how software engineering in general should be performed also changed. During that time, a number of different life cycle models were developed, besides the waterfall model and independent of a structured or object-oriented perspective. Examples are the rapid prototyping model described in [Lan85], the incremental model (as for example the variation described in [Gil88]), Microsoft's synchronize-and-stabilize model [CS95], or the spiral model as described in [Boe88].

Additionally, processes specifically tailored to object-oriented engineering also emerged, such as the Fountain Model [HSE90], Objectory [JCJO92], or round-trip gestalt design [Boo94b]. These object-oriented life cycle models have an incremental and iterative character in common, inherited from the life cycle models described above.

Regardless of the underlying process, object-oriented software development embraces at least the following phases [Sch99]. Note that test and documentation creation are considered to be part of each phase. Hence, no stand-alone testing or documentation activities are mentioned in the following descriptions.

Requirements Phase. The stakeholder's requirements are elicited and the problem area is successively explored and refined in order to determine the desired functionality of the system. The output from this phase is usually a textual description of the stakeholder's requirements, but may be supplemented with high-level models, e.g. in the form of use cases or sketchy package and class diagrams.

Analysis Phase. The analysis phase, sometimes referred to as specification phase, comprises the analysis of the elicited requirements and their transformation into an unambiguous specification of what the system is intended to do. This is manifested with the help of use cases and package and class diagrams representing the static structure of the system, but still without describing the internal behavior.

Design Phase. The object-oriented design phase consists of refining the output of the analysis phase (use cases and coarse class diagrams). This

is done by extracting more detailed objects (and classes) from the use cases and by elaborating the class diagrams and adding behavior specifications in the form of state charts, activity diagrams, or interaction diagrams. Furthermore, for distributed systems, the distribution of the parts of the system among systems components and the run-time deployment are determined.

Implementation Phase. In the implementation phase, the specifications from the design phase are translated to an object-oriented programming language, followed by building the executable software from it.

Integration Phase. The integration phase comprises the test of the implemented components as a whole, followed by the acceptance test by the stakeholders. After successfully passing the integration phase, the system is put into service.

Maintenance Phase. Like the structured development process, the maintenance phase of object-oriented systems includes all reparation, improvement and update activities that the system experiences after being put into operation and before being removed from operation.

Retirement. In this final phase, the system is removed from service.

Unlike structured approaches, the object-oriented core concepts are kept throughout the complete development life cycle. Thus, the transitions between the development phases are smooth. This allows for frequent iterations of phases, as it is not necessary to transform the concepts between different phases. However, this also blurs the differences between the phases, as it allows for misusing the unchanging concepts for trial-and-error approaches to analysis and design.

2.2.3 Comparison

Revolutionaries (the camp described by Yourdon in [You89], who consider object orientation rendering structured approaches obsolete) often view the emergence of object orientation and the evolution of life cycle processes to be inseparably linked. However, life cycle models such as rapid prototyping, the incremental model, the synchronize-and-stabilize model, or the spiral model, concentrate on improving the development process and are equally well suited for structured as well as for object-oriented approaches. In opposition to the revolutionaries' view, the development of other life cycle models

than the popular waterfall model of the 1970s and 1980s, could rather be seen as natural evolution of software development processes, parallel to the emergence of object orientation. This evolution tried to cope with the rising demands on software production, such as complexity issues, maintainability and increased reuse of software components. Also, specific object-oriented life cycle models, such as the Fountain Model, Booch's model, or Objectory, build on the inevitable sequence of core development phases: requirements engineering, analysis, design, implementation, integration, maintenance and retirement.

Table 2.1 summarizes structured and object-oriented life cycles, based on a comparison by Schach [Sch99]. From this summary, it becomes clear that the core phases (or activities) of structured and object-oriented development processes are equal to each other. The major differences between the processes (or life cycle models) lie in their iterative and incremental character. Iterative processes explicitely allow for iterations of sub-sequences of phases. Incremental processes deliberately allow for (unfinished) intermediate states of the system and its specification. However, in summary, the greater parts of structured and object-oriented development are similar from a process perspective.

Tab. 2.1: Activities in structured and object-oriented life cycle phases

Phase	Structured	Object-Oriented
Requirements	Problem definition, feasibility study	Requirements elicitation
Analysis	Define, what is necessary to solve the problem	Analyze requirements, extract objects; determine, what the system is intended to do
Design	Preliminary design and detailed design, defining how the solution will be implemented	Add behavior to objects, refine use cases and static structure
Implementation	Translating design specifications into a programming language	Translating design specifications into a programming language
Integration	Acceptance testing, operation mode	Acceptance testing, operation mode
Maintenance	Repair, improve and update the system	Repair, improve and update the system
Retirement	Remove the system	Remove the system

2.3 Analysis and Design - The Focus of Interest

This section motivates why it is important to concentrate on analysis and design when integrating structured and object-oriented development approaches.

As described in Subsection 2.2.3, there are no major differences between the different development processes in terms of their core activities. This applies to structured as well as to object-oriented development. The processes merely differ in the strictness of their analysis and design phases, the degree of parallelism of these phases, and their iterative and incremental character. When contrasting structured and object-oriented software development, it is particularly interesting to focus on the divergent phases, namely analysis and design. Other phases (implementation, test, maintenance, etc.) are of less interest as their structured and object-oriented shaping are very similar and have less potential for contributing to the integration of structured and object-oriented approaches. Thus, analysis and design form the scope for the work in the remainder of this thesis.

Table 2.2 summarizes the different life cycle models that are considered in this thesis and their respective naming of the analysis and design phase. The models considered are the waterfall model [Roy89], the rapid prototyping model [Lan85], a variation of the incremental model [Gil88], the spiral model [Boe88], as well as the Fountain model [HSE90] and Objectory [JCJO92], representing object-oriented life cycle models. The Build-and-fix model representing the ad hoc approach to software development (see [Sch99]), has been excluded from consideration, as it is not a widely accepted approach. Also, the synchronize-and-stabilize model [CS95] has been excluded as it still can only be successfully employed at Microsoft.

Tab. 2.2: Analysis and design in different life cycle models

Life Cycle Model	Analysis	Design
Waterfall	Analysis and verification	Design and verification
Rapid Prototype	Analysis and verification	Design and verification
Incremental	Analysis and verification	Architectural design and verification, detailed design for each build
Spiral	Risk analysis, analysis and verification	Risk analysis, design and verification
Fountain	(object-oriented) analysis	(object-oriented) design
Objectory	Analysis	Design

2.4 Technique and Concept Level Comparison

2.4.1 Structured Perspective

As outlined in Section 2.1, structured approaches separate data and the processing of data. These views are supported by a number of different specification techniques that support either of both views. Table 2.3 presents some of the best-known structured specification techniques. However, the most prominent (most widely used) specification techniques are the data-flow diagram, defining the logical flow of data, the entity-relationship diagram, describing a data-oriented view, and the finite-state machines for describing behavior. The other techniques have similarities to the data-flow diagram, entity-relationship modeling, or finite-state machines, or can, for the greater part, be mapped to those.

Data-Flow Diagrams. A Data-flow diagram mainly illustrates functions and the functional hierarchy of a system. It also shows the flow of data between external entities (not part of the system) and the system, and among the system functions themselves. Several notations have emerged for data-flow diagrams, such as the one of Gane and Sarson [GS79], DeMarco's [DeM78], or Yourdon and Constantine's [YC79]. However, the concepts of these notations are very similar in terms of their semantics.

Tab. 2.3: Structured specification techniques

Life Cycle Phase	Structured Techniques
Analysis	Data-Flow Diagram
	Data Dictionary
	Entity-Relationship Modeling
	Pseudocode and mini-specs
	Functional-Flow Block Diagram
	and others
Design	Data-Flow Diagram
	Data Dictionary
	Entity-Relationship Modeling
	Finite-State Machine
	Petri-Net
	and others

Data Dictionary. A data dictionary is a central repository that contains definitions of a system's data items. The data dictionary is tightly coupled to other techniques, such as the data-flow diagram, entity-relationship diagram or behavioral specifications in pseudo-code. It provides definitions for "all data elements that are pertinent to the system" ([You89]). There are a number of mainly identical specifications of data dictionaries available, e.g. in [Pre00].

Entity-Relationship Modeling. Entity-relationship modeling is used to describe data items of the system and their relationships to other data items. Entity-relationship modeling was originally described in [Che76]; however, a number of variations have been published. One of the most widely used variations is the one by Martin [Mar88]. Also, a number of extensions to the original entity-relationship modeling have been proposed, such as extended entity-relationship modeling, which allows multi-valued attributes. However, these features are either not widely used or can also be expressed with the basic elements of entity-relationship modeling.

Pseudo-Code and Minispecs. Pseudo-code and minispecs are used to textually describe the behavior of a system function using a quasi natural language form. Pseudo-code and minispecs are used to provide a brief overview of the behavior, they do not formally define it. No commonly accepted variation exists for both.

Functional-Flow Block Diagram Function-flow block diagrams are used to illustrate the functions of a system and their sequential and hierarchical relationships. Functional-flow block diagrams are very similar to data-flow diagrams. They can be transformed into equivalent representations using data-flow and entity-relationship diagrams.

Finite-State Machine. Finite-state machines are used to model the behavior of a system in terms of possible states the system may be in and event-triggered transitions that cause the system to change its state. Among the most widely used notations for finite-state machines are decision matrices and the state-transition diagrams [Har87], [Kam87].

Petri-Net. Petri nets (originally published in [Pet62]) are also employed to describe the behavior of a system, using places (states the system may be in) and transitions (change of state) between places. Petri nets and finite-state machines have a similar syntax; however, they differ widely in their semantics. However, there exist mappings from Petri-nets to finite-state machines and vice-versa (see for example [CL98]).

2.4.2 Object-Oriented Perspective

Like for structured approaches, a number of techniques for object-oriented engineering have emerged over time, as already outlined in Section 2.1. Among them are class diagrams (such as the basically equivalent ones described in [Boo94b], [CY91], and [RBP$^+$91]), use case diagrams from [JCJO92], and interaction diagrams from [JCJO92] and [Boo94b]. Furthermore, a number of structured techniques have also been adopted and refined for object-oriented development, such as finite-state machines or entity-relationship modeling, which is reflected in the class diagrams.

Unlike structured development, object-oriented development is heading towards a commonly used specification notation that unifies a number of different object-oriented techniques. As described in Section 2.1, three of the most widely employed object-oriented methods, namely Booch's method, Rumbaugh's OMT, and Jacobson's OOSE, were joined at Rational, resulting in the Unified Modeling Language (UML). The UML was adopted by the Object Management Group (OMG) and published as OMG standard in 1997. It aspires to become the leading object-oriented notation and has in the meantime attained wide acceptance and use. Because of the exceptional position of the UML and the fact that it incorporates three of the

major object-oriented method, the UML in its version 1.4 [Gro01] forms the object-oriented reference for the work in this thesis. Hence, the object-oriented techniques and concepts presented in the remainder of this thesis are basically UML techniques and concepts.

Table 2.4 provides an overview of object-oriented (UML) specification techniques that are employed for the analysis and design activities in object-oriented development processes. The UML instantiates these techniques in a number of different diagram types, each explained in the following paragraphs.

Tab. 2.4: Object-oriented (UML) specification techniques

Life Cycle Phase	Generic Object-Oriented Techniques
Analysis	Use Case Diagram Static Structure Diagram (Package diagram, class diagram, object diagram)
Design	Use Case Diagram Static Structure Diagram (Package diagram, class diagram, object diagram) Behavior diagrams (Statechart diagram, activity diagram, sequence diagram, collaboration diagram)

Use Case Diagram. The UML use case diagram is based on the concepts from OOSE [JCJO92]. It is a widely used technique for illustrating coarse system functions in early specification phases.

Static Structure Diagram. The static structure diagram comprises the package diagram, the class diagrams, and the object diagram. The different diagram types are employed to specify the static structure of a system at different level of abstraction.

Statechart Diagram. The UML statechart diagram is based on Harel's statecharts [Har87]. It is a widely used technique for specifying the time-dependent behavior of a system.

Activity Diagram. In the UML specification [Gro01], the activity diagram is described as a derivative of the statechart diagram.

Sequence Diagram. The UML sequence diagram is based on a number of similar diagrams from other object-oriented methods, published as message trace diagrams, interaction diagrams, or event tracing diagrams.

Collaboration Diagram. In the UML specification [Gro01], the collaboration diagram is tightly coupled to the sequence diagram. However, it focuses on the structural aspect rather than the temporal. Otherwise, the collaboration diagram illustrates the same specification elements as the sequence diagram.

Implementation Diagrams. The UML component and deployment diagrams, summarized in the UML specification [Gro01] as implementation diagrams, are employed in implementation activities.

The use of inheritance and polymorphism is allowed in all of the above diagram types. Moreover, the indication of different levels of encapsulation can be made visible in detailed versions of the diagrams.

2.4.3 Comparison

The major characteristics of object orientation, namely encapsulation of data and data processing, inheritance and polymorphism, have no semantic equivalent in structured approaches. Equivalent structured representations can only be achieved by extending and constraining the use of structured concepts. Otherwise, a great proportion of object-oriented techniques and concepts build on previous structured achievements, e.g. finite-state machines or entity-relationship modeling. However, the differences between structured and object-oriented techniques and concepts are more obvious than the differences between their life cycles. This is due to the influence of the disadvantages of structured approaches on the creation of object-oriented techniques and concepts. object-oriented techniques and concepts consider data and the processing of data to be equally important, thus, providing integrated support for both. For example, the object-oriented concept of a class, having no direct semantic structured equivalent, comprises the operation and attribute concepts that are technically equivalent with the structured function (or procedure) and variable. Besides the technical overlap of structured and object-oriented concepts, the semantics of object-oriented concepts have partially been changed due to the use of the concept in a specified context and under a different development approach. For example, the object-oriented concept of an attribute has its structured equivalent in a variable. However, a variable also implies that it is a constituent of a class, whereas a variable is not automatically embedded in a similar way. Another important difference to structured approaches is the existence of "basic object-oriented concepts" [Bal96], namely object, class,

attribute, operation, message, inheritance, and polymorphism. These basic concepts are used throughout several phases of the development process, allowing for smooth transitions between the development phases, as opposed to necessary translations between concepts of different phases in structured approaches.

It can be concluded that object-oriented techniques and concepts have their main technical foundation in structured elements. However, object-oriented concepts are more integrated than their structured predecessors.

2.5 Comparison Conclusions

Object orientation can be seen as natural evolution of software engineering, as also described by Schach in [Sch99], page 204. It is a sum of sound software engineering practices with the focus on encapsulation (or information hiding), inheritance, polymorphism and consistence of concepts throughout the product life cycle. These object-oriented characteristics promise to improve the efficiency of software engineering as a whole. Object-oriented engineering is technically deeply grounded on achievements of the structured period of engineering. There is a great technical overlap between both. Object-oriented techniques include and/or refine existing structured concepts. However, object-oriented development takes a different approach, namely treating data and data processing at the same level of importance. Hence, object orientation requires a different way of thinking and thus partially also requires different approaches to analysis and design. In summary, object orientation can be considered to be another step in the evolution of development approaches rather than being a new paradigm.

3. EXTRACTING CONCEPTS FROM STRUCTURED AND OBJECT-ORIENTED TECHNIQUES

In this chapter, representatives of structured and object-oriented specification techniques are selected for later integration. Their elementary concepts are extracted and described. This chapter serves as a foundation for meta-modeling the selected techniques in Chapter 4, as well as for their common meta-model in Chapter 5 and the work on integration in Chapter 6.

3.1 Extracting Concepts from Structured Techniques

This section describes the structured software specification techniques that are considered to be most important for being integrated with object-oriented techniques. The various techniques are briefly described and unfolded into their elements, followed by a description of the syntax and semantics of each element.

3.1.1 Selecting Representative Structured Techniques

In order to represent different aspects of system specification, the following specification techniques, presented in Subsection 2.4.1 for specifying data, function and behavior, are considered for integration. The various data-flow diagram notations described in [GS79], [DeM78] and [YC79] are very similar. For integration purposes, Yourdon and Constantine's notation ([YC79]) is considered as representative of data-flow diagrams. There are also a number

of very similar descriptions for data dictionaries. For integration purposes, the description by Pressman [Pre00] is considered. Among the variations that have been published for entity-relationship modeling, the variation by Martin [Mar88] is considered for the integration because it is one of the most widely used. Less common extensions, such as extended entity-relationship modeling that allows multi-valued attributes, have been excluded from consideration. These features are either not widely used or can also be expressed with the basic elements of entity-relationship modeling. Pseudo-code and minispecs are omitted from the comparison. They neither have a commonly accepted variation, nor are they formally defined, i.e. they can be at most transferred as uninterpreted text between structured and object-oriented specifications, without carrying specification details. Functional-Flow Block Diagrams are not considered as they can be transformed into equivalent representations as data-flow and entity-relationship diagrams. As mentioned in Subsection 2.4.1, a number of different techniques for modeling finite-state machines have been published. For integration purposes, the state-transition diagram (a visual representation of finite-state machines) is considered in a simplified variation of Harel's statecharts [Har87]. Petri-nets are excluded from the integration considerations as there exist transformations for the greater part of Petri-nets into finite-state machine representations and vice-versa (see for example [CL98]).

Table 3.1 summarizes the selected structured specification techniques. The following subsections describe these specification techniques and their constituting elements in terms of syntax and semantics.

Tab. 3.1: Selected structured specification techniques

Structured Techniques	Selected Representative
Data-Flow Diagram	Yourdon and Constantine
Data Dictionary	Pressman
Entity-Relationship Diagram	Martin
Finite-State Machine	Harel's Statechart (State-Transition Diagram)

3.1.2 Data-Flow Diagram (Yourdon and Constantine)

Data-flow diagrams are employed to illustrate the system functions, the hierarchy of system functions, and the data channels (data-flows) between system functions. Furthermore, persistent data stores (usually representing a repository in the form of a file or a database) and external entities (entities that are considered to be outside the scope of the system) can be represented. However, the focus of data-flow diagrams is on a system's functionality, and how system functions interact with each other and external entities. A data-flow diagram consists of processes, data-flows, data stores, external entities, as well as control processes and control-flows. Their respective graphical notation is shown in Figure 3.1.

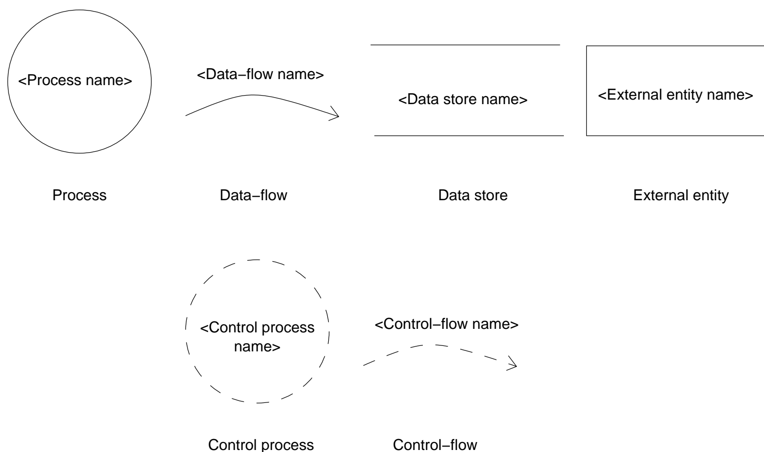

Fig. 3.1: Elements of a data-flow diagram

Process and Control Process. A process represents a function of the system, transforming incoming data into outgoing data. The incoming and outgoing data go through data-flows connected to the process. The internals of a process may be further refined through another data-flow diagram associated with the process. A process is depicted as a circle, labeled with the name of the process in the middle of the circle.

A special kind of process is a control process. A control process coordinates the execution sequence of processes in a data-flow diagram. It does not represent processing activities associated with the actual task of the data-flow diagram. A control process can only have control flows as outgoing flows for initiating (waking up) a process. Also, a

control process can only have control flows as incoming flows that indicate the finished execution of processes or extraordinary events that require special processing. The internal behavior of a control process is usually refined through a state-transition diagram, see 3.1.5. A control process is depicted as a dashed circle, labeled with the name of the control process in the middle of the circle.

Data-flow. A data-flow represents an unidirectional data channel between a process, an external entity, or a data store as source and a process, an external entity, or a data store as sink. A data-flow is depicted as a directed arc from the source to the sink, labeled with the name of the data that flows through the data channel. A special kind of data-flow is a control flow, as described as follows.

Control Flow. A control flow initiates a process. As opposed to "normal" data-flows, it only carries events, i.e. binary signals, and not typed values. A control flow is depicted as a directed dotted arc from the source to the sink, labeled with the name of the event that is associated with the control flow.

Data store. A data store represents a repository of data, e.g. a file in a file system or a table in a database. A data store is depicted as two equally long horizontal lines, one above the other, and labeled with the name of the data store between the lines.

External entity. An external entity represents an entity that is considered to be outside of the scope of the system. However, external entities are modeled as they are source or sink of the system's inputs respective outputs. An external entity is depicted as rectangle, labeled with the name of the external entity in the middle of the rectangle.

Figure 3.2 shows an example of a data-flow diagram illustrating a system that allows a user to input a login name and a password, verifies the password through reading the correct password from a file, passing the login name and the correctness of the password to the next process that reads the balance of an account if the password was correct, and transmits the account balance back to the user.

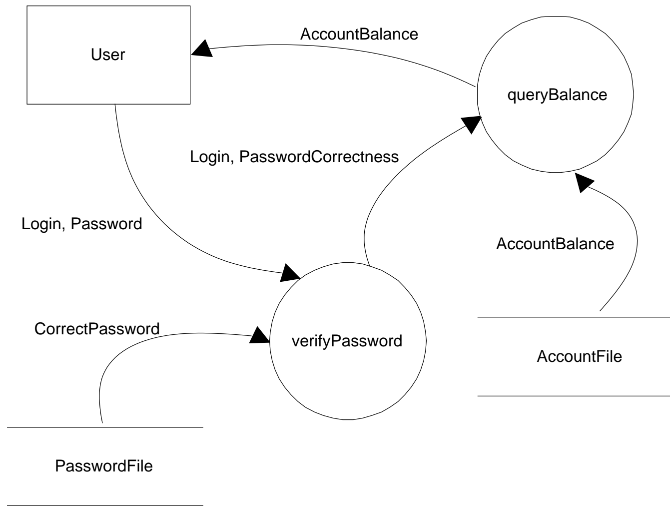

Fig. 3.2: Example of a data-flow diagram

3.1.3 Data Dictionary (Pressman)

A data dictionary is a textual listing that uses a quasi-formal grammar to describe the structure of data and control information. It is used as a central repository of definitions used by several other techniques, e.g. the structure of the data flowing through a data-flow in a data-flow diagram. A data dictionary consists of keywords assigned to aggregates of sequences, selections, repetitions, or optional elements. Additionally, comments can be used for a more detailed description of a keyword. The textual representation of the elements in a data dictionary is shown in Table 3.2.

Tab. 3.2: Elements of a data dictionary

<table>
<tr><th>Element</th><th>Notation</th><th>Remark</th></tr>
<tr><td>Keyword</td><td><Keyword></td><td></td></tr>
<tr><td>Aggregation</td><td>=</td><td></td></tr>
<tr><td>Sequence</td><td>... + ...</td><td></td></tr>
<tr><td>Selection</td><td>[... | ...]</td><td></td></tr>
<tr><td>Repetition</td><td>{ ... }n</td><td>n: number of repetitions</td></tr>
<tr><td>Option</td><td>(...)</td><td></td></tr>
<tr><td>Comment</td><td>* ... *</td><td></td></tr>
</table>

Keyword. A keyword represents a data item or a control information item. It is depicted by the name of the represented item, starting a new line in the data dictionary, followed by the aggregation sign and the declaration of its internal structure through a sequence.

Aggregation. An aggregation declares the composition of a data dictionary item. It is represented by the equals sign, preceded by a keyword, and followed by the declaration of the structure, usually in the form of a sequence.

Sequence. A sequence is a concatenation of data dictionary keywords, selections, repetitions, or optional parts. The concatenation symbol between the single elements of the sequence is represented with the plus sign.

Selection. A selection represents an number of choices of which one must be selected if the selection is instantiated. A selection is started with an opening square bracket, the options are separated by a pipe sign (a vertical line), and the selection is ended with a closing square bracket.

Repetition. A repetition represents a number of sequential repetitions of an expression. A repetition is depicted by curly brackets surrounding the expression to be repeated, followed by an optional integer number, defining the number of repetitions.

Option. An option represents an expression that may be omitted. An option is depicted by round brackets surrounding the optional expression.

Comment. A comment represents a comment in the data dictionary, which is not part of the definition and is thus not interpreted. Comments are solely used for improving readability of the data dictionary. A comment is depicted by two asterisks, surrounding the comment.

Figure 3.3 shows an example of a data dictionary, describing the structure of an address and its compartments. First, the basic types are built up from single characters. Then, the compartments are built up, based on the basic types "String" and "Number", followed by the definition of the address, namely the sequence of "Name", "Street", "ZipCode", and "City".

```
1  * A 'Character' is any character from A to Z, lower case or upper case *
2  Character = [ 'A .. Z', 'a .. z' ]
3  Digit = '0.. 9'
4  String = \{ Character \} * any number of characters *
5
6  Name = String
7  Street = String
8  ZipCode = \{ Digit \}5
9  City = String
10
11 Address = Name + Street + ZipCode + City
```

Fig. 3.3: Example of a data dictionary

3.1.4 Entity-Relationship Diagram (Martin)

An entity-relationship diagram is employed to describe data items of a system and the structural relationships among data items. An entity-relationship diagram can be viewed as representing the contents of a system's memory. An entity-relationship diagram solely illustrates the data aspect, without describing relationships to the system's functions. It consists of entities, attributes of entities, relationships among entities, associative entity indicators, and supertype / subtype indicators. Their respective graphical notation is shown in Figure 3.4.

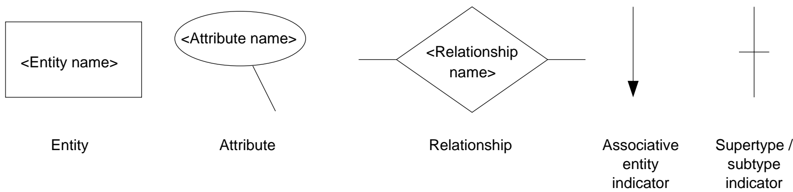

Fig. 3.4: Elements of an entity-relationship diagram

Entity. An entity represents a collection of instances (material and non-material) of the same kind, each being uniquely identifiable and necessary for the system under specification. Entities may optionally be described by one or more attributes. An entity is depicted by a rectangle with the name of the entity inside the rectangle.

Attribute. An attribute describes an instance of an entity in more detail. An attribute applies to all instances of the respective entity, i.e. each instance holds a value for the attribute. An attribute is depicted by an ellipse with the attribute's name inside, which is connected by a line with the associated entity representation.

Relationship. A relationship represents a set of connections of the same kind between entities. Relationships are connections that need to be remembered by the system under specification, and are independent of other relationships, i.e. they cannot be derived. A relationship is depicted by a diamond with the name of the relationship inside. The diamond is connected through lines from its corners to the entity representations that are involved in the relationship.

Associative entity indicator. An associative entity indicator is used if additional data about a relationship needs to be stored. It connects an

entity with a relationship. The entity is solely used for storing data associated with the relationship, i.e. the entity is not involved in any other relationships. The associative entity indicator is depicted by an arrow, starting at a corner of the relationship diamond symbol, pointing at the associated entity representation.

Supertype / subtype indicator. A supertype / subtype indicator represents a relationship between a more general entity (the supertype) and a more specific entity (the subtype). The subtype is fully consistent with the supertype, it inherits all of its relationships and attributes and may have additional relationships and attributes. A supertype / subtype indicator is depicted by a crossing bar through the connecting line from the supertype entity to the unnamed relationship representation, which in turn is connected to the subtype entity.

Figure 3.5 shows an entity-relationship diagram example that illustrates the data structure and relationships of an employee who is employed by a company. The *Employee* is a specialization of *Person*, indicated by the supertype / subtype indicator between both entities. Thus, *Employee* inherits the *Name* attribute of *Person*. The associative entity *Job* is used for storing more details of the relationship *isEmployeeAt*, namely the *Salary*.

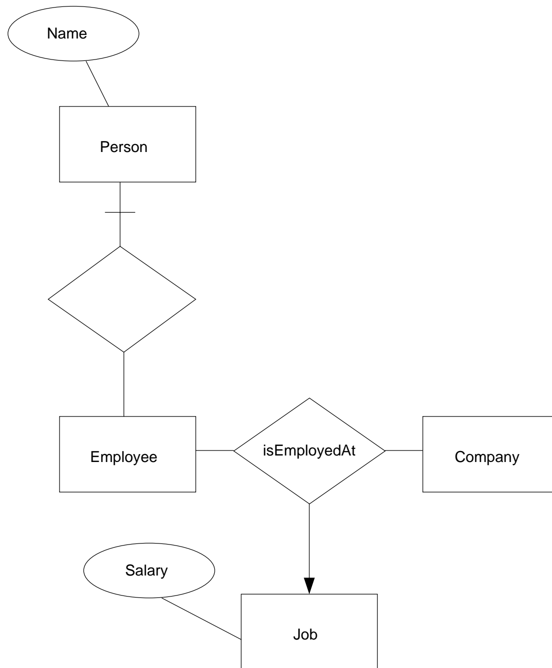

Fig. 3.5: Example of an entity-relationship diagram

3.1.5 State-Transition Diagram (Harel)

A state-transition diagram is a graphical representation of a finite-state machine. It is used to design the timing-dependent behavior of a system or parts of the system. A state-transition diagram illustrates the possible states of the system and how the system's state can change. It consists of a number of different states and labeled transitions between the states. The system (or part of the system) can only be in one state at a time if not stated otherwise, e.g. through parallel (or-) states. The graphical notation of the elements of a state-transition diagram is shown in Figure 3.6.

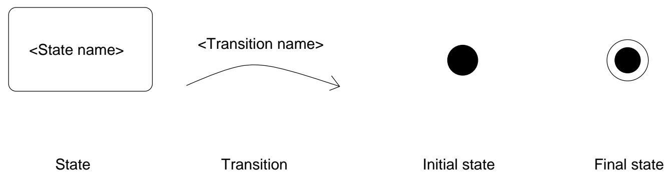

Fig. 3.6: Elements of a state-transition diagram

State. A state in a state-transition diagram represents a possible state of a system, i.e. a set of circumstances and attribute values of the system under consideration at a given time, waiting for events to change the state. A state is depicted by a rounded rectangle with the name of the state inside the rectangle. Additionally, a state can be refined through another state-transition diagram representing the internal state machine of the decomposed state. Hence, state-transition diagrams can be hierarchically composed. A decomposition of a state may consist of several concurrent states (or-states), i.e. states that are concurrently active. Additionally, there are notations for "pseudo-states" indicating the initial state and optional termination points, represented in the state-transition diagram as follows.

Initial State. An initial state is used to indicate the starting point of a state-transition diagram by being connected through a (usually unlabeled) transition to the starting state. There is only one initial state for each concurrent part of a state-transition diagram. There can be no transitions to an initial state. An initial state is a "pseudo-state", i.e. the system cannot be in this state. An initial state is depicted by a filled black circle.

Final State. A final state is used to indicate the termination on an event of the finite state machine represented by the state-transition

diagram. Final states are "pseudo-states", i.e. the system cannot be in such a state. State-transition diagrams may have zero or more final states. There can be no transitions from a final state. A final state is depicted by a filled black circle surrounded by a second (non-filled) circle.

Transition. A transition represents a possible change between two states. A transition is triggered by an event and takes only place if the corresponding optional guard expression evaluates to "true". Subsequently, an optionally associated action is executed. In Harel's state-transition diagrams, transitions take no time. A transition is depicted by an arrow, directed from the source state to the sink state, and labeled with the event's name, the optional guard expression, and the optional action to be executed when the transition is taken.

Figure 3.7 shows an example of a state-transition diagram that illustrates the high-level behavior of a system with error handling. The system consists of the two high-level states *On* and *Off*. The state *On* is a composite concurrent state, i.e. it consists of concurrent sub-states, namely *Operation* and *Error_handling*. The normal operation of the system is (after being started) to initialize, and then enter a loop of loading the next item and then processing the item. If an error occurs during loading, the system falls back into initializing. In parallel, if the error handling catches an *error* event, it checks the system. If the system can recover from the error, the error handling changes state to *OK*. Otherwise, if a *timeout* occurs during checking the system, a *stop* event is generated and the error handling quits. The *stop* event causes the system to change its state back to *Off*. Note that the events shown in the diagram need not be generated from the source state of a transition that reacts to the event. Events may be generated from anywhere in the system, e.g. there could be a switch somewhere else in the system (not modeled in the example diagram) that could generate a *stop* event and make the system change its state to *Off* during normal operation.

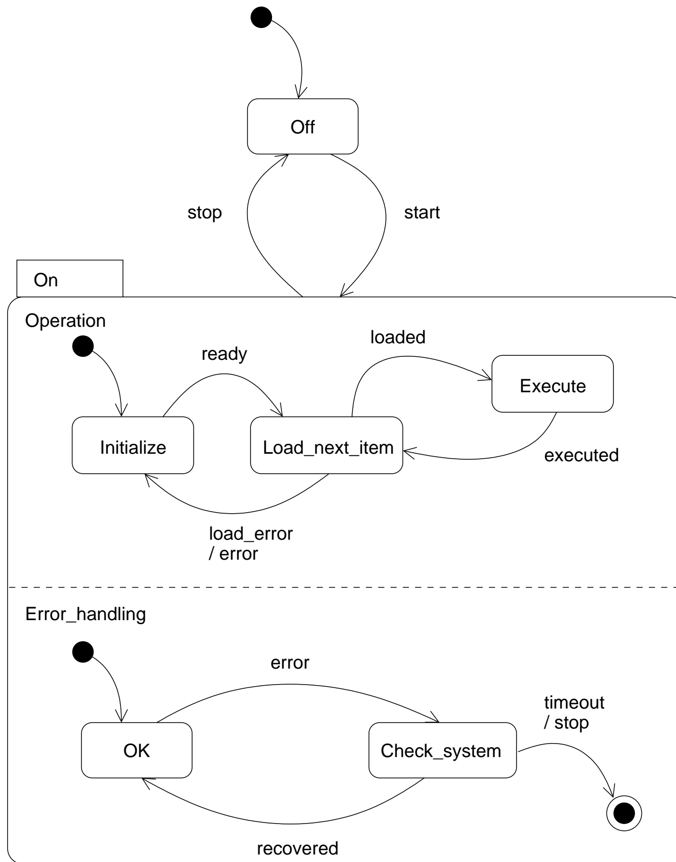

Fig. 3.7: Example of a state-transition diagram

3.2 Extracting Concepts from Object-Oriented Techniques

This section describes the object-oriented techniques that are considered to be most important for being integrated with the structured techniques presented in the previous Section 3.1. The different techniques are briefly described and decomposed into their elements, followed by a description of the syntax and semantics of each element. Note that the specification techniques and concepts presented herein are subsets of UML diagram types; more extensive descriptions can be found in [Gro01].

3.2.1 Selecting Representative Object-Oriented Techniques

The object-oriented specification techniques considered for the integration purposes are all members of the UML notation. As for the structured specification techniques described in Section 3.1, this section also focuses on the major elements of each technique rather than describing the complete set of elements as presented in [Gro01]. In particular the details of the UML meta-model that represent minor modeling artifacts have been left out. Furthermore, the focus is on elements that describe the abstract specification of a system, rather than on elements that describe instantiations, e.g. the object or the stimuli concepts of the UML (see [Gro01]). Elements of this kind basically mirror the definitions of their abstract counterparts, e.g. an object requires a class definition, it only adds support for storing run-time values. Similarly, a stimulus instantiates a message, which does not add new important aspects to the concept of messages. In summary, the representations of specification techniques considered in this section are subsets of the respective UML techniques. However, the basic principle of integration presented in this thesis can also be expanded to the full representations of the UML specification techniques according to the UML meta-model.

The UML use case diagram (based on the concepts from OOSE [JCJO92]) is considered for integration purposes as it is a widely used technique for illustrating coarse system functions in early specification phases. The static structure diagram is also considered; however, object diagrams are excluded from the considerations because they merely illustrate instantiations of an existing class diagram without having additional semantically interesting elements. The UML statechart diagram (based on Harel's statecharts [Har87]) is widely used and thus considered for being integrated with other specifi-

cation techniques. The activity diagram is omitted from the integration because in [Gro01] it is considered to be a derivative of the statechart diagram, which will be considered for integration. The sequence diagram is a widely used technique for specifying the temporal sequence of messages between different entities, and is also considered for integration. The collaboration diagram is tightly coupled to the sequence diagram representations, though, focusing on the structural aspect rather than the temporal aspect. The collaboration diagram shares for the greater part the meta-model with the sequence diagram in the UML specification [Gro01]. Hence, it is also (implicitly) considered for integration. The UML component and deployment diagrams (summarized in the UML specification [Gro01] as implementation diagrams) are not considered for integration because they are employed for implementation activities, which are outside the scope of this thesis.

Table 3.3 summarizes the selected object-oriented (UML) specification techniques that are considered for integration purposes in this thesis. The following subsections describe the techniques and their concepts in terms of their syntax and semantics.

Tab. 3.3: Selected UML specification techniques

Selected UML Technique
Use Case Diagram
Static Structure Diagram
Statechart diagram
Sequence diagram / Collaboration diagram

3.2.2 Common Concepts

The UML meta-model [Gro01] makes intense use of inheritance. The declaration of structural and behavioral features are distributed on different logical levels of abstraction and actual elements of a diagram type are composed by "inheriting" features from superordinate elements. In this way, a number of UML concepts are defined independently of a specific diagram type, but are commonly used in several diagram types. Before elaborating on the different diagram types in the following subsections, the common concepts are described in the following paragraphs.

Classifier. A classifier is a generic model element that possesses structural and behavioral features. It combines capabilities for both, storing data in the form of typed variables (called attributes) as well as for functions (called operations). A classifier is an abstract model element, i.e. it cannot itself be instantiated. It is superordinate to other model elements that are explained later, e.g. class, use case, or actor. The features of a classifier can be:

Structural Feature. A structural feature is a named abstract property of a classifier that is used to describe data structures that belong to the classifier. Currently, the only inheriting element is the static (non-abstract) attribute, which is employed by instantiations of classifiers (i.e. their subclasses, as classifiers are abstract) to store values according to the value domain described by the data-type of the structural feature. An attribute is a named structural feature of its owner that describes a value domain (attribute type) of values that instances of the owner (objects) may hold.

Behavioral Feature. A behavioral feature is an abstract property of a classifier that describes a dynamic behavioral aspect of the classifier. Behavioral features are instantiated in the form of one of the subclasses of classifiers, e.g. as operations / methods. One subclass of the behavioral feature is the operation. An operation is a named behavioral feature of its owner that references dynamics of its owner in the form of functional specifications. Operations may have a list of parameters, the "signature". A call of the operation must comply with the signature of the operation, i.e. the types of parameters of the call must match the types declared in the operation signature.

Relationship. A relationship is a generic connection between elements. It is an abstract model element, only its subordinates (i.e. association or generalization) may be instantiated.

Association. An association is a subclass of the abstract relationship that represents a semantic relationship between two or more classifiers. The classifiers involved are connected via association ends to an association. Each association end defines a number of properties for the involvement of the respective classifier in the association. These are the name (role name) of the classifier, the multiplicity (a set of allowable cardinalities) of instances involved in the association, and the reachability (whether the classifier is visible for other involved classifiers in the respective association). Furthermore, aggregation and composition of classifiers through associations can be specified through an association end.

Generalization. A generalization is used to define an inheritance hierarchy among classifiers. It represents a relationship between a more general and a more specific classifier. The more specific classifier inherits all structural and behavioral features as well as the relationships from the more specific classifier, and adds new features and relationships for a specialized purpose.

3.2.3 Static Structure Diagram

Static structure diagrams are employed to illustrate the static elements of a system as well as the relationships among them. The static structure diagram is based on concepts of Rumbaugh's OMT class diagram [RBP+91] and Booch's class diagram [Boo94a]. The static structure diagram also allows for representing class diagrams at instance level, the object diagrams, illustrating class instances (objects) and instantiated associations (links) between them. Furthermore, it allows for hierarchical decomposition of class diagrams by grouping classes into packages and showing relationships among packages. In summary, static structure diagrams consist of classes, different kinds of relationships, and packages. Note that elements that represent instantiations (objects, links, etc.) are excluded from the description, as motivated in Subsection 3.2.1. The respective graphical notation of elements of the static structure diagram is shown in Figure 3.8.

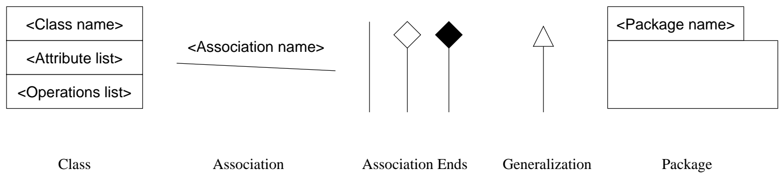

Fig. 3.8: Elements of a static structure diagram

Class. A class is a classifier (see 3.2.2) that represents a template for a set of objects that share the same semantics and have the same features and relationships. As a class inherits from classifier, the features of a class are also of a structural (data structures) or a behavioral (operations) kind. Relationships among classes are usually associations with other classes, or generalizations, i.e. illustrations of inheritance hierarchies. A class is depicted by a rectangle, labeled with the name of the class inside the rectangle. Additionally, inside the rectangle, the compartments for attributes and operations may also be displayed, each separated by a horizontal line. In this case, the attribute compartment textually lists the attributes of the class, and the operations compartment lists the operations and the operation signatures of the class.
Special case:

Association Class. An association class is both a class and an association, i.e. it inherits from both, a class and an association. It is

used if additional information (usually only data, i.e. structural features) needs to be stored about the association.

Association. The semantics of associations are explained above, see Subsection 3.2.2. Associations are depicted as a line between the classifiers involved in the association, labeled with the name of the association. Association ends are represented by different graphical adornments, depending on the kind of association end. "Normal" association ends have no special graphical notation. Association ends that represent an aggregation are depicted by a hollow diamond or, in the case of strong aggregation (composition), a filled diamond. For all kinds of association ends, the role name of the respective classifier is displayed near the association end. Furthermore, if applicable, the navigability is indicated by an arrowhead pointing at the associated classifier. Optionally, the multiplicity of the association end can be displayed as text.

Generalization. The semantics of generalizations are explained above, see Subsection 3.2.2. Generalizations are depicted as lines from the more specific classifier to the more general classifier, with a hollow arrowhead at the end pointing at the more general classifier.

Package. A package is a logical grouping of model elements. Packages are employed to create a logical, high-level organization of model elements. A package is depicted by a stylized folder, i.e. by a large rectangle with a small rectangle, the tab of the "folder", at top left. The name of the package is placed either in the tab or inside the large rectangle.

Figure 3.9 shows an example of a static structure diagram, illustrating the relationship between an employee and his employer. The class *Employee* inherits the features of the class *Person*, i.e. the attributes *name* and *address*. In the association *Job*, the class *Employee* plays the role of *employee*, and the class *Company* plays the role of *employer*. The association class *Job* stores more detailed information about the *Job* association, namely the *salary* of the employee. The class *Department* is an aggregate of the class *Company*, i.e. an instance of *Company* is composed of *Department* instances.

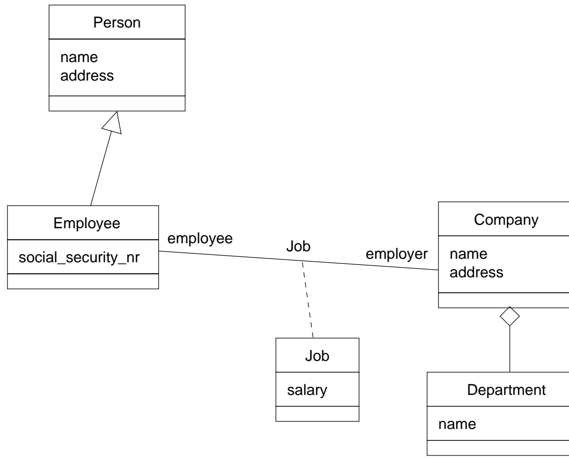

Fig. 3.9: Example of a static structure diagram

3.2.4 Use Case Diagram

Use case diagrams are employed to specify the functionality of a system (or a part of it) from a high-level perspective, i.e. without revealing structural or behavioral details. It provides an overview of the system and how it is embedded in its environment. Use case diagrams are based on Jacobson's approach to object-oriented software engineering, originally described in [JCJO92]. The main constituents of a use case diagram are actors and uses cases, connected by different kinds of relationships. Their respective graphic notation is shown in Figure 3.10.

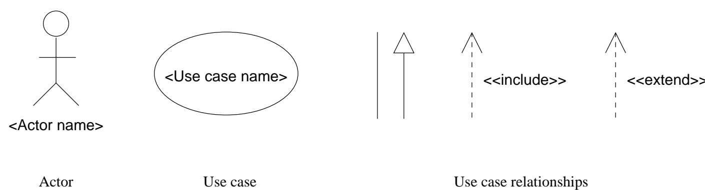

Fig. 3.10: Elements of a use case diagram

Actor. An actor represents a classifier (see 3.2.2), taking a certain role in the use case under consideration. Actors are considered to be external to the system, i.e. their internal structure is not of interest. Although actors are depicted by a stick man, they do not need to represent persons, they can also represent external systems. The stick man representing the actor is labeled with the name of the actor under the stick man figure.

Use case. A use case represents a grouping of several actions to a single item of functionality of the system that is invoked by an actor in order to achieve a certain goal. A use case can further be refined through another use case diagram associated with the use case. A use case is depicted by an ellipse, labeled with the name of the use case inside the ellipse.

Use case relationship. In use case diagrams, actors are connected to use cases through associations (see description above, 3.2.2). Associations are depicted by lines between the actor and the referred use case. Use cases can be interrelated through "include" or "extend" relationships, indicating that one use case includes or extends the other use case, respectively. In the case of an extension, a specific location in the use case (the extension point) is specified, where the referenced use case is

extended. A use case may have none or several extension points. The relationships are depicted by dashed arrows from the use case including / extending the referred other use case, labeled with the keyword *include* or *extend*, respectively. Furthermore, generalizations between either actors or use cases can be established as described in 3.2.2. A generalization is depicted by an arrow with a hollow arrowhead, pointing from the more specific classifier (i.e. actor or use case) to the more general one.

Figure 3.11 shows an example of a use case diagram illustrating an excerpt of a system specification for maintaining customer data. The "normal" *User* uses the system for registering a new customer in the *Register customer* use case, which in turn makes use (includes) of the *Create new dataset* use case. The *SuperUser* inherits from the *User* actor, i.e. the *SuperUser* is also able to use the system as described by the *Register customer* use case. Additionally, the *SuperUser* may also use the system to delete a customer's dataset in the *Delete customer* use case.

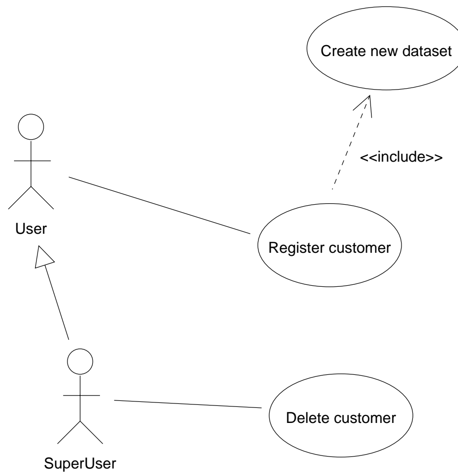

Fig. 3.11: Example of a use case diagram

3.2.5 Collaboration Diagram

Collaboration diagrams and sequence diagrams (see next subsection) are both employed to specify interactions among a set of classifier instances. However, both kinds of diagrams each illustrate a different aspect of the behavior. Collaboration diagrams emphasize the links between the involved classifier instances, whereas sequence diagrams emphasize the temporal ordering of messages between the classifier instances. In summary, collaboration diagrams consist of classifier instances (usually classes), and links among them, optionally labeled with a message that establishes the link and a sequence number that indicates the ordering of messages in the diagram. The graphical notations of the elements of a collaboration diagram are summarized in Figure 3.12.

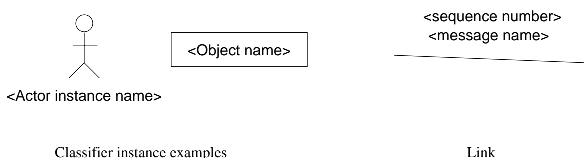

Fig. 3.12: Elements of a collaboration diagram

Classifier Instance. The abstract classifiers are described in Subsection 3.2.2. The classifier subclasses class and actor are described in Subsection 3.2.3 and Subsection 3.2.4, respectively. In a collaboration diagram, the instances of classes and actors are used as classifier instances. They are depicted in the form of a class or an actor (see 3.2.3 and 3.2.4, respectively).

Link. A link is an instance of an association (see Subsection 3.2.2). It represents a list of references to classifier instances that are jointly involved in a collaboration. Links are depicted, as associations, by lines between the classifier instances that are involved in the respective association. Also, the role names of the involved classifier instances may be shown near to the respective link ends.

Figure 3.13 shows an example of a collaboration diagram illustrating the necessary elements for a user to print a customer's order. The *User* sends the *printOrder* message to the *OrderManager*. The *OrderManager* fetches the requested order data by sending the *getOrder* message to the *Order* class, which in turn fetches the name and address using the messages *getName* and *getAddress* from the *Customer* class. The colon before class names indicates

that all instances of the respective classes may take part in this collaboration, otherwise explicit instance names would be stated.

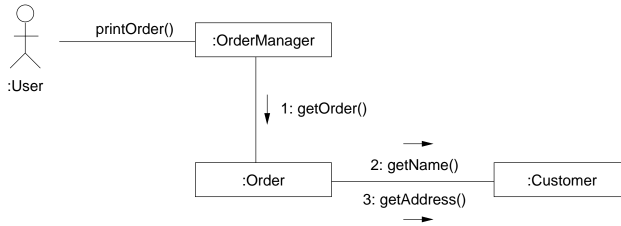

Fig. 3.13: Example of a collaboration diagram

3.2.6 Sequence Diagram

As described above (see Subsection 3.2.5), sequence diagrams are employed to illustrate temporal ordering of messages in an interaction between classifier instances. The involved instances are represented in the horizontal dimension of the sequence diagram, the vertical dimension represents time progression, where time usually proceeds from top to bottom. A sequence diagram consists of classifier instances (usually objects or actor instances), their lifelines and activations, and messages between activations. Their respective graphical notation is shown in Figure 3.14.

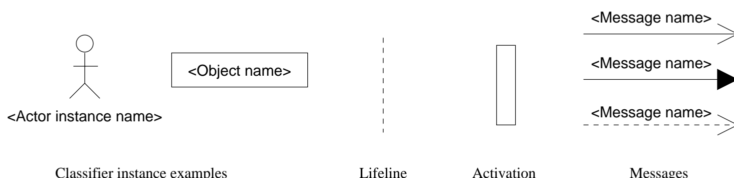

Fig. 3.14: Elements of a sequence diagram

Classifier Instance. Classifier instances are described above, see 3.2.5. In a sequence diagram, the instances are distributed horizontally, with their lifelines directed downwards. All instances are usually placed at the top of the sequence diagram. However, if an instance is created during the illustrated interaction, the respective instance representation may be placed vertically lower, corresponding to its creation time.

Lifeline. A lifeline is associated with a classifier instance and denotes the existence of the instance during a period of time. A lifeline is represented as a vertical dashed line, starting at the representation of the associated instance and ending at the corresponding point in time where the instance is destroyed.

Activation. An activation represents an action (or a sequence of actions) that is performed by an instance, i.e. a period of time in that the respective instance is active. An activation is depicted by a rectangle on the lifeline of an instance, starting at the corresponding time when the instance is activated and ending at the corresponding time when it completes the action.

Message. A message represents a unidirectional communication between two instances, invoking an operation of the destination instance. Messages are depicted by different kinds of arrows between activations

of the involved instances, depending on the kind of communication. Synchronous messages (procedure calls) are depicted by arrows with a solid filled arrowhead, asynchronous messages are depicted by arrows with a stick arrowhead. Dashed arrows with a stick arrowhead represent a return from a procedure call, i.e. from an activation that has previously been initiated by a synchronous message.

Figure 3.15 shows an example of a sequence diagram. The example describes the establishment of a phone line for a phone call between a caller and a callee, using two telephones and a switch between both. The *Caller* (an object of class *Person*) sends the synchronous message *liftReceiver* to object *phone1*. Then, *phone1* answers with *dialTone*. The *Caller* sends a *dial* message to *phone1*, which in turn answers with *ringTone*. *phone1* sends a *connect* message to the *switch*, which in turn sends a *connect* message to *phone2*. *phone2* sends a *ring* message to the *Callee*, who answers with *liftReceiver*, whereupon *phone2* answers with *stopRing* and sends *answer* back to the switch, indicating that the *Callee* has answered the call. The *switch* signals the *phone1* that the *Callee* is *connected*, whereupon the *phone1* sends a *stopRingTone* to the *Caller*, indicating that the connection is established and *Caller* and *Callee* can start to talk.

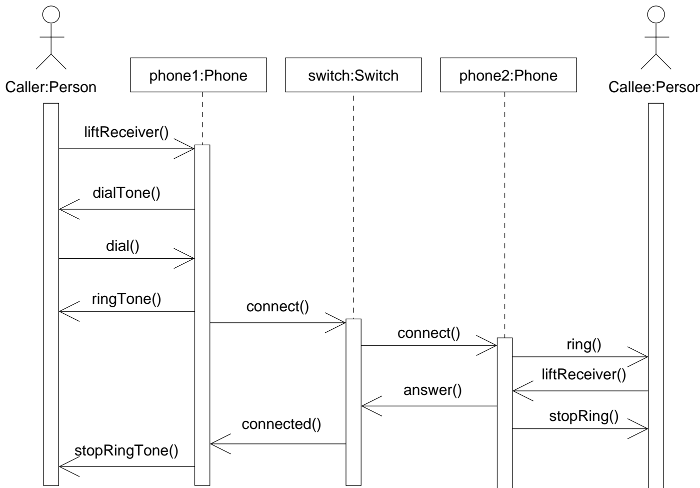

Fig. 3.15: Example of a sequence diagram

3.2.7 Statechart Diagram

Statechart diagrams are employed to specify the discrete behavior of (parts of) a system by the possible states the system can be in and the discrete events (e.g. operation / procedure calls) that make the system change its state. Statechart diagrams are a graphical representation of finite-state machines. The statechart diagrams are based on Harel's statecharts, originally described in [Har87], see also Subsection 3.1.5. The notation of statechart diagrams is very similar to Harel's statecharts. However, the semantics of some of the elements differ due to the different context (the UML) in which the statechart diagram is embedded. Harel's statecharts are employed to specify the behavior of a process, whereas statechart diagrams are employed to specify the behavior of a class or of one of its operations. The major differences between the UML statechart diagram and Harel's statecharts can be summarized as follows, see also [Gro01], page 2-175, for a more detailed description.

- Events may carry parameters.
- Call events triggering operations have been added.
- Transition actions may take time.

Statechart diagrams consist of different kinds of states and labeled transitions between states. Their graphical notation is similar to those of state-transition diagrams shown in Figure 3.6 and explained in Subsection 3.1.5. However, in order to provide structural and semantical differences to the state-transition diagrams, the statechart variation in this thesis does not support concurrent composite states. This lack is intentional in order to demonstrate later the capabilities of the proposed principle for mappings between differing concepts in Chapter 6 and Chapter 7.

Figure 3.16 shows an example statechart diagram that illustrates how a phone instance behaves during setting up a phone connection. It is a simplified adaption from the example in [Gro01], page 3-138. The example consists of the top-level states *Idle* and *Active*. The state *Active* is refined through a nested statechart diagram. The system changes from *Idle* to *Active* if the *lift receiver* event occurs. At the same time, the *liftReceiver* action is invoked. Entering the *Active* state, the system initializes the dialing in *InitDialing*, where the *dialTone* action is invoked on entering the state. If the *dial number* event occurs (carrying the number dialed in the *phoneNumber* parameter), the system changes its state to *Connecting*. After the connection

is established, the system implicitly changes its state to *Ringing*, where the *ringTone* action is invoked on entering the state. The *callee answers* event makes the system change its state to *Connected* and at the same time invokes the *stopRingTone* action.

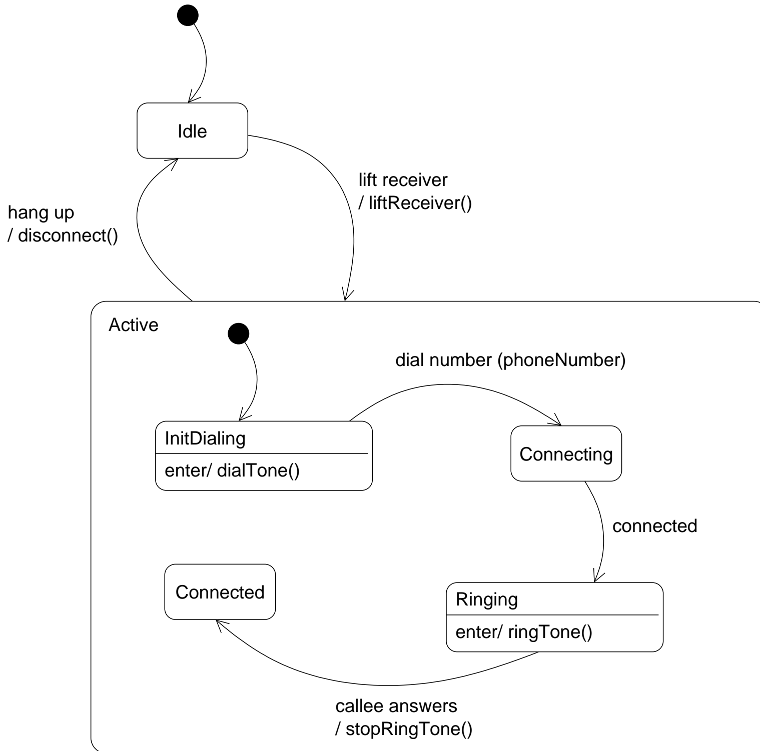

Fig. 3.16: Example of a statechart diagram (adapted from [Gro01])

4. CREATING META-MODELS OF STRUCTURED AND OBJECT-ORIENTED TECHNIQUES

This chapter presents and describes meta-models for the specification techniques presented in Chapter 3. The meta-models in this chapter describe the data that may be generated using the different specification techniques, e.g. in tools. In the next chapter, Chapter 5, a new common meta-model is synthesized from the meta-models in this chapter. Then, in Chapter 6, mappings between the meta-models are described that allow for transformations of specifications between different representations. Note that the meta-models presented herein are only one possible way of meta-modeling the specification techniques, there may be structurally different solutions that exhibit the same semantics.

4.1 Meta-Modeling Framework

This section presents the modeling framework that is employed in this thesis for creating meta-models of the above presented specification techniques, the common meta-model and the transformations between the meta-models.

The international standard for the exchange of product model data ISO 10303 (STEP) has been developed within systems engineering for describing systems engineering data and for modeling data exchange protocols for specific domains. STEP is well-established in many systems engineering areas, and a number of widely used data exchange protocols have been created using STEP, e.g. ISO 10303-210 for printed circuit assembly product

design data, or ISO 10303-214 for automotive mechanical design processes. Comparable other frameworks such as the combination of the UML [Gro01] and the Meta-Object Family (MOF [Gro00]), or the XML-based mechanisms of the Semantic Web initiative [BLHL01] are not as mature as STEP and currently not well-established within systems engineering for modeling standardized data exchange protocols. Also in the SEDRES projects (see Subsection 1.3.3), STEP has also been employed for preparing the ISO 10303-233 (AP-233) proposals, a protocol for the exchange of systems engineering design data. The work presented in this thesis can be seen as an extension of the work performed in the SEDRES projects. Thus, STEP is also employed for implementing the principle presented in this thesis, besides the primary focus of this thesis on systems engineering as its major application area.

4.1.1 STEP

As motivated above, the framework used for illustrating the proposed integration principle in this thesis is the Standard for the Exchange of Product Model Data (STEP) ISO 10303, published by the International Standardization Organization (ISO [ISO02]). STEP was designed for representing and exchanging data associated with a product during its life-cycle, independent of a specific platform or tool. It consists of a number of parts, designated by a unique number. STEP is meant to be extended by new parts for providing support for specific concepts of specific domains, usually in the form of application protocols (APs), e.g. the future AP-233 for systems engineering design data exchange. The parts of STEP are logically grouped as follows.

Description Methods. This group contains methods for describing integrated resources and application protocols. Important parts of this group are the members of the EXPRESS language family, e.g. EXPRESS and EXPRESS-G (both ISO 10303-11) and EXPRESS-X (ISO 10303-14). EXPRESS, EXPRESS-G and EXPRESS-X are briefly explained in Subsections 4.1.2 and 4.1.3.

Implementation Methods. This group contains bindings from models built with the description methods to implementations. Parts from this group are the part-21 interchange file format (ISO 10303-21), repositories (ISO 10303-43), and programming language support, e.g. for C++ (ISO 10303-36).

Conformance Testing. This group provides a framework for testing STEP-based implementations.

Integrated Resources. This group contains concepts that are commonly used by different application protocols.

Application Protocols. This group contains domain-specific data models for the exchange of data between applications, e.g. AP-210 for electronic printed circuit assembly (ISO 10303-210).

In summary, STEP can be seen as a standardization framework that provides a number of tools for creating data exchange standards in the form of protocols between software applications. Application protocols that have been accepted as international standards are included in ISO 10303 under a new unique part number and can be publicly accessed and used. STEP standards are valid for a period of five years. After this time the respective standard needs to be evaluated and re-confirmed, probably including changes and updates.

The following Subsections 4.1.2 and 4.1.3 briefly present the members of the EXPRESS family that are employed in this thesis for describing the meta-models (using EXPRESS and EXPRESS-G) and the transformations between the meta-models (using EXPRESS-X). Thereafter, each subsection describes an EXPRESS-G meta-model for each of the specification techniques presented in Chapter 3.

4.1.2 EXPRESS and EXPRESS-G (ISO 10303-11) Overview

EXPRESS (ISO 10303-11) is an object-flavored textual specification language that has is roots in entity-relationship modeling and database modeling. The basic elements are entities and attributes (unidirectional relationships between entities). EXPRESS also supports the object-oriented concepts of inheritance and encapsulation. EXPRESS is mainly used to specify integrated resources and application protocols of the above presented STEP standard.

EXPRESS-G (also described in ISO 10303-11) is a graphical subset of the EXPRESS language. It does not support all features of the textual EXPRESS language. EXPRESS-G is usually the starting point for modeling, followed by a refinement of the model in the textual notation EXPRESS. EXPRESS modeling tools often allow for modeling both EXPRESS and

EXPRESS-G, while keeping both representations consistent.

The most important elements of EXPRESS-G are explained in the following paragraphs. Their graphical notation is shown in Figure 4.1.

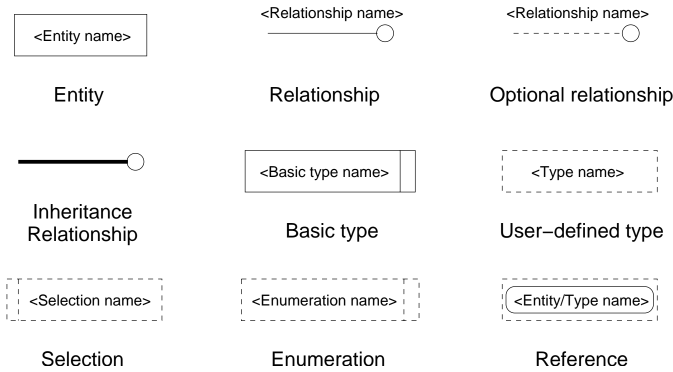

Fig. 4.1: Important elements of the EXPRESS-G notation

Entity. An entity represents a domain of objects with a common structure and semantics, comparable to the entity concept of entity-relationship modeling (see Subsection 3.1.4) and the class concept in object-oriented techniques (see Subsection 3.2.3).

Relationship. In EXPRESS, a relationship is to be interpreted as an attribute of the entity it origins from, i.e. an attribute is a named unidirectional relationship between an entity and the data-type of the attribute. The data-type may be a basic type, a user-defined type, a selection, an enumeration, or another entity. Optional attributes are represented by a relationship symbol with a dashed line, as opposed to a solid line used for depicting mandatory attributes.

Inheritance relationship. An inheritance describes a relationship between a more general and a more specific entity, just as the object-oriented inheritance concept. The more specific entity inherits the relationships of the more generic entity.

Basic type. Basic types are the building blocks for complex data structures. Basic types are integers, strings, booleans, etc., but also arrays, lists, sets, and bags.

User-defined type. A user defined type can be described as alias for an entity or another data-type. In EXPRESS-G, this is illustrated by a

single unnamed relationship from the user-defined type symbol to the aliased element.

Selection. A selection is a list of entities and represents the union of the domains of the entities in the list, of which one is selected when the selection is instantiated.

Enumeration. An enumeration is an ordered set of names. It is used as a data-type, where the names represent the possible values of the data-type. Note that the graphical notation for an enumeration does not show the possible values and that it is very similar to the graphical representation of a selection. A distinguishing mark between both EXPRESS-G representations is the fact that enumerations have no further relationships, whereas selections need to reference at least one more element.

Reference. A reference represents an entity or a type (i.e. a user-defined type, a selection, or an enumeration) that is defined in another schema. The reference symbol is labeled with the name of the referenced schema, a dot and the name of the referenced element, e.g. *global.label* for referencing the *label* type in the schema *global*.

Besides the basic EXPRESS-G elements presented above, the following non-standard extensions to EXPRESS-G, and their graphical notation shown in Figure 4.2, are employed in this thesis in order to illustrate references between models as well as instantiations of models.

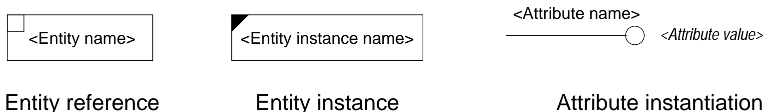

Fig. 4.2: Extensions to the EXPRESS-G notation

Entity reference. An entity reference represents a placeholder for an entity, indicating that the entity and its attributes are defined somewhere else in the same or in another model. This notation is a simplification of the original EXPRESS-G entity reference, which additionally indicates the location of the definition of the referenced entity. However, due to the logical partitioning of the models in this thesis, the simple indication of a reference is sufficient.

Entity instantiation. In EXPRESS-G, entity instantiations are usually not shown. However, in order to illustrate how an actual specification is

instantiated under the different meta-models during a transformation from a source to a sink tool, the graphical notation in Figure 4.1 for illustrating entity instantiations is used in this thesis.

Attribute instantiation. Like entity instantiations, attribute instantiations are usually not shown in EXPRESS-G models. However, for the same reasons as for entity instantiations, the notation in Figure 4.1 is used in this thesis to represent attribute values in an actual specification.

Figure 4.3 shows an EXPRESS-G example illustrating the relationships between an *employee*, his employing *company* and *projects*.

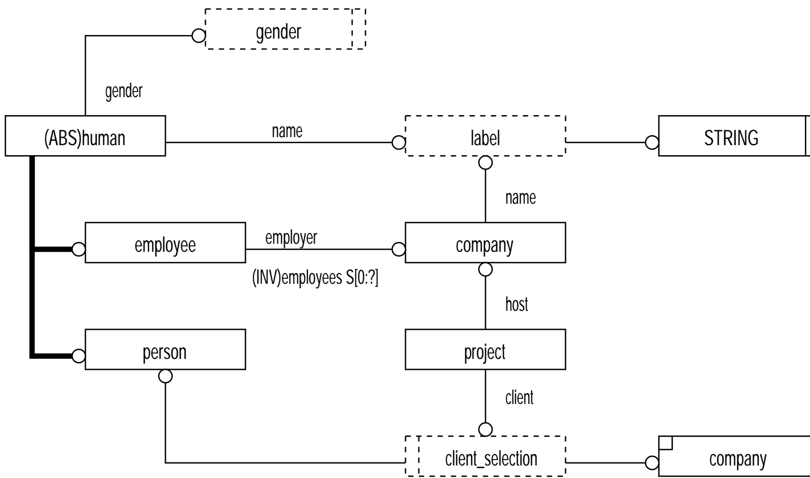

Fig. 4.3: EXPRESS-G example

The *employee* inherits from the abstract super-class *human*, which has two attributes, namely *gender* and *name*. The *gender* is represented by the *gender* enumeration. Note that the possible values (*male* or *female*) of the *gender* enumeration are not shown in EXPRESS-G, but only in the underlying EXPRESS code (see below). The *name* of a *human* is of the user-defined data-type *label*, which in turn is associated with the basic EXPRESS type *STRING*. In EXPRESS, relationships are to be interpreted as attributes of the entities they originate from, e.g. the *name* relationship between *human* and *label* is represented as attribute of the entity *human*. Inverse relationships can also be described, e.g. the relationship *employer* is, from the perspective of a *company*, a relationship *employees*, in this case with zero or more instantiations of *employee* belonging to the inverse relationship. The

project illustrates the use of a selection with its *client* attribute, which either references a *person* or a *company* through the *client_selection*.

The underlying EXPRESS code of the above EXPRESS-G example in Figure 4.3 looks as illustrated in Figure 4.4.

```
1  SCHEMA example;
2
3   TYPE label = STRING;
4   END_TYPE;
5
6   TYPE gender = ENUMERATION OF (female, male);
7   END_TYPE;
8
9   TYPE client_selection = SELECT (person, company);
10  END_TYPE;
11
12  ENTITY human
13   ABSTRACT SUPERTYPE;
14   name : label;
15   gender : gender;
16  END_ENTITY;
17
18  ENTITY employee
19   SUBTYPE OF (human);
20   employer : company;
21  END_ENTITY;
22
23  ENTITY company;
24   name : label;
25  INVERSE
26   employees : SET [0:?] OF employee FOR employer;
27  END_ENTITY;
28
29  ENTITY project;
30   host : company;
31   client : client_selection;
32  END_ENTITY;
33
34  ENTITY person
35   SUBTYPE OF (human);
36  END_ENTITY;
37
38 END_SCHEMA;
```

Fig. 4.4: Example of an EXPRESS schema

In EXPRESS, models can be partitioned into schemata (see *SCHEMA* keyword in the above code), which represent groups of logically related types, entities, functions, etc. This supports a modularized structure of EXPRESS models and allows for reusing schemata in several models. A more detailed introduction to EXPRESS and EXPRESS-G is provided by Schenck and

Wilson in [SW94]. The EXPRESS specification can be obtained from ISO [ISO02].

Figure 4.5 shows an example instantiation of the model in Figure 4.3, using the instantiation elements presented in Figure 4.2. This example instantiation describes the case where two employees (*Marc Meyer* and *Rebecca White*) are employed by *Software Production Inc*, which hosts a project for their client *Systems Development Inc*.

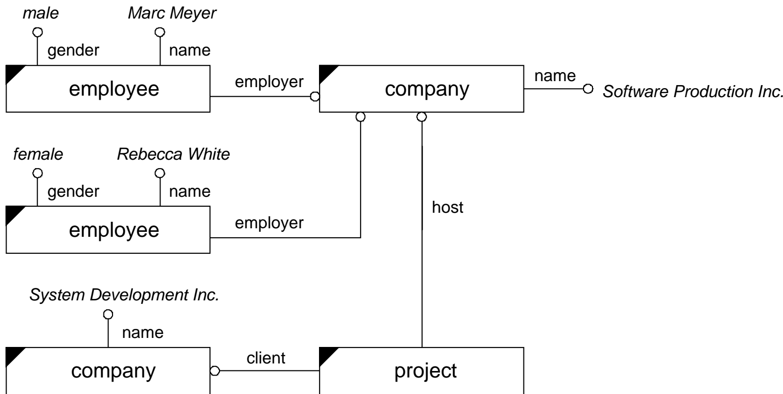

Fig. 4.5: EXPRESS-G example instantiation

4.1.3 EXPRESS-X (ISO 10303-14) Overview

EXPRESS-X (ISO 10303-14) is a textual notation for formally specifying mappings between EXPRESS models. There are EXPRESS-X tools that allow for compiling EXPRESS-X code into executable transformation engines that allow for transforming data between representations under the different EXPRESS models. In this thesis, EXPRESS-X is employed to describe the transformations between the meta-models of the specification techniques and the common meta-model. EXPRESS-X is a natural choice, as the work in this thesis employs the ISO 10303 (STEP) framework, of which EXPRESS-X is a part.

Chapter 6 describes these mappings, accompanied by a more detailed introduction to EXPRESS-X in Section 6.2.

4.2 Common Types

The meta-models presented in Sections 4.3 and 4.4 make use of common types that are shown in Figures 4.6 and 4.7. The basic types *string_data_type*, *integer_data_type*, *natural_number_type*, and *boolean_data_type* have been introduced to avoid the explicit use of EXPRESS data-types in the meta-models. This has been done in order to centralize type definitions for the ease of later modifications. Along the same line, the *label* and *text* types have been defined. Currently, their definition is technically equivalent. However, they differ in semantics. The *label* type represents a name of an element, whereas the *text* type allows to store a text that may consist of several lines and paragraphs of text, including punctuation. The *expression* entity represents a boolean expression, which is specified in its *specification* attribute. The language in which the specification is referenced in the *language* attribute.

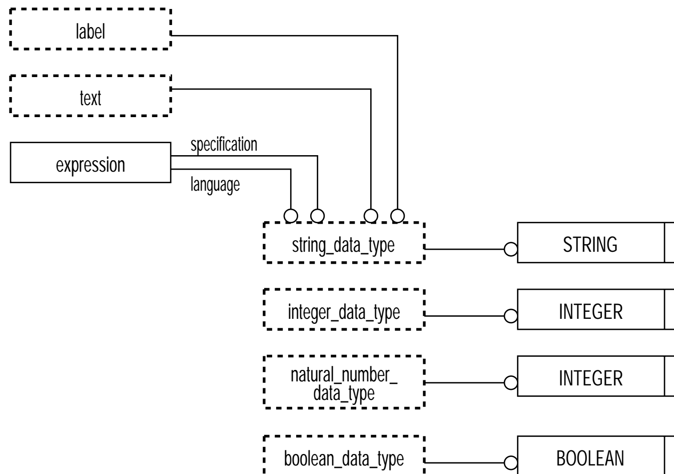

Fig. 4.6: Meta-model of common basic type

The cardinality of elements in a given context is stored in the multiplicity types shown in Figure 4.7. Multiplicity can either be declared as a single multiplicity or as a multiplicity range, represented through the *single_multiplicity_selection* type and the *multiplicity_range* entity, respectively. The single multiplicity is either a natural number or an infinity sign. The multiplicity range defines the inclusive start and end of the range in the

attributes *range_start* and *range_end* of the *multiplicity_range* entity.

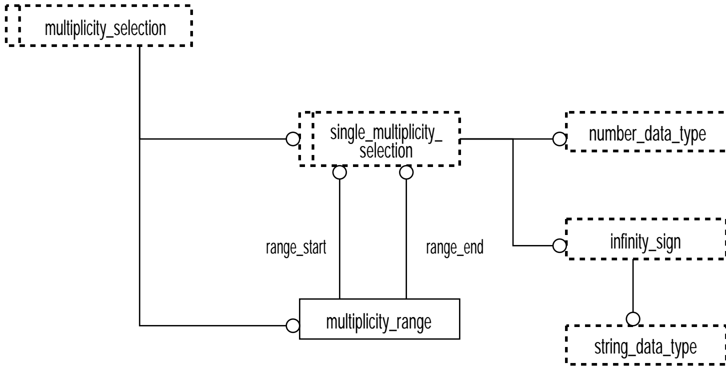

Fig. 4.7: Meta-model of the common multiplicity type

The meta-models in the following sections make use of the common types by referencing the respective type through a *global.* preceding the type name. For example, *global.label* references the common type *label* described above.

4.3 Meta-Models of Structured Techniques

The following subsections each describe a meta-model of one of the structured specification techniques presented in Section 3.1. The meta-models presented herein are created on the basis of the descriptions in Subsection 3.1, i.e. the concepts of each specification technique and their inter-relationships. Note that each of the subsections presents only one possible way of meta-modeling the respective specification technique, alternative representations are feasible.

4.3.1 Data-Flow Diagram

The meta-model for data-flow diagrams described in Subsection 3.1.2, is shown in Figure 4.8. In the meta-model, the data-flow diagram itself is represented by the *dfd_diagram* entity. It consists of diagram elements, that are represented by the *dfd_element* entity. Each *dfd_element* is owned by a data-flow diagram, depicted by the *owner* relationship to *dfd_diagram*. Furthermore, each diagram element has a name, represented by the *name* attribute of *dfd_element*. The actual elements in a dataflow diagram are represented by the following entities:

dfd_process. The *dfd_process* entity represents a process in a data-flow diagram. A process may be refined through another data-flow diagram, which is represented by the optional *detail* relationship of *dfd_process*.

dfd_data_flow. The *dfd_data_flow* entity represents a data flow. It connects a source and a target element in the form of a *dfd_process*, a *dfd_data_store*, or a *dfd_external_entity*. The identifiers associated with the data that flows through a data-flow, are represented by the *data_identifiers* relationship.

dfd_data_store. The *dfd_data_store* entity represents a data store.

dfd_external_entity. The *dfd_external_entity* entity represents an external entity.

dfd_incoming_control_flow. The *dfd_incoming_control_flow* entity represents a control flow that is going into a control process (*dfd_control_process*), which is referenced through the *target_connector* relationship. The *source_connector* relationship represents the source of the incoming control flow and references either a *dfd_process* or a *dfd_external_entity*.

dfd_control_process. The *dfd_control_process* entity represents a control process.

dfd_outgoing_control_flow. The *dfd_outgoing_control_flow* entity represents a control flow that is going out from a control process (*dfd_control_process*), which is referenced by the *source_connector* relationship. The target of the outgoing control flow is referenced through the *target_connector* relationship. It references either a *dfd_process* or a *dfd_external_entity*.

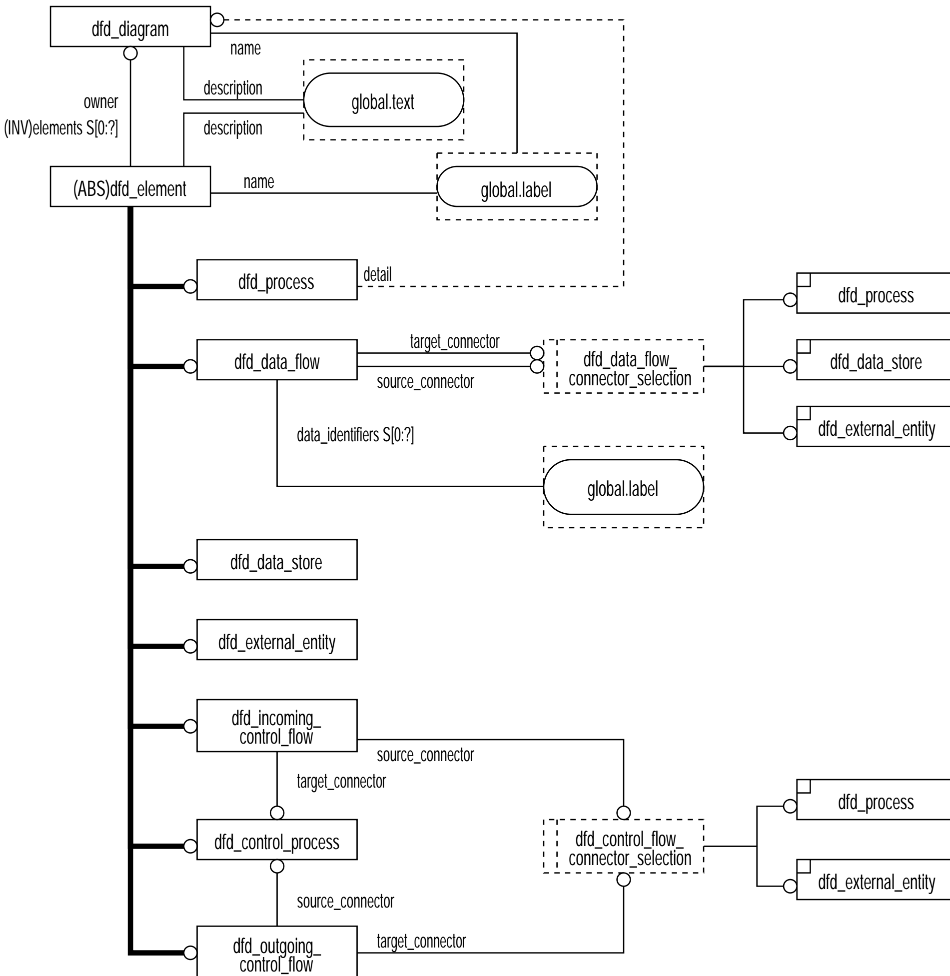

Fig. 4.8: Meta-model of data-flow diagrams

4.3.2 Data Dictionary

Figure 4.9 shows the meta-model for data dictionaries, which are described in Subsection 3.1.3. In the meta-model, the data dictionary itself is represented by the *dd_specification* entity. It contains different data dictionary elements, represented by the *dd_element* entity, which is associated through the *owner* relationship with the data dictionary it belongs to. The single elements used in a data dictionary are represented by the following entities.

dd_sequence. The *dd_sequence* entity represents a definition in a data dictionary consisting of a keyword and its associated definition in the form of a concatenated list. The keyword is stored in the *keyword* attribute, the definition consists of *dd_element* entities, i.e. an ordered concatenation of different data dictionary elements, represented by the *elements* relationship.

dd_selection. The *dd_selection* entity represents a selection, i.e. the choice of one element from a set of alternatives, where the alternatives are listed in the *elements* relationship.

dd_option. The *dd_option* entity represents an optional element, i.e. an element that may be omitted. The respective element is referenced by the *element* attribute.

dd_repetition. The *dd_repetition* element represents a repetition, where the repeated element is referenced through the *element* relationship and the number of repeats is stored in the *repeat_count* attribute. The repeat count can either be a natural number or special sign that indicates an undefined repeat count, which in turn means that the number of repeats is 0 or more.

dd_literal_element. The *dd_literal_element* entity represents literal elements in a data dictionary, i.e. elements built from basic characters and numbers. The associated textual representation of the literal element is stored in the *text* attribute.

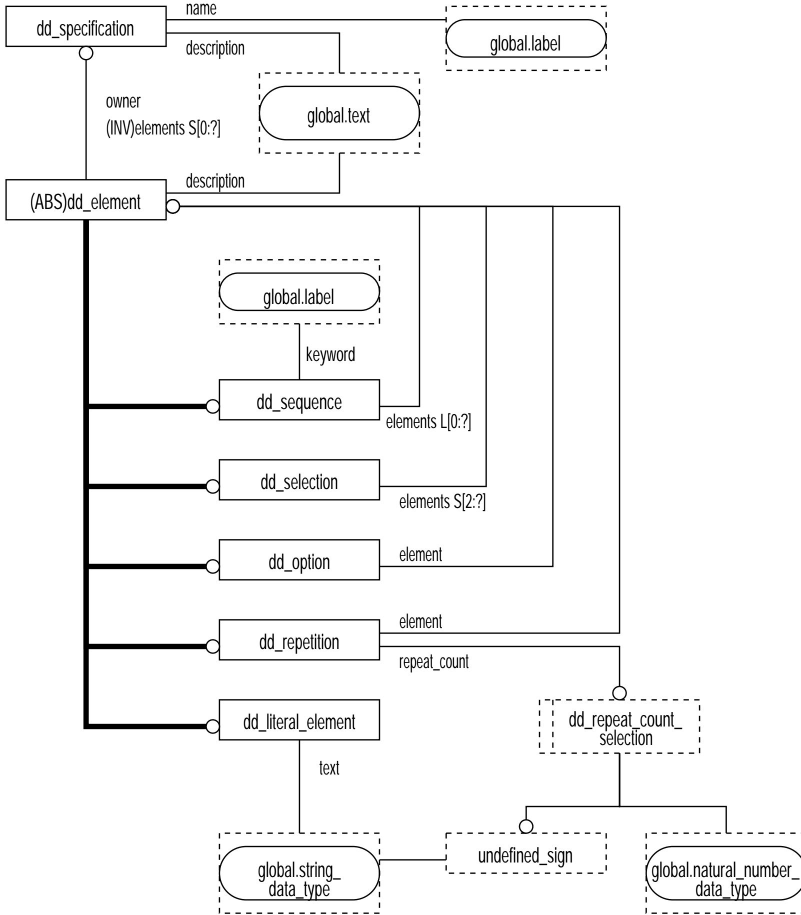

Fig. 4.9: Meta-model of data dictionaries

4.3.3 Entity-Relationship Diagram

Figure 4.10 shows the meta-model for entity-relationship diagrams, which were described in Subsection 3.1.4. In the meta-model, the entity-relationship diagram itself is represented by the *erd_diagram* entity. The ownership of the diagram elements is represented by the *owner* attribute of their common abstract superclass entity *erd_element*. The diagram elements are represented by the following entities:

erd_entity. The *erd_entity* entity represents an entity in an entity-relationship diagram. The name of the entity is stored in the *name* attribute of *erd_entity*, and its associated attributes are referenced through the *attributes* relationship. The attributes of an entity are represented by *erd_attribute* entities. The name of an attribute is stored in the *name* attribute of *erd_attribute*. The ownership of an attribute by an entity is referenced through the *owner* attribute of *erd_attribute*.

erd_relationship. The *erd_relationship* entity represents a relationship in an entity-relationship diagram. The related entities are referenced through the *source* and *target* relationships of *erd_relationship*. Note that this construct allows only binary relationships. N-ary relationships are not considered, as they can be viewed as a simple extension to binary ones. Associative entities may optionally be referenced through the *associative_entity* attribute.

erd_supertype_subtype_relationship. The *erd_supertype_subtype_relationship* entity represents a supertype / subtype relationship between entities. The respective supertype entity is referenced through the *super_type* attribute, whereas the associated subtypes are referenced through the *sub_type* attribute, which can hold a set of subtype entity references.

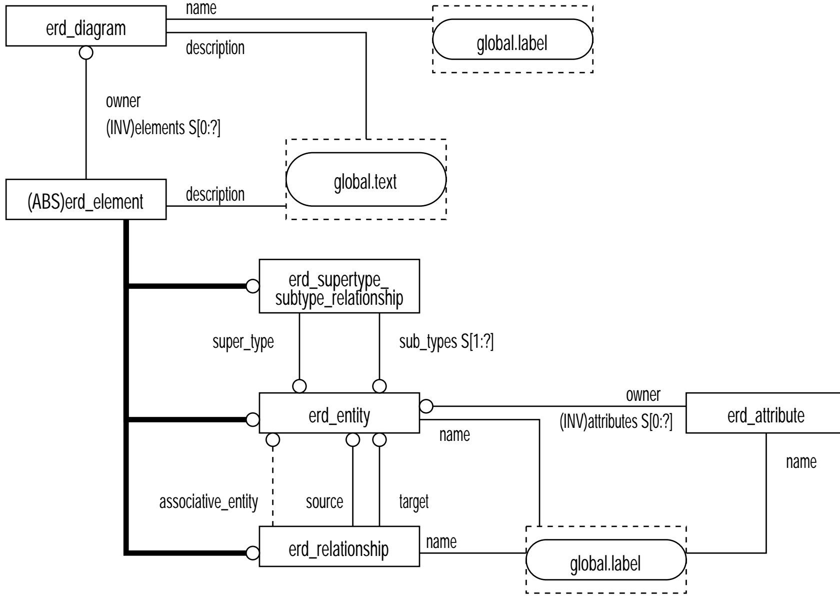

Fig. 4.10: Meta-model of entity-relationship diagrams

4.3.4 State-Transition Diagram

The meta-model for state-transition diagrams (described in Subsection 3.1.5) is shown in Figure 4.11. In the meta-model, the state-transition diagram itself is represented by the *std_diagram* entity. The abstract entity *std_element* is the superclass for all diagram elements and is associated with the diagram through the *owner* attribute. The diagram elements themselves are represented by the following entities:

std_state. The *std_state* entity represents a state. The name of the state is stored in the *name* attribute. The kind of state is stored in the *state_kind* attribute with the possible values *initial* for an initial state, *normal* for a normal (non-pseudo) state, or *final* for a final state. The internal behavior of a state can optionally be refined through additional state-transition diagrams, which is referenced through the *concurrent_substates* attribute. This allows a hierarchy of state-transition diagrams to be built up. Furthermore, this construct is used to model concurrent states, namely by modeling a state that consists of concurrent substates, referenced through the *concurrent_substates* attribute.

std_transition. The *std_transition* entity represents a transition between two states, whereas the source state is referenced through the *source_state* attribute and the target state through the *target_state* attribute. The event that triggers the transition is stored in the *event_expression* attribute, an optional guard can be stored in the *guard_expression* attribute. Furthermore, an optional action that is associated with the transition can be referenced through the *action* relationship.

std_action. The *std_action* entity represents an action that may be executed when a transition is taken. The name of the action is stored in the *name* attribute, the specification of the action, e.g. a pseudo-code specification, is stored in the *specification* attribute.

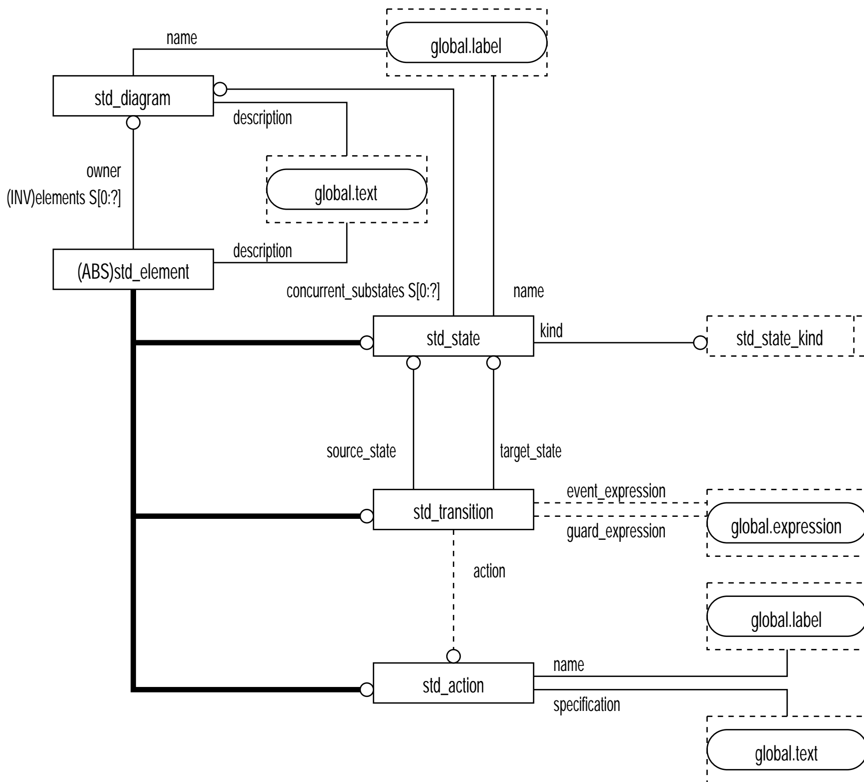

Fig. 4.11: Meta-model of state-transition diagrams

4.4 Meta-Models of Object-Oriented Techniques

The following subsections each describe a meta-model of one of the object-oriented specification techniques presented in Section 3.2. The meta-models presented herein are created on the basis of the descriptions in Subsection 3.2, i.e. the concepts of the specification techniques and their interrelationships. Each of the subsections presents only one possible way of meta-modeling the respective specification technique, alternative representations are possible. Due to the the fact that the object-oriented specification techniques in this thesis are based on the UML, the meta-models in this section have many similarities with the UML meta-model presented in [Gro01]. Furthermore, this section makes strict use of UML naming for object-oriented concepts, e.g. classifier, association or association class.

Due to the similarity with the UML, there are also in this section (like in the UML) types and concepts that are commonly used by several of the meta-models of the object-oriented specification techniques. The common types and concepts are presented in the Subsections 4.4.1, 4.4.2, and 4.4.3, followed by subsections each concerned with one of the selected specification techniques of Section 3.2.

4.4.1 Common Object-Oriented Types

The types commonly used by the meta-models in this Section are shown graphically in Figure 4.12. They are defined as follows:

oo_data_type_selection. The *oo_data_type_selection* type represents a logical grouping of elements that can serve as data-types. Besides the basic string and number data-types, it allows for selecting a classifier as type.

visibility_kind. The *visibility_kind* type represents the domain of values that describe the accessibility of model elements by other model elements. An instance of the *visibility_kind* type holds one of the following values:

- **public.** The respective model element can be accessed without limitations by other model elements.
- **private.** The respective model element can only be accessed by its owner or by model elements that are part of its owner's subclass hierarchy.

protected. The respective model element can only be accessed by its owner.

aggregation_kind. The *aggregation_kind* type represents the domain for values that describe the aggregation semantics of an association. An instance of this kind holds one of the following values:

none. The respective association is a normal association, i.e. it does not represent an aggregation in the form of a logical grouping or physical hierarchy.

aggregation. The respective association describes an aggregation, i.e. an "is a part of" relationship.

composition. The respective association describes a composition, i.e. the strong form of aggregation. If the whole is deleted, the parts are also deleted, and every part can only be part of one whole.

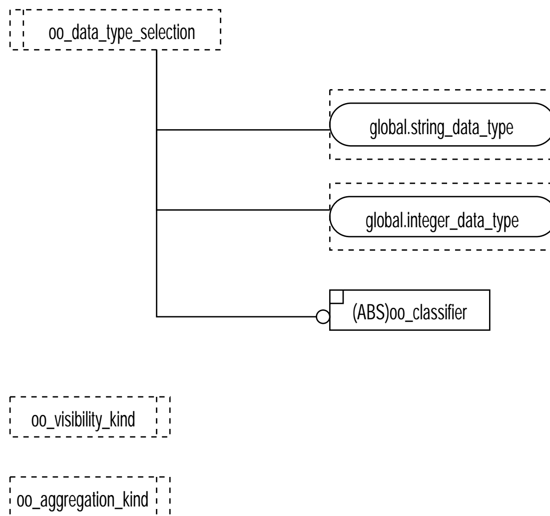

Fig. 4.12: Meta-model of common object-oriented types

4.4.2 Classifier

The meta-model of classifiers described in Subsection 3.2.2, is shown in Figure 4.13. Classifiers are represented by the *oo_classifier* entity and are instantiated as classes, actors, or use cases. They exhibit structural and dynamic features, which are represented by the *oo_feature* entity and its subclasses *oo_attribute* and *oo_operation*.

oo_attribute. The *oo_attribute* entity represents an attribute (a structural feature) of its owning classifier. The type of the attribute is referenced through the *data_type* relationship. Its cardinality, i.e. the number of values of the same data-type the attribute can hold, is stored in the *multiplicity* attribute.

oo_operation. The *oo_operation* entity represents an operation (a dynamic feature) of a classifier. The parameters of the operation are inversely referenced through the *parameters* attribute. An optional return value type of the operation is referenced by the *return_type* attribute. The specification, e.g. in the form of pseudo-code or source code, is stored in the *specification* attribute. The entity *oo_operation_parameter* represents an operation parameter. The operation that owns the operation parameter is referenced through the *owner* attribute, its data-type is specified through the *data_type* attribute, and its name is stored in the *name* attribute.

The name of a feature is stored in the *name* attribute of the *oo_feature* entity. Its visibility is stored in the *visibility* attribute. The owner of a feature is referenced through the *owner* attribute.

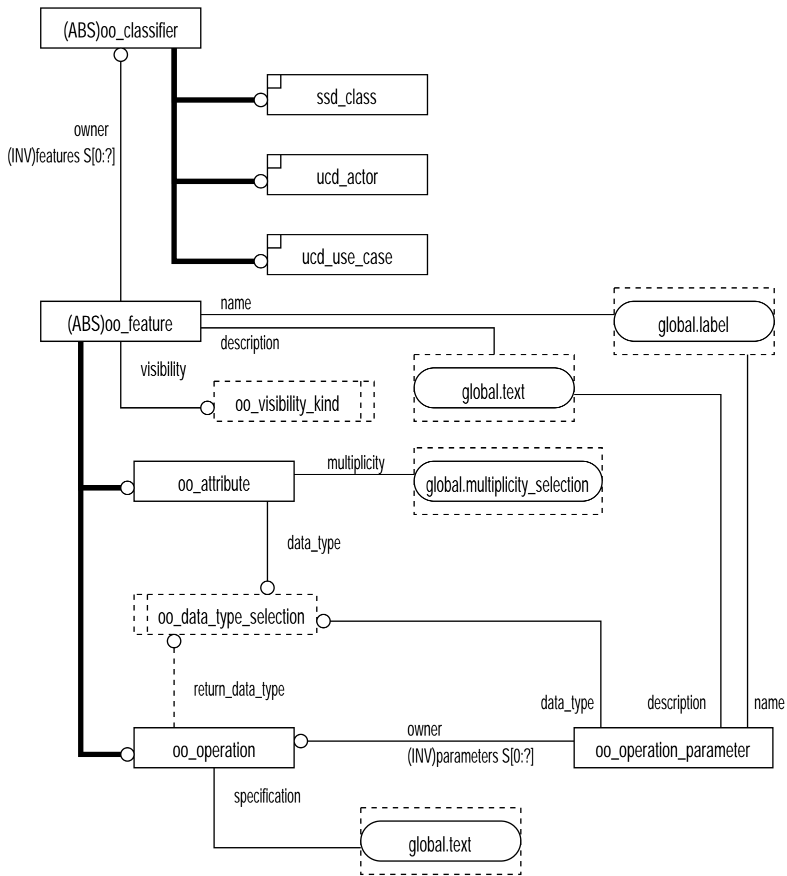

Fig. 4.13: Meta-model of classifiers

4.4.3 Relationships

The meta-models of relationships described in Subsection 3.2.2, are shown in Figure 4.14 and Figure 4.15.

Associations (Figure 4.14) are represented by the *oo_association* entity. In order to be able to distinguish associations in static structure diagrams from associations in use case diagrams, the subclasses *ssd_association* and *ucd_association* of *oo_association* are introduced. The name of an association is stored in the *name* attribute, and the reading direction of the name is encoded in the *reading_direction* attribute that references the "target" end of the association. Association ends are represented by the *oo_association_end* entity. The association that they belong to is referenced through the *association* attribute. The classifier participating in the association through the association end is referenced by the *participant* attribute; its role name is stored in the *role_name* attribute. The kind of association end, i.e. normal association or a kind of aggregation, is stored in the *aggregation* attribute and the cardinality of the classifier in the association is stored in the *multiplicity* attribute.

Generalizations (Figure 4.15) are represented by the *oo_generalization* entity, having a *parent* attribute referencing the more general element of the generalization, and a *child* attribute referencing the more specific element. Both elements can either be a classifier (i.e. a class, a use case, or an actor) or an association.

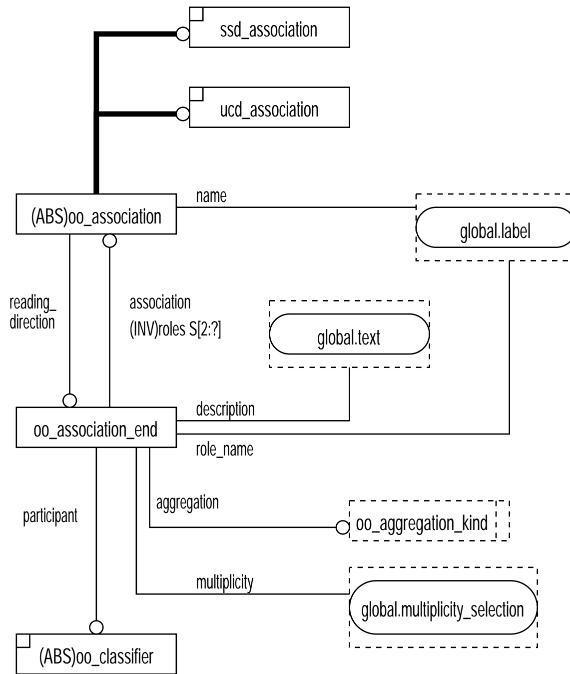

Fig. 4.14: Meta-model of associations

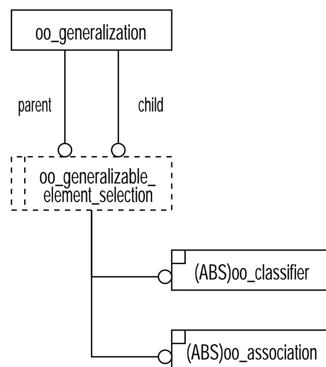

Fig. 4.15: Meta-model of generalizations

4.4.4 Static Structure Diagram

Figure 4.16 shows the meta-model of static structure diagrams, which are described in Subsection 3.2.3. A static structure diagram itself is represented by the *ssd_diagram* entity. The diagram elements are represented by the abstract *ssd_element* entity and its subclasses, whereby the name of the diagram element is stored in the *name* attribute. The actual diagram elements are represented by the following entities.

ssd_package. The *ssd_package* entity represents a package. The content of the package is described by another static structure diagram, which is referenced through the *content* attribute.

ssd_class. The *ssd_class* entity represents a class.

ssd_association. The *ssd_association* represents an association in a static structure diagram. The only difference to its superclass *oo_association* (see Subsection 4.4.3) is that it can be referenced by an association class.

ssd_association_class. The *ssd_association_class* entity represents an association class (a class associated with an association in order to store more information about an association). The association is referenced through the *association* attribute, the class (declared to be the association class) is referenced through the *class* attribute.

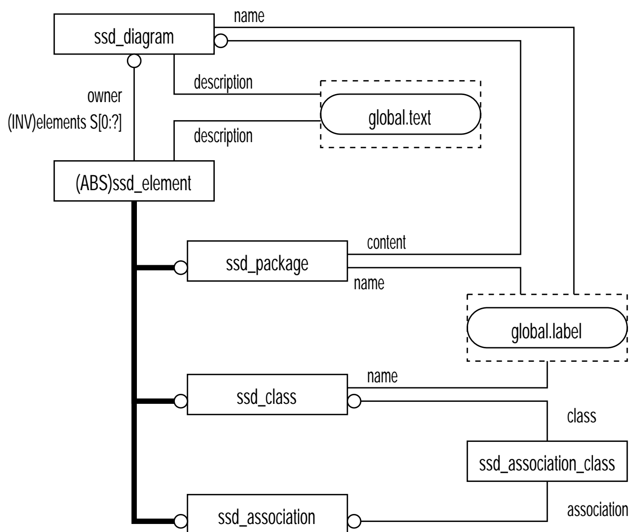

Fig. 4.16: Meta-model of static structure diagrams

4.4.5 Use Case Diagram

Figure 4.17 shows the meta-model of use case diagrams, which are described in Subsection 3.2.4. The use case diagram itself is represented by the *ucd_diagram* entity, the elements of the diagram are abstractly represented by the *ucd_element* entity. The actual diagram elements are represented by the following subclass entities:

ucd_actor. The *ucd_actor* entity represents an actor. The actor's name is stored in the *name* attribute.

ucd_association. The *ucd_association* represents an association in a use case diagram.

ucd_use_case. The *ucd_use_case* represents a use case in a use case diagram. The name of the use case is stored in the *name* attribute. A use case may be refined through another use case diagram, which is represented by the *detail* attribute. A use case may have several extension points that are represented by the *ucd_extension_point* entity. The name of the extension point is stored in the *name* attribute, the owning use case is referenced through the *use_case* attribute, and the location of the extension point within the use case is described by the *location* attribute. The inclusion and extension relationships among use cases are represented by the following entities:

ucd_inclusion. The *ucd_inclusion* entity represents an inclusion of one use case by another use case. The including use case is referenced through the *base* attribute, whereas the included use case is referenced through the *addition* attribute.

ucd_extension. The *ucd_extension* entity represents an extension of a use case. The extended use case is referenced through the *base* attribute, the respective extension point is referenced through the *extension_point* attribute, and the extending use case is referenced through the *extension* attribute.

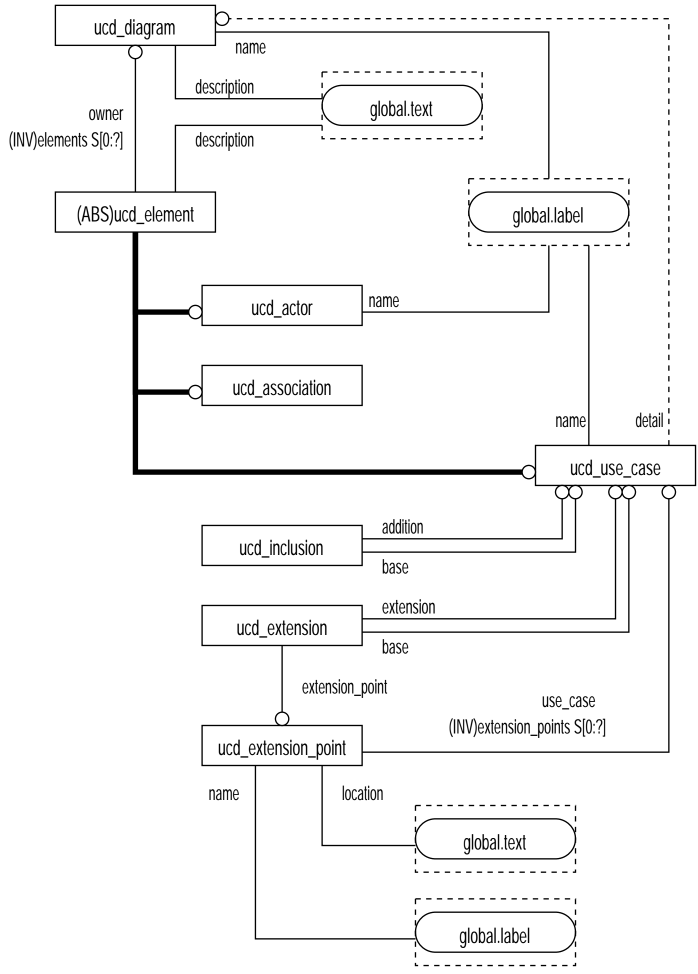

Fig. 4.17: Meta-model of use case diagrams

4.4.6 Collaboration and Sequence Diagram

Like in the UML, collaboration and sequence diagrams, which are described in Subsection 3.2.5 and 3.2.6, have a common meta-model, which is shown in Figure 4.18. The diagrams themselves are represented by the *cd_collaboration* entity, which implements either an operation or a use case diagram, referenced through the *implementing* attribute. The elements of a collaboration diagram are classifiers and associations playing a certain role in the collaboration, abstractly represented by the *cd_element* entity. The actual elements are represented by the following entities:

cd_association_role. The *cd_association_role* entity represents the role that an instance of an association (a link) plays in the collaboration. The instantiated association is referenced through the *association* attribute, the cardinality of the association role is stored in the *multiplicity* attribute.

cd_association_end_role. The *cd_association_end_role* entity represents the end of an association role (link end). It is an instance of an association end, which is referenced through the *association_end* attribute. The owning association role is referenced through the *association_role* attribute. The cardinality of the association end role in the context of the association role is stored in the *multiplicity* attribute.

cd_classifier_role. The *cd_classifier_role* entity represents the role that a classifier plays in a collaboration. The classifier is referenced through the *classifier* attribute; its cardinality within the context of the collaboration is stored in the *multiplicity* attribute.

Sequence diagrams and collaboration diagrams depict an interaction, which in turn actually instantiates a collaboration. Interactions are represented by the *cd_interaction* entity, which references its collaboration context by the *interaction_context* attribute. Messages between classifiers are represented by the *cd_message* entity, the sending classifier role is referenced through the *sender* attribute and the receiving classifier through the *receiver* attribute.

Note that the stimuli concept of the UML is not modeled in the meta-model for the purposes of this thesis, as only the elements that specify the abstract structure of the system and not its actual instantiation are considered. Furthermore, the kind of message, i.e. whether it is a synchronous message (wait on return) or an asynchronous message (no waiting on return, concurrent execution), is not distinguished, because there is no structural difference

between both in the diagrams. To support this, it would be enough to add a simple flag that indicates the synchronous or asynchronous character of the message, respectively.

The temporal ordering of messages, which is implicitly encoded in sequence diagrams, is captured by the *cd_message_temporal_relationship* entity by referencing a preceding message through its *predecessor* attribute and a succeeding message through its *successor* attribute.

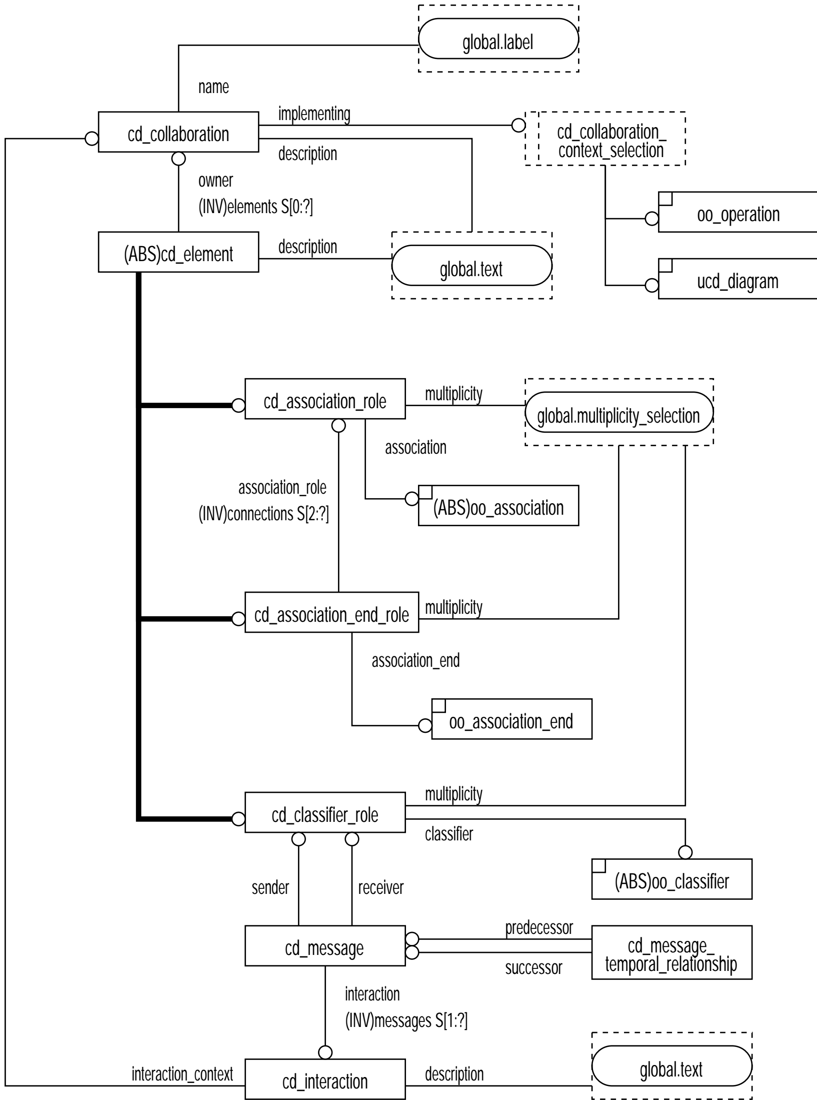

Fig. 4.18: Meta-model of collaboration and sequence diagrams

4.4.7 Statechart Diagram

The meta-model of statechart diagrams described in Subsection 3.2.7 is shown in Figure 4.19. The statechart diagram itself is represented by the *scd_diagram* entity. Its initial state is referenced through the *initial_state* attribute. The elements of a statechart diagram are abstractly represented by the *scd_element* entity, their ownership allocation to the statechart diagram is defined in the *owner* attribute. The actual elements of a statechart diagram are represented by the following entities.

scd_generic_state. The *scd_generic_state* entity abstractly represents the following different kinds of states.

- **scd_initial_state.** The initial pseudo-state of a statechart diagram is represented by the *scd_initial_state* entity.
- **scd_final_state.** The *scd_final_state* entity represents a final pseudo-state of a statechart diagram.
- **scd_state.** The *scd_state* entity represents a normal (non-pseudo) state, its name is stored in the *name* attribute. Furthermore, the subclass entity *scd_concurrent_composite_state* represents a composite state that is refined by two or more concurrent state machines, referenced through the *substates* attribute.

scd_transition. The *scd_transition* entity represents transitions between two states. The source and target states of the transition are referenced through the *source_state* and *target_state* attributes, respectively. The name of the event that triggers the transition is stored in the *event* attribute, an optional guard expression is stored in the *guard* attribute, and an optional action associated with the transition is referenced through the *action* attribute.

The *scd_action* entity represents actions that may be executed upon the execution of a transition. The name of the action is stored in the *name* attribute, its specification is referenced through the *specification* attribute.

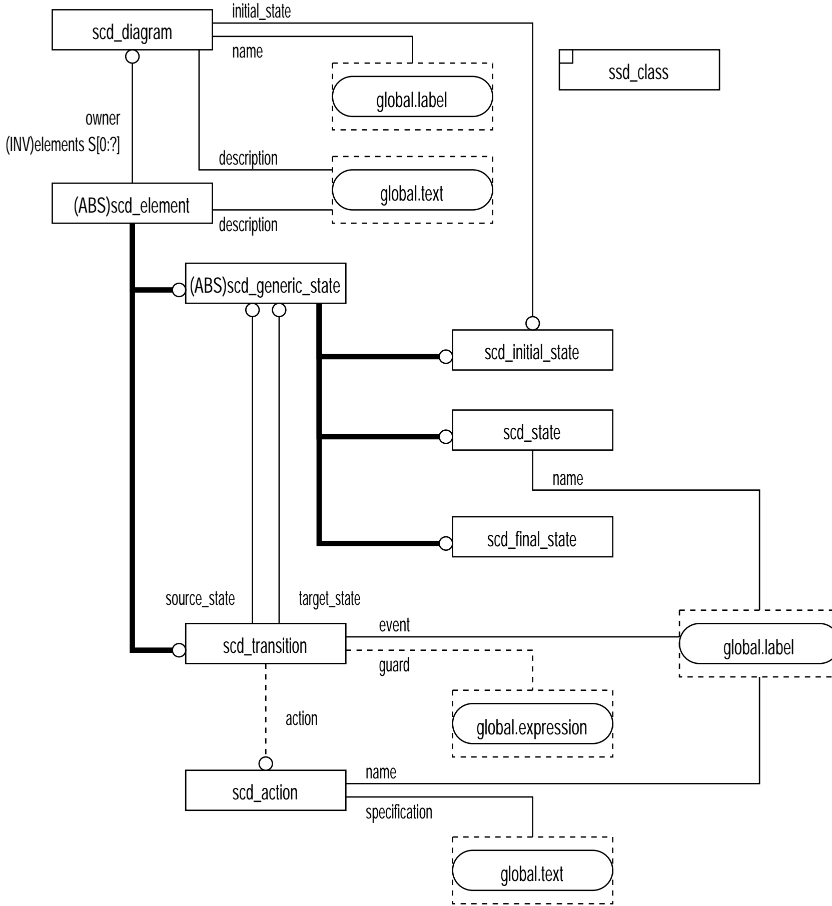

Fig. 4.19: Meta-model of statechart diagrams

5. CREATING THE COMMON META-MODEL

In this chapter, a common meta-model is synthesized from the meta-models presented in Chapter 4. First, the general principles of unifying semantically matching concepts in the common meta-model are explained. Second, the principles are applied within each of the three modeling aspects (data, function and behavior). Note that the common meta-model presented herein is only one possible way of modeling an integrated meta-model of the meta-models presented in Chapter 4.

5.1 Integration Principles

As outlined in the approach overview in Subsection 1.4.1, the elements of the common meta-model are based on the elements of the underlying specification techniques. For this reason, the elements of the considered specification techniques need to be examined with respect to their semantics. For similar elements, a common super-element is created, integrating the semantics of the constituting elements. Summarized in one model, the super-elements form the common meta-model of the underlying specification techniques.

With respect to their semantics, two elements $a \in A$ and $b \in B$, where A and B each represent the set of elements of a specification technique, are either semantically synonymous, including, partial conform, or disjoint. Figure 5.1 illustrates these four relationships, using a dashed circle for depicting the specification domain of element a and a dotted circle for the one of b. The specification domain of an element describes the subset of all possible specification aspects that the element provides support for.

The following paragraphs describe the four possible relationships between element a and b in more detail.

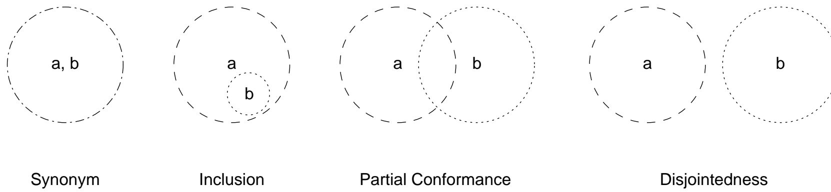

Fig. 5.1: Cases of semantic matching

Synonym. If element a and element b support the same specification domain, then the elements are synonymous, element a is a synonym for element b and vice-versa. A super-element in the common meta-model that is based on synonymous elements allows for full transformations of representations of the synonymous elements between the underlying specification techniques.

Inclusion. If the specification domain of element a includes the specification domain of b, then element a includes b, and element b is contained in element a. A super-element in the common meta-model that is based on such an inclusion, allows for a full transformation of the common meta-model representation into the representation using the including element. However, it permits only a partial transformation of the respective common meta-model element representation into a representation of the included element.

Partial Conformance. If the specification domain of element a and the specification domain of element b overlap partially, then element a is partially conform to element b and vice-versa. The specification domain of a super-element in the common meta-model that is based on a partial conformance, is the union of the specification domains of the constituting elements. Such super-elements allow only for partial transformations between representations of the partially conform elements, namely in the area of their overlapping specification domains.

Disjointedness. If the specification domains of element a and element b do not overlap, then the elements are disjoint. In this case, a super-element in the common meta-model need not be created. Thus, direct transformations between representations of the elements are not supported.

In summary, super-elements in the common meta-model are only created from elements with overlapping specification domains. If elements have

different but intersecting specification domains, their unification as super-element is semantically "richer" than its constituents (as described under "inclusion" and "partial conformance"). This means that the super-element makes it possible to specify a broader or more detailed part of the system specification than each of the constituting elements. This, in turn, implies that a representation under the common meta-model may not be fully transformable into representations of the constituting specification techniques. A super-element cannot be created for specification elements with disjoint specification domains. In order to identify disjoint elements and hence, in order to reduce the number of comparisons of specification elements, the fact that specification techniques usually only specify a certain aspect of a system can be utilized. For example, entity-relationship modeling primarily focuses on data items, it does not provide support for modeling system functions or system behavior. Hence, it makes sense to only compare elements of specification techniques within one specification aspect. Therefore, the specification techniques are grouped into the following three different aspects of modeling.

Data Aspect. The data aspect considers data items, the static structure and static relationships among data items in the form of hierarchical data structures.

Functional Aspect. The functional aspect considers the functionality of the system under specification and how it is distributed among elements of the system.

Behavioral Aspect. The behavioral aspect considers details of system functions and illustrates how the system reacts to different kinds of events.

The specification techniques examined in Subsections 3.1.1 and 3.2.1, can be assigned to these aspects as shown in Table 5.1. Note that the grouping in Table 5.1 is not strict, as for example data-flow diagrams do not purely model functions but also data in their data-flows and data stores. However, only the primary modeling aspect of the respective specification technique has been taken into account, i.e. modeling of functions with data-flow diagrams.

As described in Section 2.1, the object-oriented specification techniques give no preference to either the data aspect or the function aspect, rather they consider both to be of equal importance. Hence, an object-oriented specification technique, such as the static structure diagram of the UML, can be used to model both the data aspect and the functional aspect, as shown in Table 5.1. In order to bring semantically matching elements (elements

Tab. 5.1: Specification techniques and their modeling aspects

Data Aspect	Functional Aspect	Behavioral Aspect
Data Dictionary, Entity-Relationship Diagram	**Data-Flow Diagram**	**State-Transition Diagram**
Common Object-Oriented Elements, Static Structure Diagram	Static Structure Diagram, Collaboration / Sequence Diagram	Collaboration / Sequence Diagram, Statechart Diagram

with overlapping specification domains) down to a common denominator, the semantically "richest" specification technique, i.e. the one that provides the most detailed support for a specific aspect, is taken as a basis. In Table 5.1, these base specification techniques are highlighted within each aspect. The elements of the respective other specification techniques are then integrated with the elements of the base specification technique, building a set of super-elements, summarized in the common meta-model.

The following sections present the common meta-model synthesized from elements of the meta-models presented in Chapter 4. Each section considers one of the aspects presented above: The data aspect in Section 5.3, the functional aspect in Section 5.4, and the behavioral aspect in Section 5.5. The semantic matching of elements within one aspect (see Table 5.1) has been examined according to the matching cases described above, i.e. whether elements are synonymous, including, partial conform, or disjoint. The actual mappings of each of the elements of the specification techniques to elements of the common meta-model are described and motivated in detail in Chapter 6.

Note that the resulting common meta-model, like the meta-models of the specification techniques presented in Chapter 4, is not mathematically derived but manually generated. The common meta-model described in this chapter represents only one possible result of building a common meta-model from the single meta-models described in Chapter 4.

The aspects presented make use of common elements, which are summarized and explained in the following general section. In the subsequent sections about the three aspects (data, function and behavior), these commonly used elements are not repeatedly explained. Note that the next section describes

constructs of the common meta-model, which are not derived from the constituting meta-models of the specification techniques. These additions have been made in order to provide a model management that makes it possible to describe inter-relationships between different models and diagrams of a system specification. This is useful for creating a global system specification from single models if the common meta-model is implemented as a central repository between different specification tools, as proposed in Subsection 1.4.1, Figure 1.2 and implemented as described in Chapter 7.

5.2 General Concepts of the Common Meta-Model

This section presents general concepts that are used by several aspects of the common meta-model.

5.2.1 Types

The basic types of the common meta-model are shown in Figure 5.2, being direct replacements (aliases) for the basic EXPRESS types *BOOLEAN*, *STRING*, and *INTEGER*. Note that the set of basic types of the common meta-model is kept small in order not to overload the model with unnecessary detail. However, the principles applied to the basic data-types can also be applied to additional and more complex data-types.

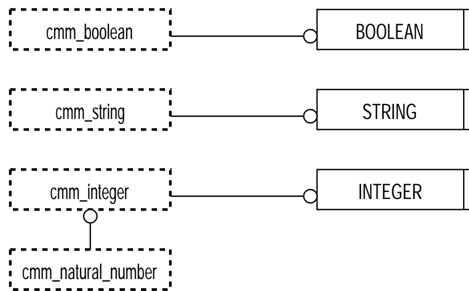

Fig. 5.2: Common meta-model of basic types

Figure 5.3 shows the more specific data-types of the common meta-model, which are the following.

Label. A *cmm_label* represents a name given to a model element.

Description. A *cmm_description* represents a textual format-free description of an associated model element.

Boolean expression. A *cmm_boolean_expression* represents a textual expression that can be evaluated to a boolean value.

Textual specification. A *cmm_textual_specification* represents a textual specification that uses a specific language. The name of the used language is stored in the *language* attribute, the specification is stored in the

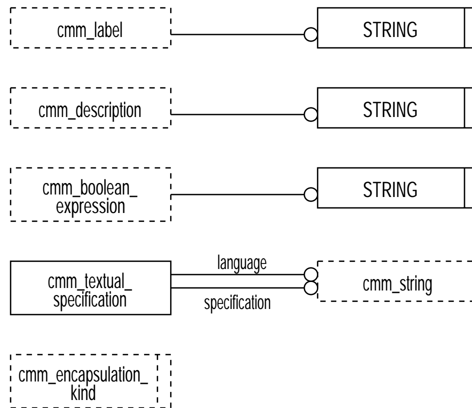

Fig. 5.3: Common meta-model of specific types

specification attribute. This construct is used for example for storing program code or pseudo-code specifying a function of a system.

Encapsulation kind. The data-type *cmm_encapsulation_kind* represents an enumeration of possible values (public, private, protected) that describe the visibility of an associated model element to other elements. This construct is equivalent to the object-oriented visibility kind, see *visibility_kind* in Subsection 4.4.1. Note that the enumeration values are not shown in EXPRESS-G representations.

The meta-model representation of cardinalities is shown in Figure 5.4. In the common meta-model, cardinalities are used to represent ranges of data-types, e.g. an array of a certain data-type. Furthermore, cardinalities are also employed to describe the range of valid numbers of instances in relationships, e.g. in a 1-to-1 or a 1-to-n relationship. The general cardinality is represented by the *cmm_cardinality* type, which allows for describing a cardinality as either a single cardinality (represented by the *cmm_single_cardinality* type) or a range of single cardinalities, where the range is described through an upper and a lower bound. Infinite cardinalities can be represented by using the *cmm_infinity* sign instead of a natural number describing discrete cardinalities.

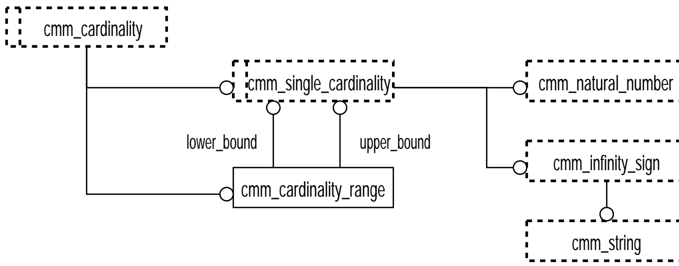

Fig. 5.4: Common meta-model of cardinality

5.2.2 Views

The meta-models presented in Chapter 4 all contain a concept that binds elements of one diagram (or specification fragment) together to a logical group. The common meta-model provides the equivalent concept of a view for logically grouping specification elements. Thus, a view represents a particular part of the system under specification, and is usually modeled using a diagram of a particular specification technique.

Figure 5.5 shows the support for views in the common meta-model. The central element is the view, represented by the abstract *cmm_view* entity. Elements in a view are linked to the view through a *cmm_view_element_role* entity that represents a role of an element in a certain view. The corresponding element is referenced through the *element* attribute, and the view that provides the context for the role is referenced through the *role_context* attribute. This construct allows the same element to be used in several views, in contrast to a direct ownership of an element by a single view. A view accesses its elements through the inverse relationship *elements*. Elements of a view are either classifications (represented through the *cmm_classification* entity) or relationships (represented through the abstract *cmm_relationship* entity and its subclasses). Views are instantiated in one of the following forms.

Module. A module is similar to the object-oriented concept of a package, see Subsection 3.2.3. A module is used to logically group model elements that belong to the same specification of a structural aspect of the system under consideration. For example, an entity-relationship diagram is represented as a module. In the common meta-model, modules are represented by the *cmm_module* entity.

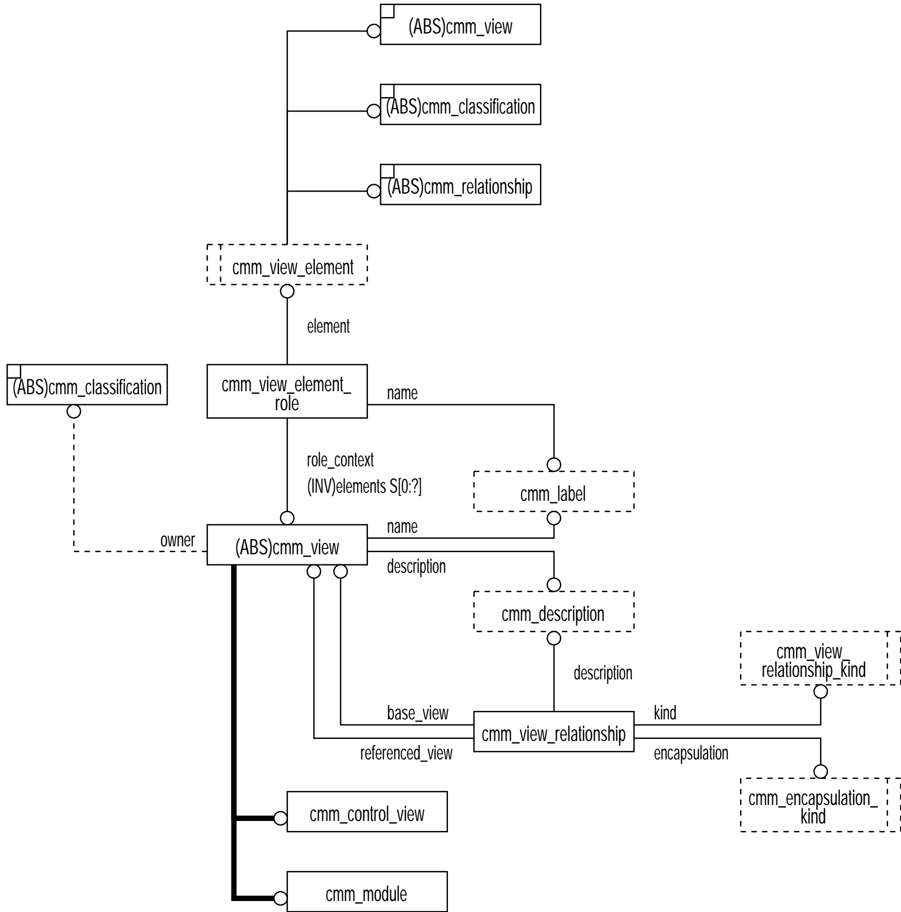

Fig. 5.5: Common meta-model of views

Control View. A control view represents a grouping of model elements that describe a dynamic aspect of the system under consideration. For example, state-transitions diagrams are represented as control views. In the common meta-model, control views are represented by the *cmm_control_view* entity.

Views can be inter-related in order to model view hierarchies (a composition of more general views from more specific views) or to express synonymous, concurrent or instantiating relationships. The synonymous type of relationship between two views means that the views describe the same part of a specification, only in a different way. The concurrent relationship type connects executable views (i.e. control views) that are executed concurrently. The instantiating relationship type describes the relationship between an abstract view and one of its instantiations. Such relationships are represented by the *cmm_view_relationship* entity, whereby one view is referenced through the *base_view* attribute and the related view is referenced through the *referenced_view* attribute. The kind of relationship (specialization, synonym, concurrent, or instantiation) is stored in its *kind* attribute. Furthermore, views can act as a detailed specification of a classifier, e.g. to describe the behavior of a class in a statechart diagram. In this case, the respective classifier is referenced through the optional *owner* attribute.

5.3 Data Aspect

The basis for the data aspect of the common meta-model are the common object-oriented structures and the elements from the static structure diagrams, as presented in Table 5.1. These are classifiers and relationships in general (see Subsection 3.2.2) and packages, classes and associations in specific (see Subsection 3.2.3). In general, the principal elements of data aspect modeling techniques are data items and their inter-relationships. In the UML, this is represented by classes and associations, in data dictionaries this is represented by keywords and aggregations, and in entity-relationship modeling this is represented by entities and relationships. These fundamental elements of the data aspect modeling techniques are reflected in the *cmm_classification* and *cmm_relationship* entities, shown in Figures 5.6 and 5.7 respectively.

5.3.1 Classifications

Figure 5.6 shows the common meta-model of the object-oriented classifiers (see Subsection 3.2.2), the data dictionary keywords, sequences, selections, and repetitions (see Subsection 3.1.3), and the entity-relationship modeling entities (see Subsection 3.1.4). These concepts are unified in the classification concept, which is represented by the abstract *cmm_classification* entity. A classification is instantiated as either a classification definition (*cmm_classification_definition*), or as an alias referencing a classifier definition.

The central element in Figure 5.6 is the *cmm_classification_definition* entity, which roughly resembles the object-oriented concept of a classifier (see the classifier meta-model in Subsection 4.4.2). A *cmm_classification_definition* may have variables associated with it (represented through the inverse *properties* attribute) that have their scope within the classification. Such variables are called properties and are represented by the *cmm_property* entity. Properties are inherited from generic variables and hence, are also of a certain data-type, referenced through the *data_type* attribute of the *cmm_generic_variable* entity. The data-type is either a simple data-type, i.e. *cmm_boolean* or *cmm_string* or *cmm_integer*, a *cmm_classification*, or a selection of data-types (*cmm_data_type_selection*). The latter construct is motivated by the data-type selection capability in data dictionaries, see Subsection 3.1.3, that represents a set of alternative data-types. Classifi-

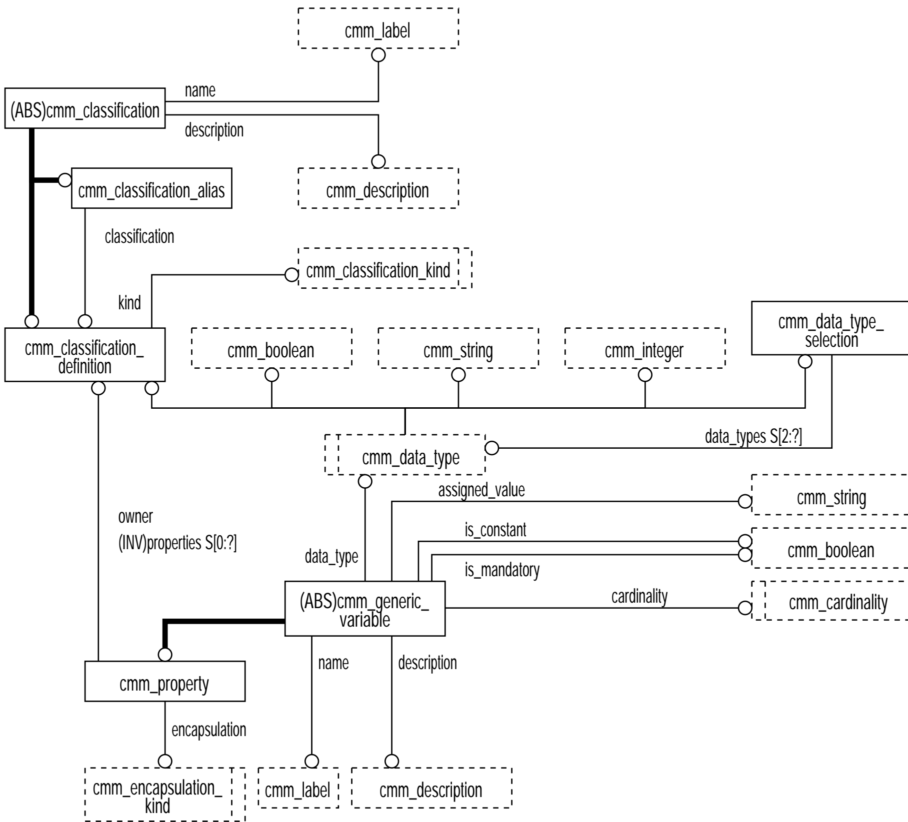

Fig. 5.6: Common meta-model of classifications (data aspect only)

cation definitions can be distinguished by the following values of their *kind* attribute.

General. The classification is not of a particular kind. This is used for classifications that are "normal" constituting elements of the specification, i.e. they are necessary for the specification and their internals may be further refined.

External. The classification is external to the specification. Only the participation of the classification in relationships, i.e. its input / output, is modeled. The internal structure of the classification is not further specified.

Functionality. The referenced classification can be briefly described as a grouping of functions. This kind of classification is explained in more detail together with the functional modeling aspect in Subsection 5.4.

Controller. The referenced classification describes the behavior of the system (or a part of it) and represents a controller that activates / deactivates functions of the system due to incoming events. This classification kind is explained in more detail with the behavioral modeling aspect in Subsection 5.5.

If a classification takes a certain role in a specific context, the role can be described using the *cmm_classification_alias* entity. This construct makes it possible to describe different roles of a classification in several contexts without having to completely specify the classification for each role. The aliased classification definition is referenced through the *classification* attribute of *cmm_classification_alias*.

5.3.2 Relationships

Figure 5.7 shows the common meta-model of structural relationships, based on the object-oriented meta-model of relationships presented in Subsection 4.4.3. It unifies the concepts of object-oriented relationships (see Subsection 3.2.2), the data dictionary aggregations (see Subsection 3.1.3), and the relationships of entity-relationship modeling (see Subsection 3.1.4).

The central element of the common meta-model for relationships is the abstract entity *cmm_relationship*. It represents a generic relationship between two classifications that take a certain role in the relationship. The *source* and *target* attributes of *cmm_relationship* refer to a *cmm_role* entity, which

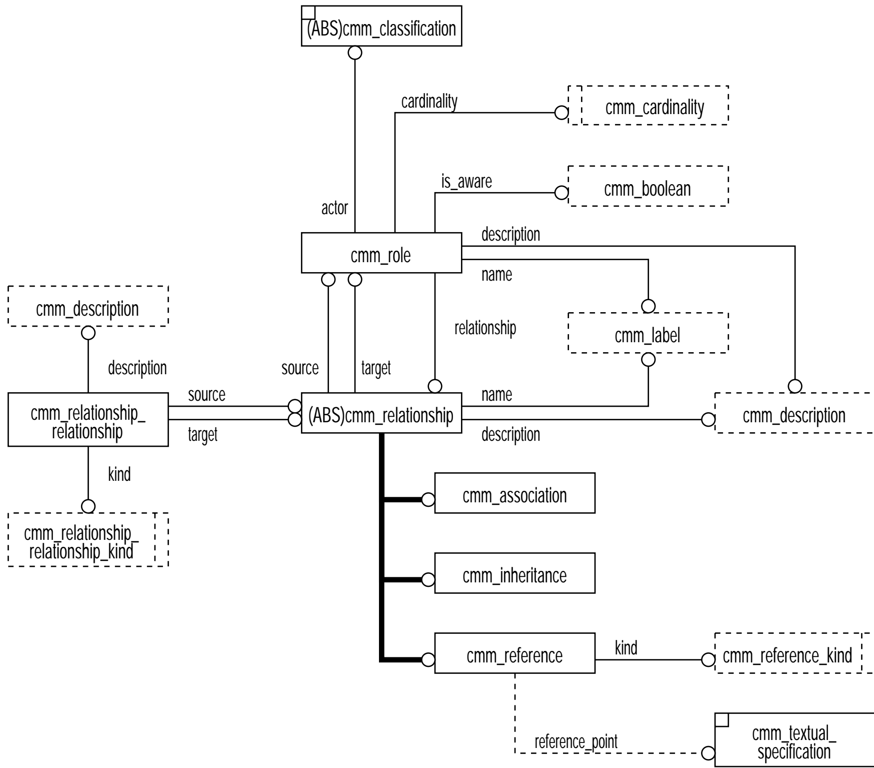

Fig. 5.7: Common meta-model of relationships (data aspect only)

in turn references the respective *cmm_classification* entity through its *actor* attribute. The role of a classification in a relationship (represented through the *cmm_role* entity) also stores whether the classification is aware of playing a role in the relationship or not (in its *is_aware* attribute). If a classification is aware of playing a role in a relationship, then it can access the related classification through the role at the other end of the relationship. Speaking in terms of data structures, this means that the role name appears to be like another attribute for the classification. The relationship in which the classification plays the respective role is referenced through the *relationship* attribute of *cmm_role*.

Furthermore, logical relationships among relationships themselves can be represented by the *cmm_relationship_relationship* entity. Such a construct is for example necessary for representing the relations between associations (in static structure diagrams) and association roles (in collaboration diagrams), which are instantiations of associations. The related relationships are referenced through the *source* and *target* attribute. The kind of relationship is stored in the *kind* attribute, which either holds *detail*, indicating that the target relationship is a refinement of the source relationship, or *instantiation*, indicating that the source relationship is a more abstract relationship and that the target relationship is an instantiation of the source relationship.

As the *cmm_relationship* entity represents only the abstract structure of a relationship, relationships are instantiated through one of the following subclasses of *cmm_relationship*.

Association. An association represents a semantic relationship between classifications. It resembles the object-oriented association concept, described in Subsection 3.2.2, and also represents the relationship concept of entity-relationship diagrams. The association is represented by the *cmm_association* entity.

Inheritance. An inheritance relationship represents the taxonomic relationship between a more general and a more specific classification. It resembles the object-oriented generalization concept, described in Subsection 3.2.2, and is represented by the *cmm_inheritance* entity.

Reference. A reference relationships represents a dependency between classifiers. It can be of one of two kinds. First, indicating the use of a classification by another, and second, indicating the extension of a classification through another. References are represented by the *cmm_reference* entity, whereby the kind of reference is stored in its

kind attribute (as values *uses* or *extends* as explained above). Furthermore, the exact reference point may be described through a textual specification which can be stored in the *reference_point* attribute.

The special case of a classification attached to an association in order to store more information about an association represented in the common meta-model as shown in Figure 5.8. This construct represents the unification of the object-oriented association class (see Subsection 3.2.3) and the associative entity indicator from entity-relationship modeling (see Subsection 3.1.4). The entity *cmm_association_classification* connects an association with the classification that is used to store additional data of the association. In this case, the referenced classification does not own dynamic features (i.e. functions). The role of the referenced classification in the association can be stored in the *role* attribute of *cmm_association_classification*.

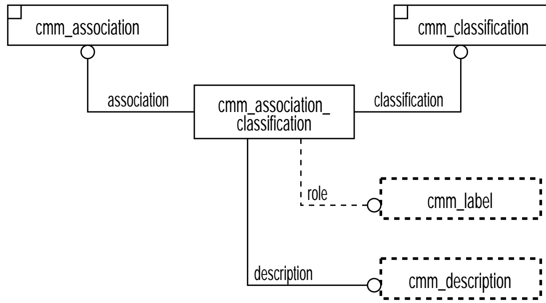

Fig. 5.8: Common meta-model of association classifications

5.4 Functional Aspect

As explained in Section 5.1, the functional aspect of the common meta-model is for the greater part based on the concepts of data-flow diagrams, described in Subsection 3.1.2. The main elements of data-flow diagrams are processes and data-flows. The control-oriented elements (control process and control data-flow) are incorporated in the constructs of the behavioral aspect, see Section 5.5. The functional aspect also covers elements from the object-oriented static structure diagrams, namely the behavioral features in the form of operations (see Subsection 3.2.2), which are also used in collaboration and sequence diagrams (see Subsections 3.2.5 and 3.2.6, respectively).

5.4.1 Functional Extension to the Classification Concept

Figure 5.9 illustrates the extensions to the classification concept of the common meta-model presented in Subsection 5.3.1 that are necessary to support the functional specification aspect.

The *cmm_function* entity represents a function of the system. It is owned by a classification that is referenced through the *owner* attribute. The textual specification of the function can be stored in the *specification* attribute, e.g. as source code of a specific programming language. Parameters of functions are represented by the *cmm_function_parameter* entity. They are implemented as a sub-class of *cmm_generic_variable*, inheriting its attributes (see Subsection 5.3.1 for a description), e.g. the associated data-type of the parameter. The affiliation of a function parameter with its function is modeled through the *owner* attribute of *cmm_function_parameter*. Inversely, the access of the function to its parameters is realized through the inverse *parameters* attribute of *cmm_function*. In the common meta-model, parameters can be of one of the following kinds.

In. The respective parameter is an input parameter of the function. This resembles the by-value parameters of functional programming languages.

Out. The respective parameter is an output parameter of the function.

InOut. The respective parameter is an input and output parameter of the function. This resembles the by-reference parameters of functional programming languages.

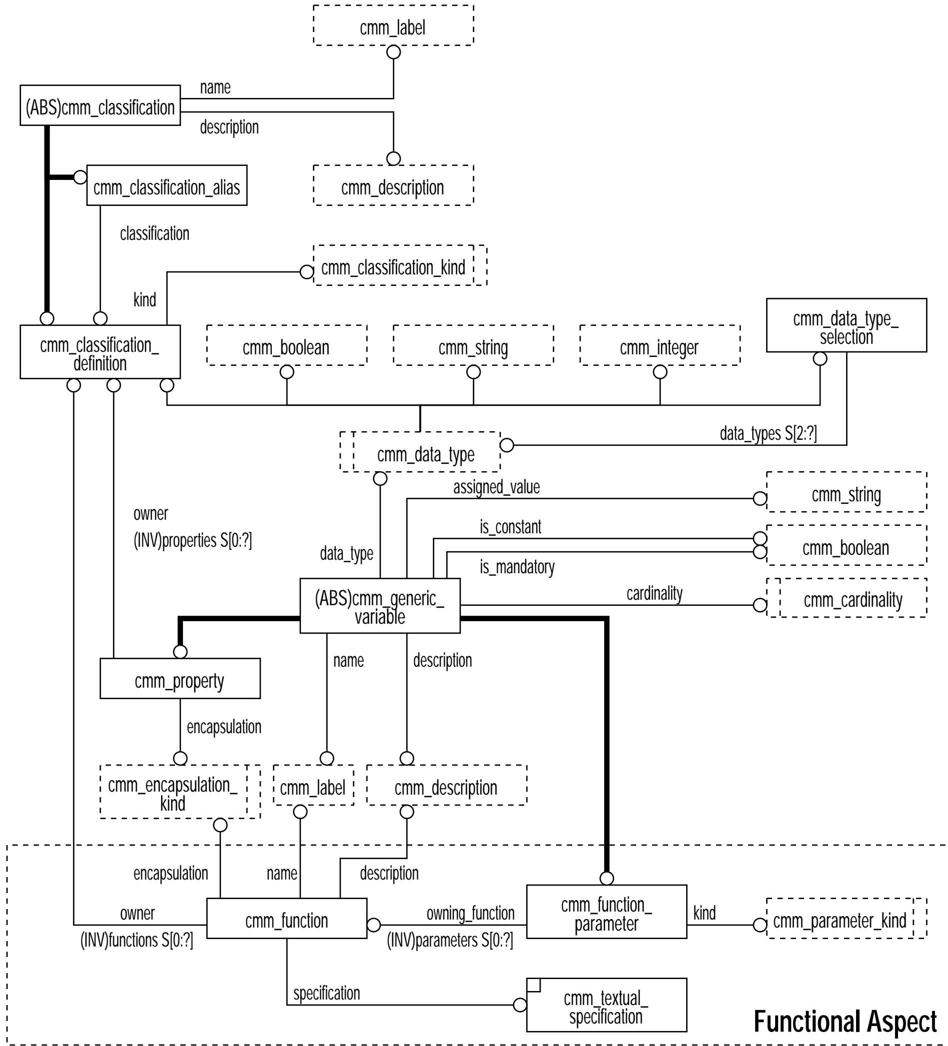

Fig. 5.9: Common meta-model of classifications (data and functional aspect only)

Return. The respective parameter is an output parameter of the function, which is associated with the function's one-dimensional value.

If a classification represents mainly a grouping of functions, the *kind* attribute of the respective *cmm_classification* entity should be given the value *functionality*, see Subsection 5.3.1. A classification has access to its functions through the inverse *functions* attribute.

5.4.2 Functional Extension to the Relationship Concepts

The support for structural relationships described in Subsection 5.3.2 is also sufficient for the functional modeling aspect. Hence, no extensions of the meta-model for relationships is necessary.

5.5 Behavioral Aspect

As presented in Section 5.1, the behavioral aspect of the common meta-model is based on the concepts from state-transition diagrams (see Subsection 3.1.5), statechart diagrams (see Subsection 3.2.7, and also on concepts from the object-oriented collaboration and sequence diagrams (see Subsections 3.2.5 and 3.2.6, respectively). The principal elements used for modeling behavioral aspects are the classifications and the states (as subclass of classifications), the transitions between classifications (and states), and the events triggering the transitions. In state-transition diagrams (see Subsection 3.1.5) and the object-oriented statechart diagrams (see Subsection 3.2.7), this resembles the notions of states, transitions and events. In the object-oriented collaboration (see Subsection 3.2.5) and sequence diagrams (see Subsection 3.2.6), this represents objects or classes and messages between them.

5.5.1 Behavioral Extension to the Classification Concept

For the behavioral aspect, the common meta-model is extended by the concept of a state as subclass of a classification as shown in Figure 5.10. States are represented by the *cmm_state* entity. It references the elements that are active while the state itself is active through its *active_elements* attribute. This construct is necessary for supporting the concept of activations in sequence diagrams (see Subsection 3.2.6). The active elements are either a function of a classification (*cmm_function*) or a classification as a whole (*cmm_classification*) without specifying the active function of the classification. The kind of state is described by the *state_kind* attribute with one of the following values.

Initial. The state is a pseudo-state (a state that the system cannot actually be in) indicating the starting point of the state machine. Initial states have no further internal structure or behavior specified.

Normal. The state is a normal (non-pseudo) state.

Final. The state is a pseudo-state indicating the termination of the state machine execution. Final states have no further internal structure or behavior specified.

A state may be specified in more detail through a number of concurrent control views (see Subsection 5.2.2). This is modeled through the *concur-*

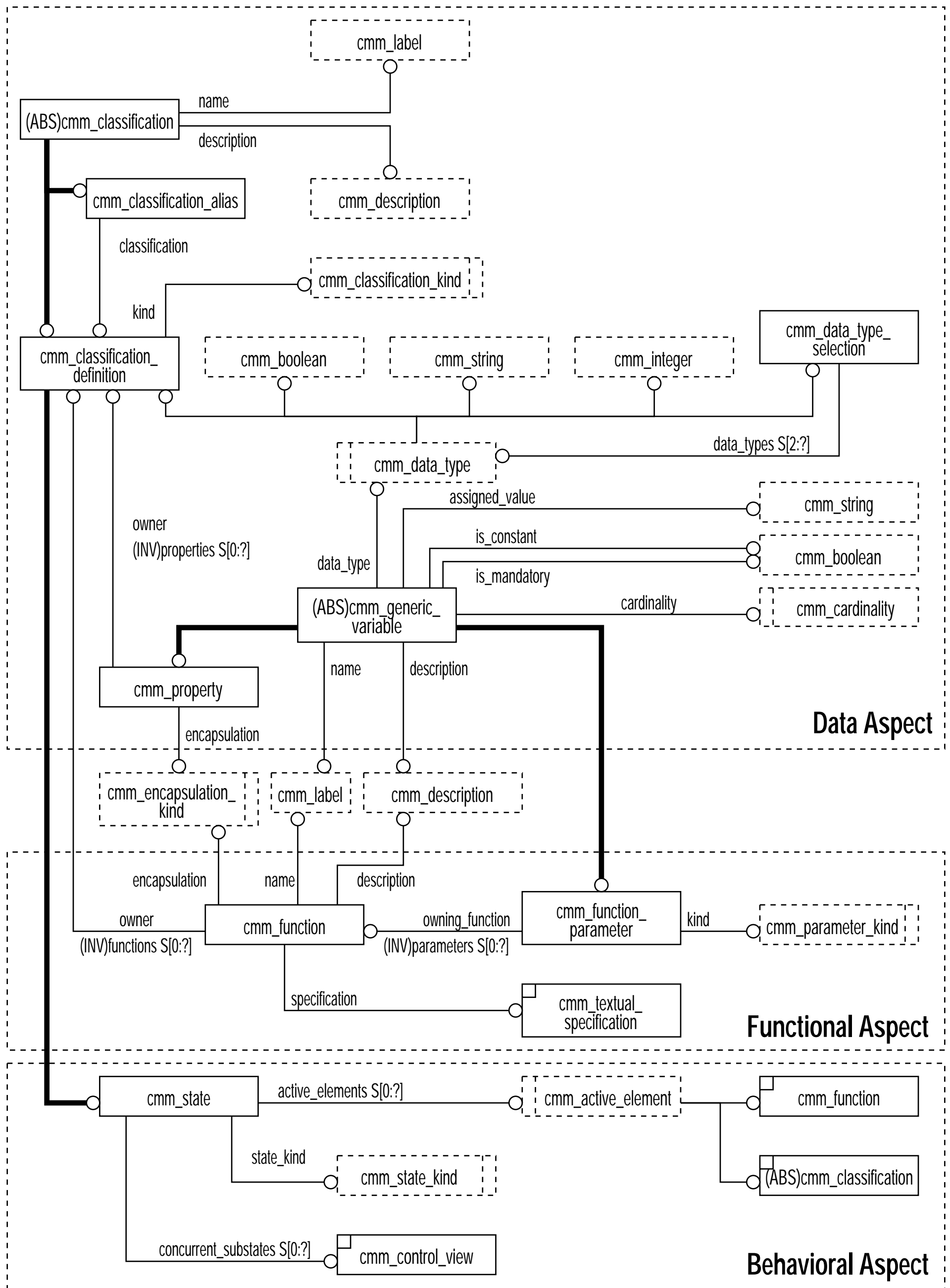

Fig. 5.10: Common meta-model of classifications

rent_substates attribute of the *cmm_state* entity. Like the state-transition diagrams (see Subsection 4.3.4), this construct allows concurrent composite states to be modeled, i.e. states that consist of several concurrent substates.

5.5.2 Behavioral Extension to the Relationship Concepts

In order to support the behavioral aspect, the common meta-model provides the concept of a transition, i.e. the passing of control from one classification to another, triggered by an event. In the common meta-model, transitions are modeled as relationships, the respective part of the common meta-model is shown in Figure 5.11.

Transitions are represented by the abstract *cmm_control_transition* entity. The triggering event of a *cmm_control_transition* is referenced through the *trigger* attribute. A condition that must be fulfilled to before firing the trigger can be stored in the *condition* attribute in the form of a boolean expression.

Figure 5.12 shows the support of the common meta-model for events that may trigger transitions. Events are generally represented by the *cmm_event* entity. Function calls are a subtype of events, as modeled in the UML meta-model in [Gro99], represented through the *cmm_function_call* entity, a subclass of *cmm_event*. The called function is referenced through the *function* attribute. The calling classification is referenced through the *caller* attribute. The callee is implicitly referenced through the called function's *owner* attribute, see Subsection 5.4.1. The attribute *order_number* allows an uninterpreted sequence number for the function call to be stored, optionally used in the object-oriented collaboration and sequence diagrams (see Subsections 3.2.5 and 3.2.6, respectively). Temporal relationships among function calls are represented by the *cmm_temporal_function_call_relationship* entity, whereby the earlier function call is referenced through the *predecessor* attribute and the following function call is referenced through the *successor* attribute.

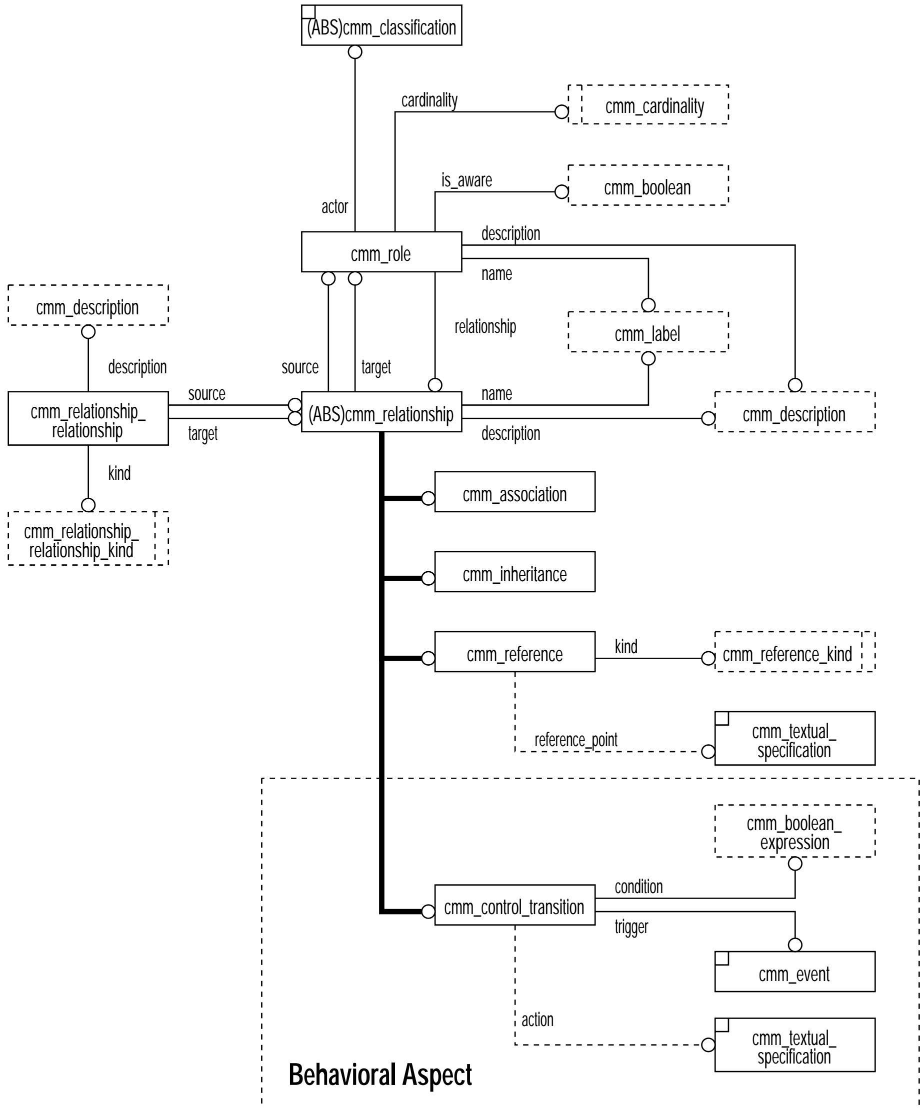

Fig. 5.11: Common meta-model of relationships

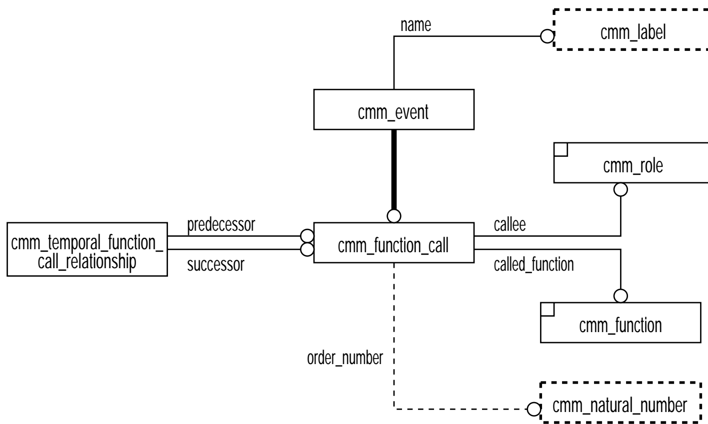

Fig. 5.12: Common meta-model of events

6. INTEGRATING THE SPECIFICATION TECHNIQUES THROUGH THE COMMON META-MODEL

This chapter describes the mappings from the meta-models of the structured and object-oriented specification techniques (described in Chapter 4) to the common meta-model (presented in the Chapter 5), and vice-versa. The mappings define how data can actually be exchanged through the common meta-model between two tools that make use of different specification techniques.

6.1 Integration Principle

A specification of a system, or a part of a system, is usually modeled using a tool that implements a particular specification technique. Hence, a specification is an instantiation of the meta-model of the underlying specification technique. The principle of integrating the different specification techniques presented in Chapter 3 is illustrated in Figure 6.1. The meta-models of the specification techniques (presented in Chapter 4) and the common meta-model (presented in Chapter 5) have been described using the EXPRESS and EXPRESS-G languages (see Section 4.1.2). The mappings between the meta-models, presented in this chapter, are described using EXPRESS-X (see Subsection 4.1.3 and Sections 6.2 and 6.3).

In order to exchange specifications between different specification tools, the specification data is first transformed into its representation in the common meta-model, and then transformed into the representation of the meta-model of the specification technique underlying the sink tool. Due to using

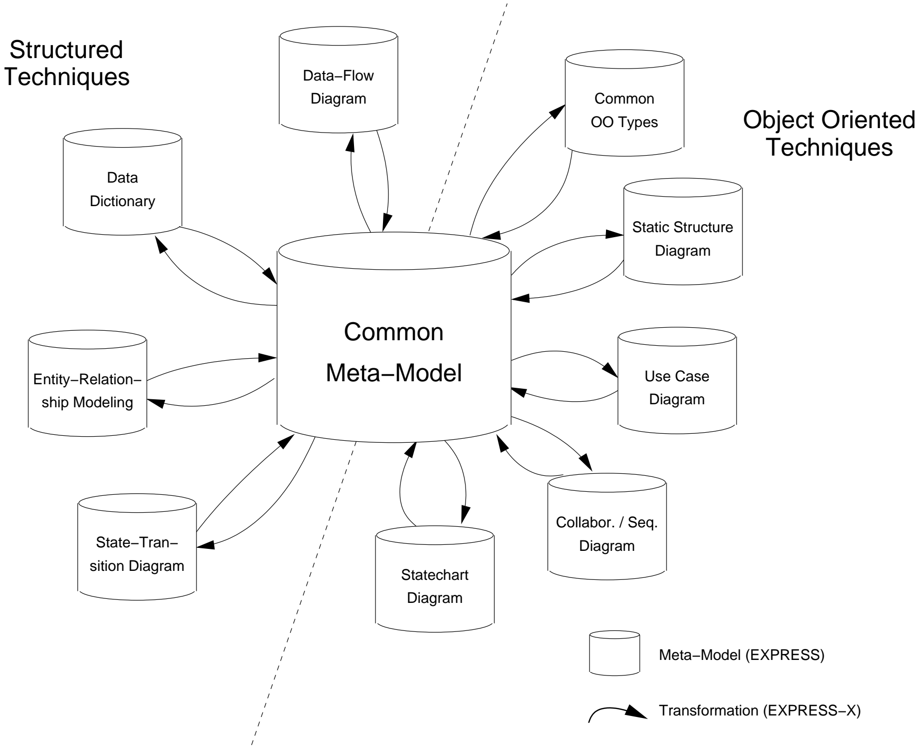

Fig. 6.1: Integration Principle

the common meta-model as central intermediate point of data exchanges, the transformation from one tool representation to another tool representation always requires two sub-transformations: one from the meta-model of the specification technique underlying the source tool to the semantically richer common meta-model, and another from the semantically richer common meta-model to the meta-model of the specification technique underlying the sink tool. The transformations are not necessarily symmetric, i.e. the transformation of one source concept to a representation in the sink meta-model and then back to a representation in the source meta-model does not necessarily result in the same representation using the same concepts. For example, as demonstrated later, the transformation of an inheritance relationship to a representation in a data dictionary results in a decomposition of the inheritance into a sequence of its compartments, which in turn is transformed to a flat composite data structure in the reverse transformation.

6.2 EXPRESS-X (ISO 10303-14)

The mappings in the remainder of this chapter are described using the EXPRESS-X mapping notation ISO 10303-14. The example presented in Figure 6.2 shows a simplified excerpt of an EXPRESS-X mapping from an EXPRESS meta-model of entity-relationship diagrams to an EXPRESS meta-model of class diagrams. Note that this is a stand-alone example and is not related to the meta-model mappings presented in the remainder of this chapter. The example merely intends to present the principles of EXPRESS-X mappings.

It is assumed that there exists an EXPRESS meta-model that defines the entities *class*, *entity*, and their attributes. The mapping from an entity-relationship entity to a class (*entity_to_class*, lines 3 to 17) generates a new *class* object (line 5) plus a set of *attribute* objects (line 6) from each *entity* object (line 8) found in the source specification. The name of the *class* object is copied from the *entity* object (line 12). The attributes of the *class* object are generally set to a (still empty) set of class attributes (line 13). The members of the set are then generated from the attributes of the source *entity* (lines 14 to 16). This is done through calling the *entity_attribute_to_attribute* mapping (line 16 and 19 to 27) for each attribute of the entity. When calling a mapping, the source elements are provided as parameters of the call. In the example, the *entity_attribute_to_attribute* mapping has one *entity_attribute* as source element (lines 22 and 23), thus, the mapping call (line 16) needs to

```
1  ...
2
3  MAP entity_to_class
4  AS
5    c: class;
6    attr: AGGREGATE OF attribute;
7  FROM
8    e: entity;
9  IDENTIFIED_BY
10   e.name;
11 SELECT
12   c.name := e.name;
13   c.attributes := attr;
14   FOR EACH a in e.attributes INDEXING i;
15     SELECT
16       attr[i] := entity_attribute_to_attribute(a);
17 END_MAP;
18
19 MAP entity_attribute_to_attribute
20 AS
21   a: attribute;
22 FROM
23   e: entity_attribute;
24 SELECT
25   a.name := e.name;
26   a.type := ?;
27 END_MAP;
28
29 ...
```

Fig. 6.2: Example of an EXPRESS-X mapping

provide a suitable parameter. Note that the class attribute is "richer" than the entity attribute; it provides an additional *type* attribute (line 26) that describes the type of the class attribute. However, this is not supported by an entity-relationship attribute. Hence, the type of the class attribute is unknown, which is expressed by assigning the *type* attribute an unspecified value (represented in EXPRESS-X by a question mark).

6.3 EXPRESS-X Visualization

In the following sections the EXPRESS-X notation presented above is used to formally describe the mappings between the different meta-models. For each section, a graphical summary of the respective mappings is presented in order to provide an overview of the mappings before presenting the details. The STEP framework does not provide a standardized graphical notation for visualizing EXPRESS-X mappings, analogous to EXPRESS-G as graphical notation for EXPRESS code. Therefore, the graphical notation presented in Figure 6.3 is used in the subsequent sections for graphically sketching the mappings between elements of the different meta-models. Note that this notation is only intended for presenting overviews of the mappings. It does not support all features of EXPRESS-X and is only used to illustrate the most important mappings, leaving out less important details as described later in Section 6.4.

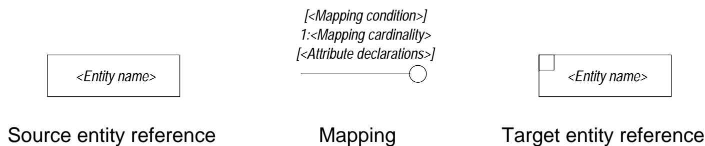

Fig. 6.3: EXPRESS-X visualization elements

The syntax of the visualization is based on elements from EXPRESS-G (see Subsection 4.1.2), specifically the entity and the relationship. In the EXPRESS-X visualizations, the elements in Figure 6.3 have the following semantics.

Source entity reference. This element references an entity of the source meta-model. It is depicted by a rectangle with the name of the referenced entity inside the rectangle.

Mapping. This element visualizes a mapping relationship of an entity of

the source meta-model to an entity of the target meta-model. The mapping is depicted by the EXPRESS-G relationship symbol, i.e. a line between the source and the target entity, whereby the line has a hollow circle attached towards the target entity's end. The mapping relationship is labeled with the optional mapping condition, the mapping cardinality, and optional attribute/value assignments for the target element. These label compartments are explained in more detail as follows.

Mapping Condition. The mapping condition is optional. It is decisive whether a transformation of a source element is actually performed. The condition expression starts with the keyword `WHERE` followed by a boolean expression, e.g. `WHERE kind = external`. Iff the expression evaluates to *true*, then the transformation is performed.

Mapping Cardinality. The mapping cardinality specifies how many target elements are created from one source element. The cardinality takes the form of `1:<target cardinality>`, e.g. `1:2` for a mapping where one source instance is transformed to two target instances.

Attribute-Value Assignment. The mapping relationship can optionally be made more explicit through additional attribute/value assignments in the form `<expression>=:<attribute>`. The `<attribute>` represents an attribute of the target element, and the `<expression>` describes how the value of the attribute is determined, using attributes of the source element and constant expressions. Note that the attribute is on the right side of the attribute/value assignment, so that it can be more easily associated with the (also right-sided) target element in the visualization. If the mapping results in several target elements, then the attribute-value assignments may be preceded by `#<target element nr>:` in order to distinguish different attribute/value assignments for different target elements. If the target attribute is a set of single values, then the operator `+=:` can be used to indicate that a value is added to the set of values already held by the attribute. If an attribute/value assignment is to be applied analogously to a set or an array of attributes, then the reference to a set is illustrated by using squared brackets in the form `<set name> [ ]`.
The expression `attributes [ ] =: properties []` for exam-

ple illustrates that the elements of the array *attributes* of the source element are mapped to the array *properties* of the target element.

In all three label compartments, the number of elements in a set is represented by including the set's name between two pipe signs in the form |`<set name>`|, e.g. |`elements`|. For example, this can be used to describe the mapping cardinality of a 1-to-n mapping, where n depends on the number of elements in a list, e.g. `1:|elements|`.

Target entity reference. This element of the EXPRESS-X visualization references an entity of the target meta-model. It is depicted by a rectangle with a small square in its upper left corner. Inside the rectangle, it is labeled with the name of the referenced entity of the target meta-model.

In order to better distinguish EXPRESS-G models and EXPRESS-X visualizations in this thesis, the EXPRESS-X visualizations use italics for labeling the visualization elements.

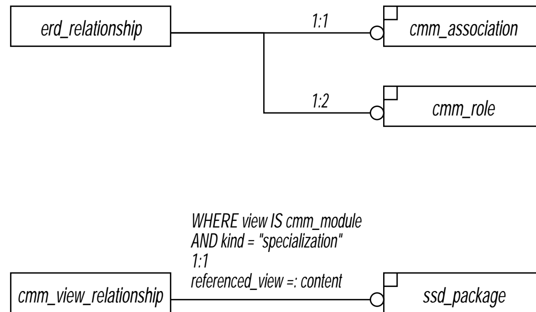

Fig. 6.4: EXPRESS-X visualization examples

Figure 6.4 shows the visualization of two example mappings. In the upper part of Figure 6.4, the source element *erd_relationship* is mapped to one target element *cmm_association* and two target elements *cmm_role*. This illustrates that one relationship instance of the source meta-model for entity-relationship modeling is mapped to one association instance in the common meta-model with two roles, the source and the target of the association (see the common meta-model of relationships in Subsection 5.5.2).

In the lower part of Figure 6.4, the source element *cmm_view_relationship* is mapped to one target element *ssd_package*, iff the condition `WHERE view IS cmm_module AND kind = specialization` applies. Furthermore, in the case of a satisfied condition, the attribute *content* of the target element *ssd_package* receives the equivalent value to the *referenced_view* attribute of the source element *cmm_view_relationship*. This illustrates that iff the superior view of a specialization relationship between two views in the common meta-model is a module, then the relationship is mapped to a package in a static structure diagram, whereby the content of the package is represented by the respective elements of the inferior view, referenced through *referenced_view* of *cmm_view_relationship*.

6.4 Common Mappings

The mappings between the meta-models of the single specification techniques (presented in Chapter 4) and the common meta-model (presented in Chapter 5) described in the following sections, are focusing on the major concepts of the respective specification technique as well as on their major attributes. The mappings of minor elements, e.g. the name of an element, are not explicitely described in the remainder of this chapter.

The common meta-model is the integrated aggregate of the underlying structured and object-oriented meta-models. It is intended to be implemented as central specification repository between different specification tools. It is assumed that the consistency within the repository is taken care of by the embedding database management system. If this is the case, then it is not important to be able to map every single attribute of every concept of the common meta-model to a representation in one of the specification techniques. It is more important during an import to put back previously exported and modified elements into their original position in the specification in the repository. Then the non-mappable elements that remained in the repository are automatically re-assigned to the modified elements. For example, the common meta-model provides a data-type for storing uninterpreted textual descriptions of a model element (*cmm_description*). This concept is not provided by any of the meta-models of the specification techniques in Chapter 4, and hence, it cannot be mapped. However, the mapping of simple textual element descriptions is not of prime importance as they are not meant to carry further specification information. If, after an export, the reverse transformation to the common meta-model is performed, then the exported elements will be put back in their original place in the general specification in the repository and, hence, are also re-assigned to their textual descriptions.

Elements that are usually not explicitely mapped, due to having analogous counterparts in the respective sink meta-model or to being of minor importance, are described in the following paragraphs. However, if the mapping of an element of these kinds is not straightforward or of special interest, then the mapping for this elements will also be explicitely provided.

Labels. Labels or names of elements, represented by the *cmm_label* datatype.

Descriptions. Uninterpreted textual descriptions, e.g. descriptions of single

diagram elements, represented by the *cmm_textual_description* data-type.

Boolean expressions. Boolean expressions, e.g. conditions for transitions in state-transition diagrams, represented by the *cmm_boolean_expression* data-type.

Textual specifications. Textual specifications of system functions, e.g. in the form of source code, represented by the *cmm_textual_expression* entity.

6.5 Mappings from Structured Techniques

6.5.1 From Data-Flow Diagrams

Figure 6.5 shows an overview of the mapping of elements of the data-flow diagram meta-model (presented in Subsection 4.3.1) to elements of the common meta-model (presented in Chapter 5). The complete formal EXPRESS-X mapping can be found in Section B.1 of Appendix B.

The mapping from the data-flow diagram meta-model to the common meta-model is carried out as follows.

Diagram. A data-flow diagram itself (*dfd_diagram*) is mapped to a module (*cmm_module*), because a diagram represents a logical grouping of specification elements, which is represented in the common meta-model through modules.

Process. A process (*dfd_process*) is mapped to a classification definition in the common meta-model (*cmm_classification_definition*), because a process basically represents a logical grouping of functions (a functionality). However, processes contribute only to the functional aspect of classifications. Hence, in order to distinguish the character of such classification definitions from others, their *kind* attribute is set to *functionality*.

Data-flow. A data-flow represents a channel through which data of a specific type can flow and resembles the association concept of the common meta-model. Thus, a data-flow (*dfd_data_flow*) is mapped to the association relationship (*cmm_association*) of the common meta-model as well as to two roles, represented by *cmm_role* entities that connect the respective elements to the association. However, the association needs to be directed, as the data-flow is also directed. This implies that the target of the association is not "aware" of the association, i.e. it does not know from which source the data comes through the association. This is modeled by setting the *is_aware* attribute of the respective *cmm_role* entity to *false*. The data identifiers associated with the data-flow (represented by the *data_identifiers* attribute of *dfd_data_flow*) are mapped to properties of a classification definition (*cmm_classification_definition*). The classification definition is then attached to the association through an association classification (*cmm_association_classification*).

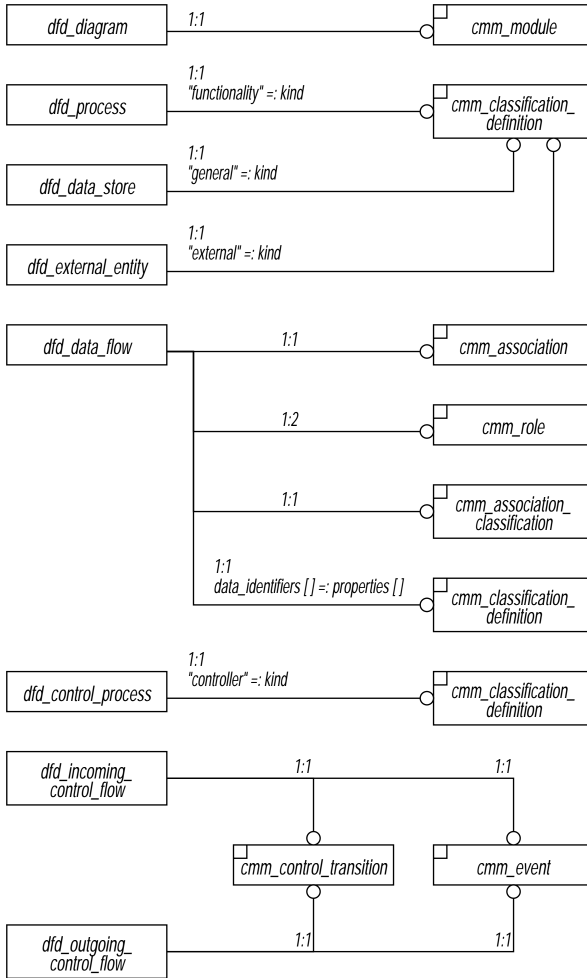

Fig. 6.5: Mapping overview: Data-flow diagram meta-model to common meta-model

Data store. A data-store is a structural description of a repository, just as the data aspect of the classification concept of the common meta-model. Hence, the data-store (*dfd_data_store*) is mapped to the classification concept (*cmm_classification_definition*) of the common meta-model.

External entity. An external entity (*dfd_external_entity*) is a placeholder for an external source or sink of data and is mapped to a classification definition in the common meta-model. External entities are usually not specified in detail, i.e. they usually do not exhibit further structural nor functional components. In order to illustrate this, the *kind* attribute of the respective *cmm_classification_definition* entity is set to *external*.

Control process. A control process (*dfd_control_process*) is, like a normal process, mapped to the classification definition concept (represented by the *cmm_classification_definition* entity) of the common meta-model. However, to distinguish its control character, the *kind* attribute of the respective *cmm_classification_definition* entity is set to *controller* (see explanation in Subsection 5.3.1).

Incoming control flow. Incoming control flows into a control process (represented by the *dfd_incoming_control_flow* entity) are mapped to control transitions (*cmm_control_transition*) in the common meta-model, having a triggering event (*cmm_event*) associated with the control transition.

Outgoing control flow. Like incoming control flows, outgoing control flows (*dfd_outgoing_control_flow*) are mapped to control transitions with an associated event (*cmm_event*).

6.5.2 From Data Dictionaries

Figure 6.6 shows an overview of the mapping of the meta-model for data dictionaries (see Subsection 4.3.2) to the common meta-model.

In more detail, the mapping of the elements of data dictionaries to respective elements of the common meta-model is carried out as follows.

Specification. A data dictionary itself, i.e. the reference to a complete data dictionary specification (*dd_specification*), is mapped to a module (*cmm_module*) in the common meta-model, which, like the data

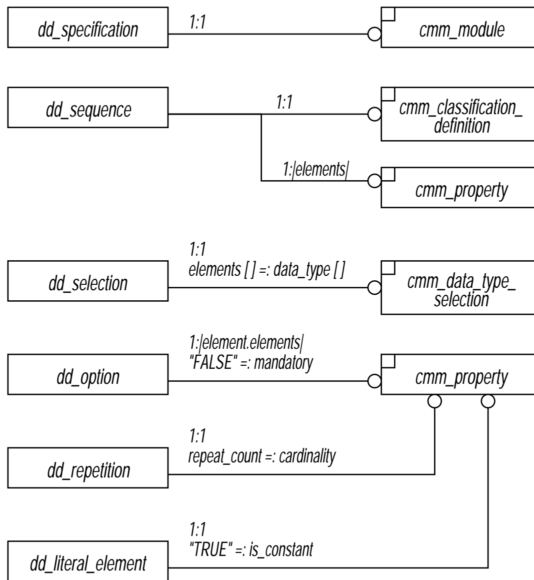

Fig. 6.6: Mapping overview: Data dictionary meta-model to common meta-model

dictionary, represents a logical grouping of elements.

Sequence. A sequence in a data dictionary (*dd_sequence*), i.e. the association of a keyword with a concatenated list of elements, is represented by a classification (*cmm_classification_definition*) that contains the elements of the sequence as properties, represented by *cmm_property* entities.

Selection. The possibility of a selection of one element from a set of elements (*dd_selection*) is mapped to the analogous data-type selection construct (*cmm_data_type_selection*) of the common meta-model.

Option. Optional sequences (*dd_option*) are mapped to a set of properties (*cmm_property*) that is owned by a classification definition (referenced through the *owner* attribute), whereby each of the properties is made optional by setting their *mandatory* attribute to *false*.

Repetition. Repetitions of data dictionary elements (represented by the *dd_repetition* entity) are mapped to a property (*cmm_property*), whose cardinality in the *cardinality* attribute is set to the number of repetitions, i.e. the *repeat_count* attribute of the respective *dd_repetition* entity.

Literal Element. Literal elements of a data dictionary (represented by the *dd_literal_element* entity) are mapped to constant properties, represented by the *cmm_property* entity, having the inherited *is_constant* attribute set to *true*.

6.5.3 From Entity-Relationship Diagrams

Figure 6.7 provides an overview of the mapping of the entity-relationship diagram meta-model presented in Subsection 4.3.3 to the common meta-model.

The detailed mapping of elements of the entity-relationship diagram meta-model to elements of the common meta-model is carried out as follows.

Diagram. An entity-relationship diagram is a logical grouping of entity-relationship diagram elements. Hence, the reference to an entity-relationship diagram (*erd_diagram*) is mapped to a module in the common meta-model, represented by the *cmm_module* entity.

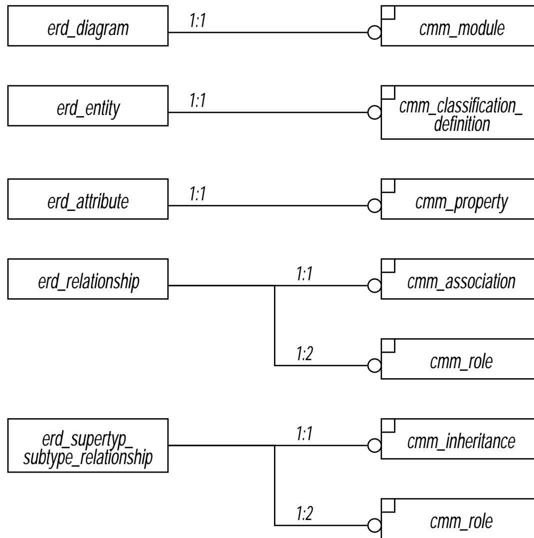

Fig. 6.7: Mapping overview: Entity-relationship diagram meta-model to common meta-model

Entity. An entity in an entity-relationship diagram (*erd_entity*) is mapped to a classification (represented by the *cmm_classification_definition* entity). This mapping serves only the data aspect of a classification definition, i.e. it provides only structural information without contributing to the functional aspect of the classification definition.

Attribute. An attribute of an entity (*erd_attribute*) is mapped to a property (*cmm_property*), which in turn is associated to the owning classification through its *owner* attribute.

Relationship. A relationship in an entity-relationship diagram (represented by the *erd_relationship* entity) is mapped to an association relationship in the common meta-model (*cmm_association*). The source and target elements of the relationship are attached to the association through roles (represented by *cmm_role* entities), which in turn are connected through their *actor* attribute with the respective classifications representing the related entities.

Supertype / Subtype Relationship. A supertype / subtype relationship in an entity-relationship diagram (*erd_supertype_subtype_relationship*) is mapped to an inheritance relationship in the common meta-model

(*cmm_inheritance*). The related entities are connected to the inheritance relationship through roles (*cmm_role* entities), which in turn reference the respective classifications representing the related entities through their *actor* attribute.

6.5.4 From State-Transition Diagrams

Figure 6.8 gives an overview of the mapping of the elements of the state-transition diagram meta-model presented in Subsection 4.3.4 to elements of the common meta-model (presented in Chapter 5). The complete formal EXPRESS-X mapping specification of this can be found in Section B.3 of Appendix B.

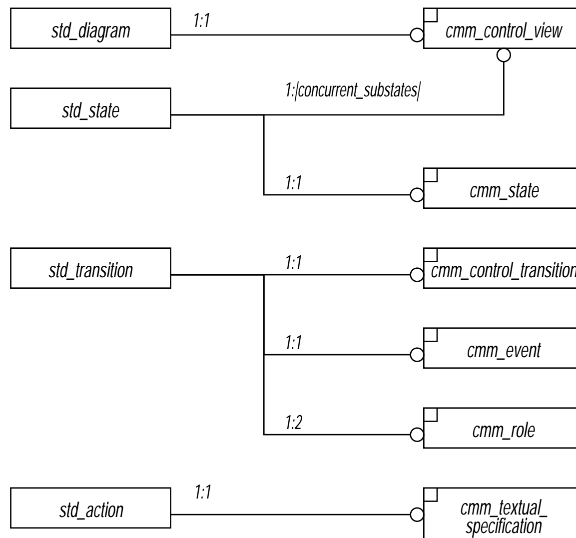

Fig. 6.8: Mapping overview: State-transition diagram meta-model to common meta-model

The detailed mapping of the elements of the state-transition diagram meta-model to the common meta-model is carried out as follows.

Diagram. A state-transition diagram (represented by the *std_diagram* entity) is a container of diagram elements. It is mapped to the analogous control view concept of the common meta-model (*cmm_control_view*),

which also represents a logical grouping of model elements that describe the behavior of a (part of a) system.

State. A state (*std_state*) is mapped to the analogous state concept of the common meta-model (*cmm_state*). If the state has concurrent sub-states (represented by the *concurrent_substates* attribute of the *std_state* entity), an additional control view (*cmm_control_view*) is created for each of the concurrent sub-states.

Transition. A transition (*std_transition*) is mapped to the control transition relationship in the common meta-model (*cmm_control_transition*) and two roles (*cmm_role*) referencing the source and sink states (through their *actor* attributes). The event that triggers the transition (represented by the *event_expression* attribute of *std_transition* entity) is mapped to the event concept (*cmm_event*) of the common meta-model.

Action. An associated action of a transition (*std_action*) is stored as a specification using a *cmm_textual_specification* entity, which is attached to the respective control transition through its *action* attribute.

6.5.5 Summary

In summary, the support of the common meta-model for the elements of the single meta-models of the structured specification techniques is obvious, as all elements of the single meta-models can be mapped to elements of the common meta-model. This is due to the fact that the common meta-model has been derived from the meta-models of the single specification techniques.

Processes, external entities, and data stores from data-flow diagrams are mapped to the classification concept of the common meta-model, whereby the difference in their semantics is kept by specifying the kind of classification as attribute value. Furthermore, data-flows are mapped to associations, and control flows are mapped to control transitions.

A sequence in a data dictionary, i.e. a definition of a keyword, is mapped to a classification. Optional components, repetitions and literal elements in such a sequence are mapped to properties, which are attached to a classification. The selection in a sequence in data dictionary is represented in the common meta-model through the analogous concept of a data-type selection, allowing the specification of a set of data-types of which one is selected in case of instantiation.

The entities of entity-relationship modeling are mapped to classifications, attributes of entities are mapped to properties (which in turn are owned by classification definitions that represent the owning entities). The relationships of entity-relationship modeling are mapped to associations, whereas the supertype / subtype relationship is mapped to the analogous inheritance concept.

For state-transition diagrams, the states are mapped to the state concept of the common meta-model. Transitions are mapped to the analogous control transitions and actions associated with transitions are mapped to textual specifications that are attached to the respective control transition.

Furthermore, the different diagrams themselves can be seen as containers for elements of a specific specification technique. Hence, all diagrams are themselves represented by one of the view concepts of the common meta-model. Most diagrams are represented by modules, except the state-transition diagram, which is represented by the control view.

6.6 Mappings to Structured Techniques

6.6.1 To Data-Flow Diagrams

Figure 6.9 gives an overview of the mapping of elements from the common meta-model to the meta-model of data-flow diagrams, presented in Subsection 4.3.1.

The detailed mapping is performed as follows.

Module. A module (*cmm_module*) is a logical grouping of diagram elements and is mapped to the analogous reference to data-flow diagram (*dfd_diagram*).

Classification Definition. The mapping from classification definitions (represented by the *cmm_classification_definition* entity) to elements of the data-flow diagram meta-model depends on the following possible values of the *kind* attribute of the *cmm_classification_definition* entity.

Functionality. The classification is transformed to a process (represented by the *dfd_process* entity) in the data-flow diagram, because it represents a piece of functionality of the system under specification, which is represented by data-flow processes, according to the descriptions in Subsection 3.1.2.

External. The classification is considered to be external to the system under specification. It is transformed to an external entity (*dfd_external_entity*) in the data-flow diagram, which represents the analogous counterpart in a data-flow diagram.

Controller. The classification only controls the execution of the part of the system under consideration, it does not represent a structural part of the system itself. Hence, the classification definition is mapped to a control process (*dfd_control_process*) in the data-flow diagram.

Global. The respective classification definition is a conventional element of the system specification, it does not specifically represent a system function or behavior. In this case, the classification definition is transformed into a data store representation (represented by the *dfd_data_store* entity).

Association and Association Classification. If an association (represented

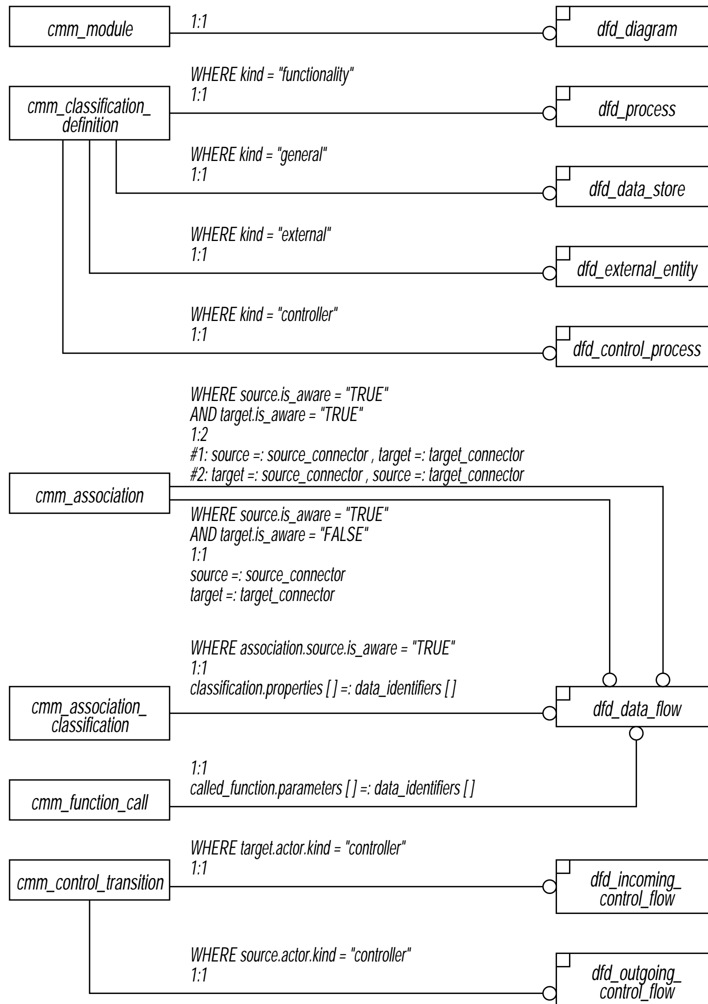

Fig. 6.9: Mapping overview: Common meta-model to data-flow diagram meta-model

by the *cmm_association* entity) is unidirectional, i.e. only the source element is aware of its participation in the association, then the association is mapped to a data-flow (*dfd_data_flow*) in the data-flow diagram. Bidirectional associations are mapped to two data-flows, one going from the source element to the target element, and one going from the target element to the source element. Unidirectional associations from the target to the source are not considered as associations are not supposed to be modeled in this way.
Association classifications are mapped to data-flows in the same fashion, however, the properties of the association classification (inverse *properties* attribute of *cmm_classification_definition*) are mapped to the data identifiers of the respective data-flow (*data_identifiers* attribute of *dfd_data_flow*).

Function Call. A function call implies a flow of data between the calling and the called process. Hence, a function call (*cmm_function_call*) is mapped to a data-flow (*dfd_data_flow*), whereby the parameters of the called function are mapped to data identifiers of the respective data-flow (*data_identifiers* attribute of *dfd_data_flow*).

Control Transition. A control transition (*cmm_control_transition*) is mapped to an incoming control flow (*dfd_incoming_control_flow*) if the target of the control transition is a classification that solely controls the execution of the system under specification, i.e. its *kind* attribute has the value *controller*. A control transition is mapped to an outgoing control flow (*dfd_outgoing_control_flow*) if the source of the control transition is such an execution controlling classification.

6.6.2 To Data Dictionaries

Figure 6.10 presents an overview of how elements from the common meta-model are mapped to elements of the meta-model for data dictionaries as presented in Subsection 4.3.2.

In detail, the mapping is carried out as follows.

Module. A module (*cmm_module*) as a logical grouping of specification elements is mapped to a reference to a data dictionary specification (*dd_specification*) as this also represents a container of modeling elements.

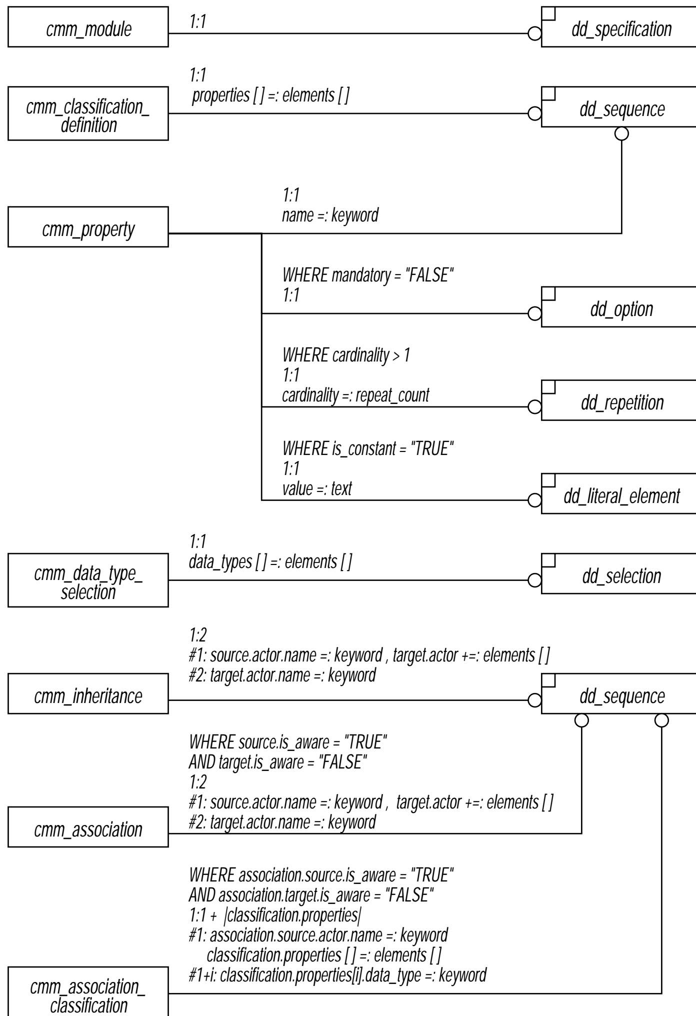

Fig. 6.10: Mapping overview: Common meta-model to data dictionary meta-model

Classification Definition. A classification definition (represented by the entity *cmm_classification_definition*) is mapped to a sequence (*dd_sequence*) in a data dictionary, whereby the keyword for the sequence is the same as the name of the classification definition. The elements of the sequence (*elements* attribute of *dd_sequence*) are derived from the properties of the classification definition (*properties* attribute of the entity *cmm_classification_definition*, referencing *cmm_property* entities).

Property. Properties (*cmm_property*) are mapped to a sequence (*dd_sequence*) in a data dictionary, whereby the keyword of the sequence (its *keyword* attribute) is set to the name of the data-type of the property. The data dictionary only illustrates the structure of a sequence without giving names to the single elements of a structure, i.e. the actual name of the property cannot be transformed into a data dictionary representation. In addition to the described mapping, a property may be mapped according to the following cases.

Option. If the property is optional, i.e. its *mandatory* attribute has the value *false*, then an option (*dd_option*) is generated in the data dictionary, which is associated with the sequence representing the property (through the *element* attribute of *dd_option*).

Repetition. If the property has a cardinality greater than 1, i.e. its *cardinality* attribute has a value greater than *1*, then a repetition (*dd_repetition*) is generated, which is associated with the "repeated" element (which in turn is a sequence, as described above) through its *element* attribute. The number of repetitions are stored in the *repeat_count* attribute of *dd_repetition.*

Constant. If the property is a constant, i.e. its *is_constant* attribute has the value *true*, then a literal element (*dd_literal_element*) is generated in the data dictionary and associated with the sequence generated from the property (as explained above) through the *elements* attribute. The *text* attribute of the literal element obtains its value from the *assigned_value* attribute of the respective property.

Data-Type Selection. The representation of a selection of data-types in the common meta-model (*cmm_data_type_selection*) is mapped to the analogous selection concept (*dd_selection*) of the data dictionary. The elements of the selection (*elements* attribute) are constituted from the data-types in the originating selection (*data_types* attribute of

cmm_data_type_selection).

Inheritance. An inheritance relationship (*cmm_inheritance*) is mapped to two sequences (*dd_sequence*). The first sequence represents the more specific element, the second sequence represents the more general element. The second (more specific) sequence is then composed from the more general sequence by assigning the *elements* attribute of the first sequence (*dd_sequence*) to the second sequence. Both sequences obtain the values of their *keyword* attribute from the *name* attribute of the respective originating element.

Association. A unidirectional association is mapped in the same way as the inheritance relationship described above, i.e. to two sequences, the first including the second (a unidirectional association is an association whose source classification is aware of participating in the association but whose target classification is not aware of it). Note that unidirectional associations going from the target to the source element are not mapped, as this is assumed not to appear in a specification. Furthermore, bidirectional associations are not mapped because they would result in two circular sequence definitions, which is not supported by data dictionaries.

Association Classification. An association classification (represented by the entity *cmm_association_classification*) is only mapped to the data dictionary if the respective association (referenced through the *association* attribute) is unidirectional from the source to the target element, as explained for associations above. Each association classification is mapped to one sequence representing the source element of the respective association and a set of elements of the sequence (*elements* attribute) that are generated from the properties of the associated classification (*properties* attribute of the classification definition referenced by the *classification* attribute). Furthermore, each property is also mapped to a sequence in order to be referenced in the *elements* attribute of the association sequence.

6.6.3 To Entity-Relationship Diagrams

Figure 6.11 gives an overview of the mapping from elements of the meta-model of entity-relationship diagrams, as presented in Subsection 4.3.3, to the common meta-model.

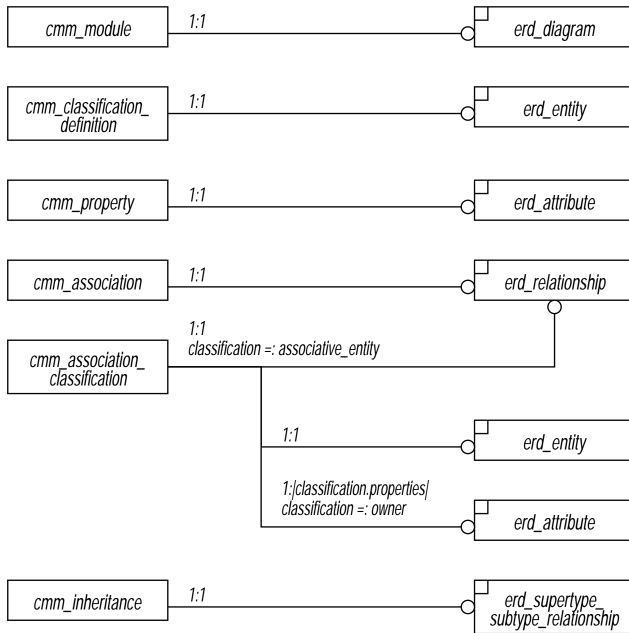

Fig. 6.11: Mapping overview: Common meta-model to entity-relationship diagram meta-model

The elements are mapped in detail as follows.

Module. A module (*cmm_module*) is mapped to the reference to a entity-relationship diagram (*erd_diagram*) as both are logical groupings of specification elements.

Classification Definition. An entity represents a structural aspect of the system under consideration, which is also represented by a classification definition (besides also representing a functional aspect). Hence, a classification definition (*cmm_classification_definition*) is mapped to an entity in an entity-relationship diagram (*erd_entity*).

Property. Properties (*cmm_property*) are mapped to attributes that are attached to an entity (*erd_attribute*) in the same manner as properties are attached to their owning classification definition. Note that attributes in an entity-relationship diagram are rudimentary strings, i.e. the greater part of the information associated with a property, e.g. datatype, cardinality, etc., can not be represented in a entity-relationship diagram.

Association. An association (*cmm_association*) is mapped to the relationship concept (*erd_relationship*) of the entity-relationship diagram meta-model, as both represent structural relationships among elements.

Association Classification. An association classification (represented by the entity *cmm_association_classification*) is mapped to the analogous associative entity concept of the entity-relationship meta-model. Furthermore, the properties of the classification definition (referenced through the *classification* attribute of the *cmm_association_classification* entity) are mapped to attributes (represented by the *erd_attribute* entity) of the associated entity.

Inheritance. An inheritance relationship (*cmm_inheritance*) is mapped to the analogous supertype / subtype relationship (represented by the *erd_supertype_subtype_relationship* entity) of the entity-relationship diagram meta-model.

6.6.4 To State-Transition Diagrams

Figure 6.12 shows an overview of how elements of the meta-model of state-transition diagrams (presented in Subsection 4.3.4) are mapped to elements of the common meta-model.

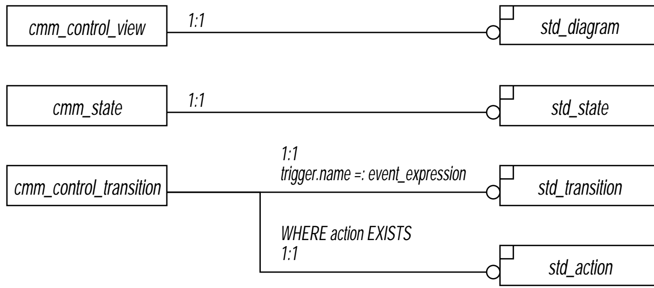

Fig. 6.12: Mapping overview: Common meta-model to state-transition diagram meta-model

The mapping is carried out in detail as follows.

Control View. A control view (*cmm_control_view*) represents a dynamic aspect of the system under consideration. This is mapped to a reference to a state-transition diagram (*std_diagram*). A module (*cmm_module*) represents a structural view of the system and can not be mapped to a representation as state-transition diagram.

State. A state under the common meta-model (*cmm_state*) is mapped to the analogous state concept of the state-transition diagram meta-model, represented by the *std_state* entity.

Control Transition. A control transition (*cmm_control_transition*) is mapped to the analogous transition concept of the state-transition diagram meta-model (*std_transition*). If an action is associated with the control transition, i.e. the *action* attribute of the *cmm_control_transition* entity is specified, then an action (*std_action*) is created according to the state-transition diagram meta-model and associated with the transition through the *action* attribute of the respective *std_transition* entity.

6.6.5 Summary

For the mapping to data-flow diagrams, the most important concepts of the common meta-model are the classification and the association. Depending on the kind of classification, respective counterparts of the data-flow diagram are created, e.g. a process for a classification that represents a piece of functionality of the system under specification. However, the structural internals of classifications can not be transformed into elements of a data-flow diagram and hence, not be modified using a data-flow diagram tool either. Associations are mapped to data-flows, as an association actually defines a channel between two elements through which actual data can flow, just as the data-flow. Bidirectional associations are transformed into two counter-directed data-flows. A function call implies the flow of data between two elements, hence, it is also mapped to a data-flow. Control transitions are mapped to the analogous control flows.

As data dictionaries consider only the structural aspect of a system, only the elements of the common meta-model that support the structural aspect can be mapped to elements of the data dictionary meta-model. Basically every transformed element is represented as a sequence, identified by a keyword, and defined by a set of elements that constitute the sequence. In this manner, classifications are mapped to sequences, whereby the structural properties of the classification each yield another component of the respective sequence representing the classification. Properties may additionally be transformed into an option, repetition or a literal element, depending on the values of their attributes. In order to represent inheritance relationships in data dictionaries, they are "flattened" into inter-linked sequences. This allows for representing the "inheriting" character, which actually describes that one element that also consists of the features of another element.

The mapping to elements of the entity-relationship diagram meta-model also considers only the data aspect because entity-relationship modeling does not support the functional and behavioral aspects. The major elements of the mapping are the classification (which is mapped to an entity) and the association (which is mapped to a relationship). The details of the mapped elements, e.g. the data-type of a property, can for the greater part not be mapped as entity-relationship modeling does not provide sufficiently detailed support for them.

The mapping from elements of the common meta-model to elements of the meta-model for state-transition diagrams is limited to elements that rep-

resent the behavioral aspect, but is relatively straightforward. The major elements are the state, which is mapped to the state concept of the state-transition diagram, and the control transition, which is mapped to the transition concept.

The module concept of the common meta-model is in most cases (except for state-transition diagrams) mapped to a diagram of the respective specification technique. State-transition diagrams are generated from control views.

In summary, the transformations from representations under the common meta-model to representations under the meta-model of a structured specification technique can often not cover the common meta-model completely. This is mainly due to the fact that the structured specification techniques consider only one aspect of the system design and thus are not able to interpret and modify other aspects. Furthermore, the supported level of design granularity within one aspect differs between the specification techniques, e.g. entity-relationship modeling allows for naming attributes as constituents of an entity (like components of a data structure), whereas a sequence in a data dictionary solely considers the components of the data structure without supporting naming of the components.

6.7 Mappings from Object-Oriented Techniques

Unlike the structured specification techniques, the object-oriented specification techniques presented in Section 3.2 are all based on a single notation, namely the UML, and share a number of common concepts. Therefore, this section describes first the mappings of these common concepts, followed by the mappings of the specific concepts of each of the object-oriented specification techniques.

6.7.1 From Common Object-Oriented Elements

The major common concepts of the object-oriented meta-models presented in Section 4.4 are the classifier and the association. Figure 6.13 provides an overview of how a classifier and its compartments are mapped to elements of the common meta-model.

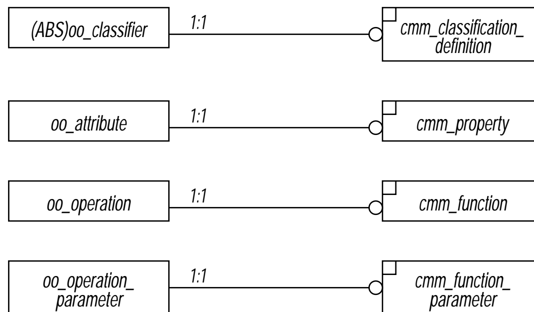

Fig. 6.13: Mapping overview: Object-oriented classifier to common meta-model

More detailed, the mapping is carried out as follows.

Classifier. Classifiers are mapped to the analogous classification definition concept of the common meta-model (*cmm_classification_definition*). However, Figure 6.13 shows only the principle of mapping the classifiers to the common meta-model, as abstract elements can actually not be mapped, because they have no instantiations that could be transformed. The actual mappings of the classifier to the classification definitions are implemented as mappings from the subclasses of the classifier, i.e. class (*ssd_class*), use case (*ucd_use_case*), or actor (*ucd_actor*).

Attribute. An attribute (*oo_attribute*) is mapped to the analogous concept of a property (*cmm_property*) of the common meta-model.

Operation. An operation (*oo_operation*) is mapped to the analogous concept of a function (*cmm_function*) of the common meta-model.

Operation Parameter. Parameters of an operation (represented by the entity *oo_operation_parameter*) are mapped to the analogous function parameter concept (*cmm_function_parameter*) of the common meta-model.

Figure 6.14 provides an overview of the mapping from common object-oriented relationships, namely association and generalization, to their counterparts in the common meta-model.

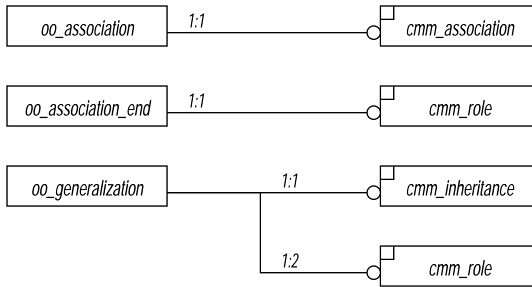

Fig. 6.14: Mapping overview: Object-oriented relationships to common meta-model

In more detail, the mapping is carried out as follows.

Association. An object-oriented association (*oo_association*) is mapped to the analogous concept of an association in the common meta-model (*cmm_association*).

Association End. The end points of object-oriented associations, represented by the *oo_association_end* entity, are mapped to the role concept (*cmm_role*) of the common meta-model.

Generalization. A generalization (*oo_generalization*) is mapped to the inheritance concept (*cmm_inheritance*) of the common meta-model, whereby also two roles (*cmm_role*) are created that reference the related classifications.

6.7.2 From Static Structure Diagrams

Figure 6.15 provides an overview of how elements from the meta-model of the object-oriented static structure diagram are mapped to elements of the common meta-model.

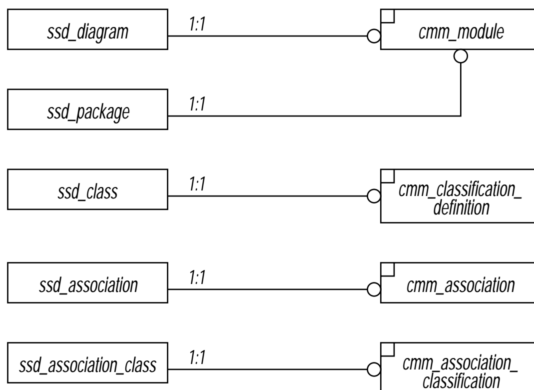

Fig. 6.15: Mapping overview: Static structure diagram meta-model to common meta-model

More specifically, the mapping is carried out as follows.

Diagram. The reference to a static structure diagram (*ssd_diagram*) is mapped to the module construct of the common meta-model (*cmm_module*), as both represent a logical grouping of diagram elements.

Package. Also, the object-oriented package (*ssd_package*) represents a logical grouping of specification elements. It is also mapped to the module concept (*cmm_module*) of the common meta-model. Furthermore, the contents of a package, usually in the form of a static structure diagram (i.e. another module) is referenced through a view relationship (*cmm_view_relationship*) between the former and the latter modules.

Class. An object-oriented class (*ssd_class*) is mapped to the classification definition (*cmm_classification_definition*) of the common meta-model.

Association. An association in a static structure diagram (*ssd_association*) is mapped to the analogous association relationship (*cmm_association*) in the common meta-model. Association ends are mapped to roles, as described in Subsection 6.7.1.

Association Class. An object-oriented association class (*ssd_association_class*) is mapped to the analogous association classification (represented by the *cmm_association_classification* entity) of the common meta-model.

6.7.3 From Use Case Diagrams

Figure 6.16 provides an overview of the mapping from elements of the meta-model of the object-oriented use case diagram presented in Subsection 4.4.5 to elements of the common meta-model.

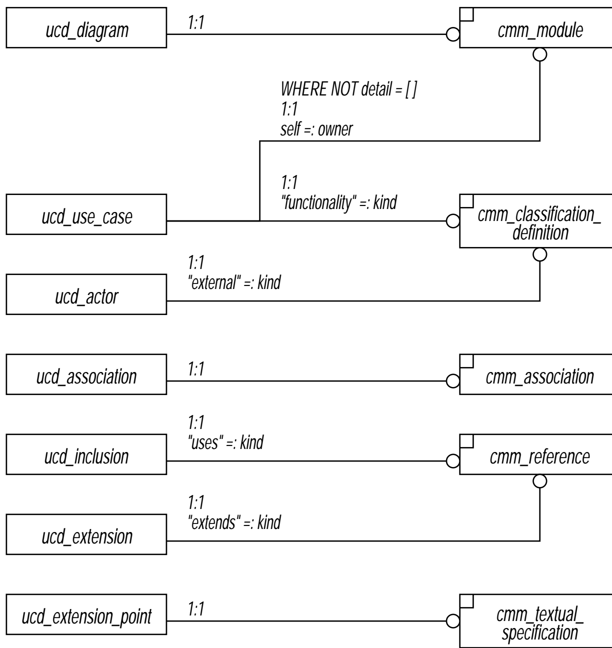

Fig. 6.16: Mapping overview: Use case diagram meta-model to common meta-model

The mapping is carried out as follows.

Diagram. The reference to a use case diagram (*ucd_diagram*) is mapped to the module concept (*cmm_module*) of the common meta-model, as both represent a logical grouping of diagram elements.

Use Case. A use case (*ucd_use_case*) is mapped to the classification concept

(*cmm_classification_definition*). The classification is marked to represent a functionality (through setting the *kind* attribute to *functionality*), because a use case illustrates a functionality of the system under specification.

Actor. An actor (represented by the *ucd_actor* entity) is mapped to a classification definition (represented by the *cmm_classification_definition* entity). However, the internals of an actor are considered to be outside the scope of the system specification, which is indicated through setting the *kind* attribute of the classification to *external*.

Association. Associations in a use case diagram (*ucd_association*) are mapped to the association relationship (*cmm_association*) of the common meta-model. Association ends are mapped to roles, as described in Subsection 6.7.1.

Inclusion. Inclusion references (*ucd_inclusion*) are mapped to reference relationships (*cmm_reference*) in the common meta-model, whereby the *kind* attribute of the reference obtains the value *uses* in order to describe the inclusive dependency between the related classifications.

Extension. Like inclusions, extensions of use cases (*ucd_extension*) are also mapped to reference relationships (*cmm_reference*) in the common meta-model. However, the *kind* attribute of the respective reference is set to *extends*, indicating the extending characteristic of the dependency between the related classifications.

Extension Point. Extension points (*ucd_extension_point*) are mapped to textual specifications (*cmm_textual_specification*) that specify the point of extension in more detail. The textual specification is associated with the respective extension reference through the optional *reference_point* attribute of *cmm_reference*.

6.7.4 From Collaboration and Sequence Diagrams

Figure 6.17 provides an overview of the mapping from elements of the collaboration and sequence diagram meta-model presented in Subsection 4.4.6 to elements of the common meta-model.

The mapping is carried out in detail as follows.

Diagrams. A reference to a collaboration diagram (*cd_diagram*) is mapped to

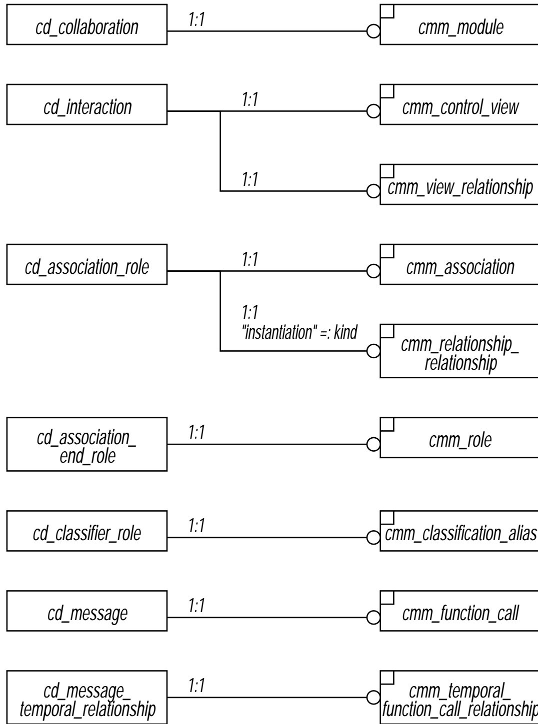

Fig. 6.17: Mapping overview: Collaboration and sequence diagram meta-model to common meta-model

a module of the common meta-model (*cmm_model*), as both represent structural groupings of diagram elements. The reference to an interaction (*cd_interaction*) is mapped to a control view (*cmm_control_view*), representing a behavioral aspect of the system specification, plus a relationship to the respective module (*cmm_view_relationship*) that represents the context of the interaction.

Association Role. The role of an association (*cd_association_role*) actually represents an instantiation of an association (*oo_association*). Thus, it is mapped to an association, whereas the instantiation character of the association playing a role in a context is reflected by creating a relationship between the association role and its definition in an association through a *cmm_relationship_relationship* of the kind *instantiation*.

Association End Role. The end points of an association role in a specific context (*cd_association_end_role*) are mapped to roles in the common meta-model (*cmm_role*). The roles are attached to the association which was created from the original association role.

Classifier Role. The classifier role represents the role that a classifier plays in a certain context. It does not define a classifier but is merely an alias for an already defined classifier. Hence, the classifier role (*cd_classifier_role*) is mapped to the analogous alias concept of the common meta-model (*cmm_classification_alias*). The respective classification is referenced through the *classification* attribute, and the name of the role is stored in the *name* attribute.

Message. A message (*cd_message*) in a collaboration or sequence diagram actually represents a function call. Thus, it is mapped to a function call in the common meta-model (*cmm_function_call*).

Message Temporal Relationship. Temporal sequences of messages, represented through the *cd_message_temporal_relationship* entity in the collaboration and sequence diagram meta-model, are mapped to the analogous concept of a temporal function call relationship (represented by the *cmm_temporal_function_call_relationship* entity).

6.7.5 From Statechart Diagrams

Figure 6.18 provides an overview of how elements from the meta-model of object-oriented statechart diagrams, presented in Subsection 4.4.7, are

mapped to elements of the common meta-model.

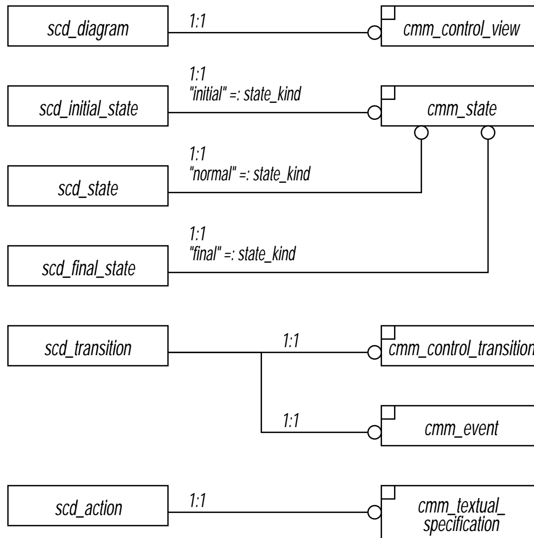

Fig. 6.18: Mapping overview: Statechart diagram meta-model to common meta-model

The mapping of the respective elements is carried out as follows.

Diagram. A statechart diagram (*scd_diagram*) as container of specification elements is a representation of a behavioral aspect of a system and hence, mapped to the control view concept of the common meta-model (*cmm_control_view*).

States. The different states in a statechart diagram are mapped to the state concept of the common meta-model, represented by the *cmm_state* entity. The different state kinds are distinguished using the *state_kind* attribute of *cmm_state*, whereby initial states (*scd_initial_state*) are represented through the value *initial*, final states (*scd_final_state*) are represented through the value *final*, and all other states (*scd_state*) are represented through the value *normal*.
Note that the statechart diagram meta-model presented herein intentionally provides no support for concurrent composite states in order to illustrate later (in Chapter 7) how specification representations can be transformed into representations under a structurally different meta-model.

Transition. A transition between states (*scd_transition*) of a statechart diagram is mapped to a control transition (*cmm_control_transition*) plus an event (*cmm_event*) in the common meta-model. The event is associated with the control transition through the *trigger* attribute of the *cmm_control_transition* attribute.

Action. An action associated with a transition (*scd_action*) is mapped to a simple textual specification (*cmm_textual_specification*), which in turn is associated with the owning control transition through the optional *action* attribute of *cmm_control_transition*).

6.7.6 Summary

The common meta-model has been built on the basis of the meta-models from both, structured and object-oriented specification techniques. Hence, like for the structured specification techniques, the support of the common meta-model for concepts from object-oriented specification techniques is obvious. However, the mapping of object-oriented concepts to concepts of the common meta-model is more straightforward because the common meta-model integrates structural, functional and behavioral aspects in the classification, which resembles the object-oriented principles of data and function integration in the class concept.

The most important common object-oriented elements, the classifier with its compartments and the relationships, have analogous counterparts in the common meta-model, namely the classification with its compartments and the relationships of the common meta-model.

Also, the elements of the static structure diagram have straightforward mappings to the common meta-model. Most prominent elements are the class, which is mapped to the classification definition concept, and the association, which is mapped to the analogous association concept of the common meta-model.

Use cases in use case diagrams are likewise mapped to the classification definition concept, as well as actors. The classification definition has an attribute that allows different semantics to be given to the classification definition. This is employed to distinguish use cases and actors from "normal" classes that merely specify the structure of the system under consideration. Associations between use cases and actors as well as among use cases themselves are mapped to associations respectively references in the common meta-model.

Elements from collaboration and sequence diagrams are mapped to the common meta-model according to the principle that collaborations and interactions (depicted by sequence diagrams) show the logical and behavioral relationships among roles that the involved classifiers play. The concept of a classifier role is mapped to the classification alias concept. Accordingly, an association role in a collaboration diagram is mapped to the association concept of the common meta-model. Additionally, a relationship between this association and its definition (by an other association) is established in order to keep the semantic difference between an association role and an

association.
Messages in sequence diagrams are mapped to function calls and the temporal ordering of messages is mapped to an analogous construct in the common meta-model.

The mapping of elements from statechart diagrams to elements of the common meta-model is quite straightforward. There is analogous support for the different kinds of states and the transition in the common meta-model.

All diagrams represent a logical grouping of specification elements, which is represented in the common meta-model through views. Hence, like for the structured techniques, the different diagrams are represented in the common meta-model as views. The different aspects of the system specification, e.g. the behavioral aspect of statechart diagrams, are maintained by mapping the diagrams to the respective subclass of the view, i.e. a module or a control view.

6.8 Mappings to Object-Oriented Techniques

The object-oriented specification techniques presented in Section 3.2 are all based on a single notation, namely the UML, and share a number of common concepts. Therefore, this section starts with describing the mappings of these common concepts first, followed by the mappings of the specific concepts of each of the object-oriented specification techniques.

6.8.1 To Common Object-Oriented Elements

Figures 6.19 and 6.20 provide an overview of the mapping from the common meta-model to the meta-models of common object-oriented elements.

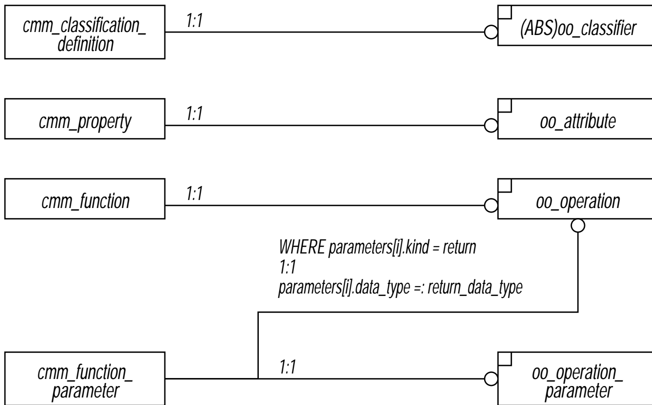

Fig. 6.19: Mapping overview: Common meta-model to object-oriented classifier meta-model

The concept of a classification definition and its compartments are mapped as follows.

Classification Definition. A classification definition, which is represented by the *cmm_classification_definition* entity, is mapped to the analogous object-oriented classifier concept, represented by the abstract *oo_classifier* entity. Note that this describes only the principal mapping, as the classifier is an abstract entity. Abstract entities cannot be mapped, as they have no instantiations that could be transformed according to the mapping description. Instead, a classification defini-

tion is actually mapped to one of the subclasses of the classifier, i.e. a class (*ssd_class*), an actor (*ucd_actor*), or a use case (*ucd_use_case*), as described in the next subsections.

Property. Properties (*cmm_property*) are mapped to the mainly analogous attribute concept (*oo_attribute*) of the object-oriented meta-models. The concept of a data-type selection (*cmm_data_type_selection*) cannot be analogously represented by elements of the object-oriented meta-models. In this case, an extra classifier is instantiated that represents the selection set by having each of its attributes associated with one element of the data-type selection and being used as data-type.

Function and Function Parameter. Functions (*cmm_function*) are mapped to the analogous operation concept (*oo_operation*). Likewise function parameters (*cmm_function_parameter*) are mapped to the analogous object-oriented operation parameters (*oo_operation_parameter*). Only if a function parameter is a return parameter, i.e. the *kind* attribute of the respective *cmm_function_parameter* has the value *return_parameter*, then the function parameter is mapped to the *return_data_type* attribute of the respective *oo_operation* entity.

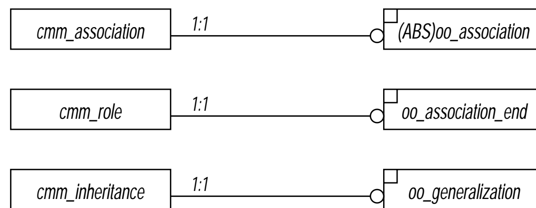

Fig. 6.20: Mapping overview: Common meta-model to object-oriented relationships meta-model

The relationships of the common meta-model are mapped to common object-oriented elements as follows.

Association and Role. An association of the common meta-model (represented by the *cmm_association* entity) is mapped to the analogous object-oriented association concept (represented by the *oo_association* entity). Note that this only describes the principle of the mapping, because *cmm_association* is an abstract entity. Instead, the *cmm_association* entity is actually mapped to one of the subclasses of *oo_association*, namely *ssd_association* for static structure diagrams or *ucd_association*

for use case diagrams. The roles that classifications play in an association (represented by *cmm_role* entities) are mapped to association ends (*oo_association_end*), which also represent a role that a classifier plays in an association.

Inheritance. Inheritance relationships (*cmm_inheritance*) are mapped to the analogous object-oriented concept of generalization (represented by the *oo_generalization* entity).

6.8.2 To Static Structure Diagrams

Figure 6.21 provides an overview of the mapping from elements of the common meta-model to elements of the static structure diagram meta-model presented in Subsection 4.4.4. Note that the mapping to common object-oriented concepts, such as attribute, operation, or generalization, are described in Subsection 6.8.1, and not repeated here. The complete formal EXPRESS-X mapping description of the mapping from the common meta-model to the static structure diagram meta-model can be found in Section B.2 in Appendix B.

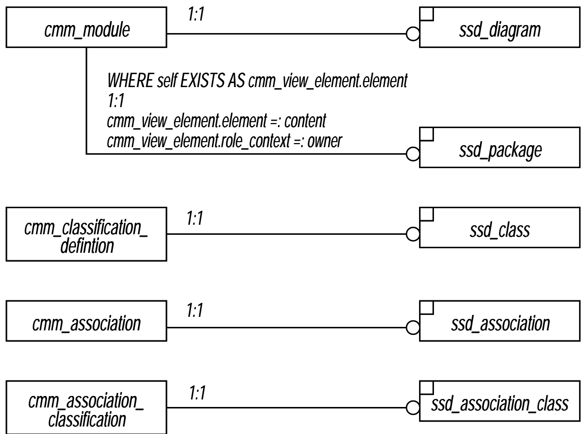

Fig. 6.21: Mapping overview: Common meta-model to static structure diagram meta-model

In detail, the mapping is carried out as follows.

Module. A module as logical grouping of specification elements (represented

by the *cmm_module* entity) is mapped to the a static structure diagram concept (*ssd_diagram*). If a module is itself contained in another module, it acts as a nesting of specification element containers. This is represented in the static structure diagram through packages. Hence, such a module is additionally mapped to the package concept (*ssd_package*) of the static structure diagram meta-model. The content of the package (*content* attribute of *ssd_package*) is described by the module itself. The owner (*owner* attribute of *ssd_package*) is derived from the *role_context* attribute of the *cmm_view_element_role* entity that linked the module to a super-ordinated view.

Classification Definition. The principles that apply for mapping a classification definition to the abstract object-oriented classifier concept are described in Subsection 6.8.1. In the case of a static structure diagram, a classification definition (*cmm_classification_definition*) is mapped to the analogous class concept (*ssd_class*), which is a sub-class of the classifier (*oo_classifier*).

Association. The mapping of an association is principally described in Subsection 6.8.1. In the case of static structure diagrams, associations are mapped to the specialized concept of a static structure association, represented by the *ssd_association* entity.

Association Classification. An association classification (represented by the *cmm_association_classification* entity) is mapped to the analogous concept of an association class (represented by the *ssd_association_class* entity). The referenced *class* is derived from the *classification* reference of the original classification definition, and the *association* is mapped from the analogous *association* attribute of the classification definition.

6.8.3 To Use Case Diagrams

Figure 6.22 presents an overview of how elements from the common meta-model are mapped to elements of the meta-model for use case diagrams, as presented in Subsection 4.4.5.

The mapping is carried out in detail as follows.

Module. A module (*cmm_module*) is mapped to the reference to a use case diagram (*ucd_diagram*), as both represent containers of logically grouped specification elements. Furthermore, if the respective module

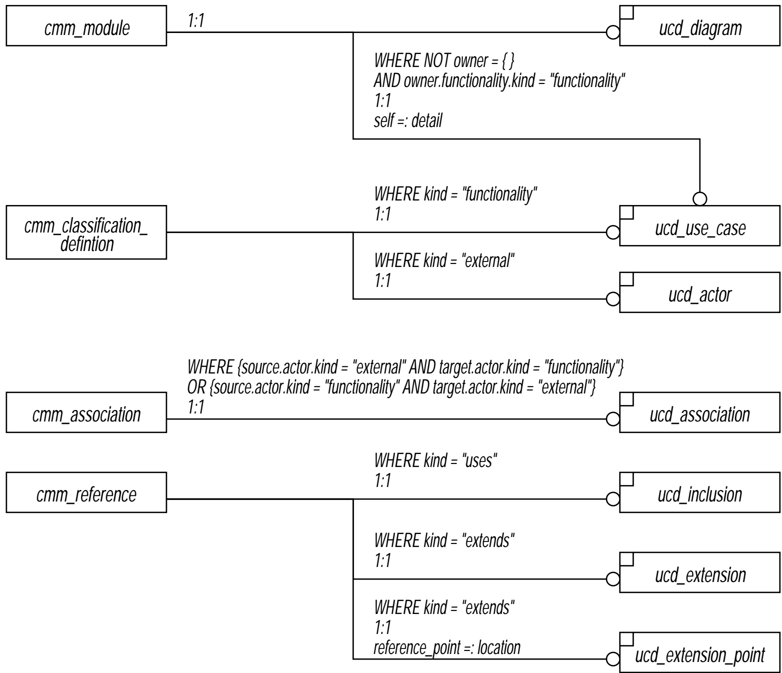

Fig. 6.22: Mapping overview: Common meta-model to use case diagram meta-model

is owned by a classification, i.e. the optional *owner* attribute references a classification that represents a functionality of the system (having set its *kind* attribute to *functionality*), then the use case diagram is a refinement of a higher-level use case. This is represented by referencing the use case diagram through the *detail* attribute of the respective higher level *ucd_use_case* entity.

Classification Definition. If a classification definition (represented by the *cmm_classification_definition* entity) represents a specification of functionality of the system under consideration, i.e. its *kind* attribute has the value *functionality*, then it is mapped to a use case (*ucd_use_case*) in the use case diagram. If a classification definition represents an entity that is external to the system, i.e. the *kind* attribute has the value *external*, then it is mapped to an actor (*ucd_actor*) in the use case diagram.

Association. Associations in a use case diagram are only allowed between actors and use cases. Hence, associations (*cmm_association*) are only mapped to a use case diagram association (*ucd_association*) if the association connects an external classification definition and a functionality classification definition (see the mapping of classification definition above). Other kinds of associations cannot be represented in a use case diagram.

Reference. If a reference (*cmm_reference*) represents an inclusion of a use case in another use case, i.e. the *kind* attribute of the *cmm_reference* entity has the value *uses*, then the reference is mapped to a inclusion relationship (*ucd_inclusion*) in the use case diagram. If a reference represents an extension of a use case, i.e. the *kind* attribute of the *cmm_reference* entity has the value *extends*, then the reference is mapped to an extension (*ucd_extension*) and an extension point (*ucd_extension_point*). The extension point is assigned to the extension through the *extension_point* attribute of the *ucd_extension*, and the specification of the location of the extension point is stored in its *location* attribute, derived from the *reference_point* attribute of the original *cmm_reference* entity.

6.8.4 To Collaboration and Sequence Diagrams

Figure 6.23 outlines the mapping of elements from the common meta-model to elements of the meta-model for collaboration and sequence diagrams as described in Subsection 4.4.6.

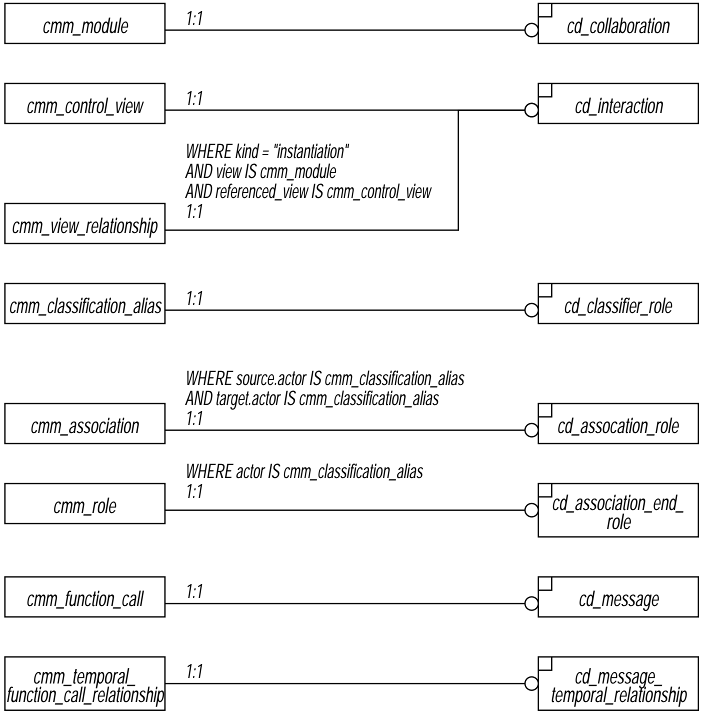

Fig. 6.23: Mapping overview: Common meta-model to collaboration and sequence diagram meta-model

The mapping is carried out as follows.

Module. A module (*cmm_module*) is mapped to a collaboration reference (*csd_collaboration*), as both represent logical groupings of specification elements.

Control View. If a control view (*cmm_control_view*) is an instantiation of a module, i.e. there exists a *cmm_view_relationship* entity that connects both and has the value *instantiation* for its *kind* attribute, then the control view is mapped to an interaction (*cmm_interaction*). The con-

text of the interaction is the related module, i.e. the *context* attribute of the respective *csd_interaction* entity references the *csd_collaboration* mapped from the related module.

Classification Alias. An alias of a classification (*cmm_classification_alias*) is mapped to the analogous concept of a classifier role (*csd_classifier_role*) of the collaboration and sequence diagram meta-model. Note that classification definitions are not mapped into a collaboration or sequence diagram, as those diagrams are not intended to illustrate the internal structure of model elements, rather they are used to illustrate the logical relationships among model elements when being used concurrently.

Association and Role. Associations (*cmm_association*) are mapped to the association role concept (*csd_association_role*) if the associated model elements are both classification aliases. Associations of other kinds are not shown in a collaboration or sequence diagram. Roles (*cmm_role*) are likewise mapped to association end roles (*csd_association_end_role*) only if they are representing a classification alias, i.e. their *actor* attribute references a *cmm_classification_alias* entity.

Function Call and Temporal Function Call Relationship. A function call, represented by the *cmm_function_call* entity, is mapped to the analogous message concept, represented by the *csd_message* entity. The temporal ordering of function calls (cmm_temporal_function_call_relationship) is mapped to the analogous message temporal relationship, represented by the *csd_message_temporal_relationship* entity.

6.8.5 To Statechart Diagrams

Figure 6.24 provides an overview of the mapping from the common meta-model (presented in Chapter 5) to the statechart diagram meta-model presented in Subsection 4.4.7. The complete formal EXPRESS-X mapping can be found in Section B.4 of Appendix B.

The mapping is carried out as follows.

Control View. A control view (*cmm_control_view*) is mapped to a statechart diagram (*scd_diagram*), because a control view specifies a behavioral aspect of a system just as a statechart diagram.

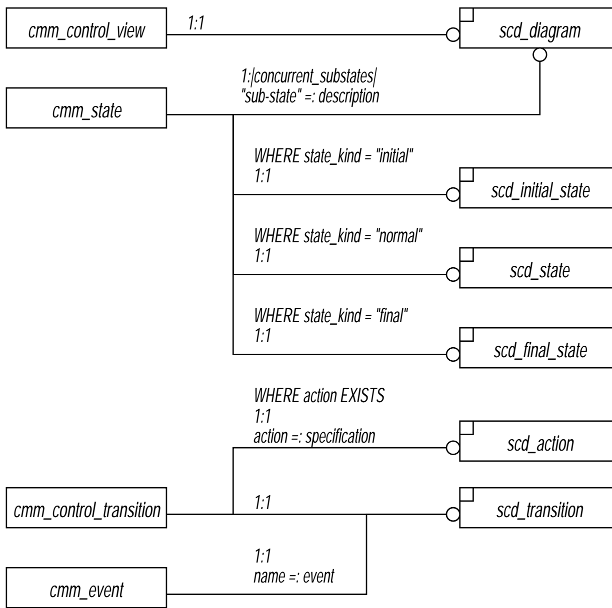

Fig. 6.24: Mapping overview: Common meta-model to statechart diagram meta-model

State. A state of the common meta-model (*cmm_state*) is mapped to either an initial state, a final state or a "normal" state, i.e. a non-pseudo-state that represents an actually possible state of the system. The mapping is decided according to the value of the *state_kind* attribute of *cmm_state*. The value *initial* yields an initial state (*csd_initial_state*), *final* a final state (*csd_final_state*), and *normal* a normal (non-pseudo) state (*csd_state*). The statechart meta-model does not provide support for composite concurrent states. If a state comprises of concurrent sub-states (referenced through the *concurrent_substates* attribute of the *cmm_state* entity), then each sub-state is mapped to a separate statechart diagram (*scd_diagram*) representing the respective sub-state.

The state-transition diagram does not support relationships among state-transition diagrams themselves, which implies that connections from a common meta-model state to its concurrent sub-states are lost after the transformation if not the sink tool keeps track of these relationships.

Control Transition and Event. A control transition (represented by the entity *cmm_control_transition*) is mapped to the analogous transition concept of statechart diagrams (*scd_transition*). The event associated with the control transition (represented by the *cmm_event* entity) is mapped to the *event* attribute of the respective *csd_transition* entity. The declaration of an action (optional *action* attribute of *cmm_control_transition*) is mapped to an action (*csd_action*) in the statechart diagram meta-model, associated with the respective transition through the *action* attribute of *csd_transition*.

6.8.6 Summary

The mapping from the common meta-model to elements of object-oriented specification techniques is in many cases straightforward, because the common meta-model resembles the principle of integrating data and function, as in the object-oriented perspective.

The most important concepts, the classification and the association, have analogous counterparts commonly used by the object-oriented specification techniques, namely the classifier and the association.

In static structure diagrams, the hierarchical relationship between views is interpreted as a packages containing specializations. The classification definition concept maps to the class concept and associations are mapped to the analogous associations used in static structure diagrams.

The mapping to use case diagrams is characterized by mapping the classification definitions that represent a piece of functionality of the system to use cases. An association is mapped to a use case association if it connects an external element with a functionality. References between classifications are interpreted as use case extensions and inclusions, depending of the kind of reference.

As for the mapping from the meta-model of collaboration and sequence diagrams to the common meta-model, the same principle of using classification aliases instead of direct references to classifications (see Subsection 6.7.6) also applies for the reverse transformations. Hence, a classification alias is mapped to a classifier role and an association among aliases is mapped to an association role. Furthermore, for sequence diagrams, the function calls are mapped to the analogous message concept in interactions, which are illustrated through sequence diagrams.

Transformations from the common meta-model to statechart diagrams are characterized by the straightforward mapping of the state concept in the common meta-model to the different states in the statechart diagram, determined by the kind of the original state. Furthermore, control transitions are mapped to the analogous transition concept.

In summary, just as for structured specification techniques, representations in the common meta-model cannot be fully transformed to representations under the meta-model of each object-oriented specification technique. This is due to the semantically richer common meta-model that provides support

for specification aspects that are not fully supported by all object-oriented specification techniques. However, the mapping from the common meta-model to the meta-models of the object-oriented specification techniques is more straightforward than the mapping to structured techniques, which can be accounted to the similarity of integrating data and function in both, the common meta-model and in object orientation.

6.9 Discussion

Due to the fact that the common meta-model is semantically richer than the single constituting meta-models of the underlying specification techniques, the transformations from representations under the common meta-model to a representation in one of the specification technique is more difficult than the inverse transformations. This is natural, because the common meta-model provides in most cases support for more specification details than each of the constituting meta-models.

As a result, a model according to the common meta-model often cannot be fully or only conditionally transformed into a representation in one specific specification technique. However, the goal of the work is still attained as the major specification elements of a specific aspect of the system specification still can be transformed to and from the common meta-model, and the exchangeability of data across different aspects, e.g. modifying an original entity-relationship model with a data-flow diagramming tool, was not intended.

7. APPLICATION EXAMPLE

In this chapter, the proposed approach is applied to a real-world example in order to illustrate an implementation of the approach and show how structured and object-oriented tools can be used collaboratively through the common meta-model. Therefore, the example specification data is transformed between representations of different meta-models described in Chapter 4 and the common meta-model described in Chapter 5, using the mappings described Chapter 6.

7.1 Scenario

The scenario for illustrating the interoperability of structured and object-oriented specification tools through data exchanges using the common meta-model implements the approach presented in Subsection 1.4.1. The actual exchanged specification data is taken from the SEDRES-2 project (see Subsection 1.3.3) and is provided by British Aerospace Systems. The specification data will be explained in more detail in Subsection 7.2.

Figure 7.1 provides an overview of the scenario, based on the approach outlined in Subsection 1.4.1 (Figure 1.1). The actors in the scenario illustrated in Figure 7.1 are the following.

Statemate. The specification data that is exchanged in the scenario has been created using Statemate MAGNUM 2.1 by i-Logix. Statemate is a specification tool that implements structured specification techniques. It provides activity charts (comparable to data-flow diagrams) and statecharts (an extension of Harel's statecharts) to specify a system's structure and behavior. Statemate uses a central data dictionary in order to allow for the consistent use of data structures and definitions across the complete specification.

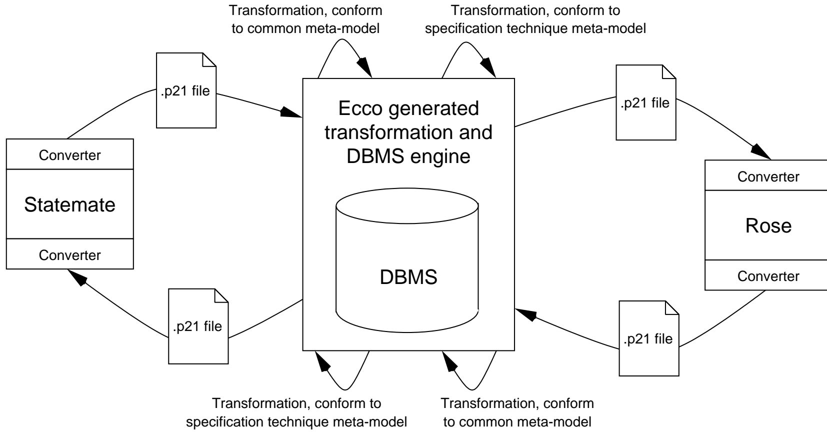

Fig. 7.1: Data exchange scenario

Rose. The sink for the specification data is the object-oriented specification tool Rose 2001 for UNIX by Rational, release version 2001.03.00. Rose follows the UML specification to a large extent (note that the three original authors of the UML are all affiliated with Rational). Its meta-model is almost congruent with the object-oriented meta-models presented in Section 4.4 of Chapter 4.

Ecco application. The central application of the scenario is generated using the Ecco toolkit v2.3 by PDTec. The Ecco toolkit allows for generating repositories on the basis of EXPRESS models and EXPRESS-X transformations. The generated application implements a database management system (DBMS) on the basis of the meta-models presented in Chapter 4 and the common meta-model as presented in Chapter 5. Furthermore, it implements the transformations described in Chapter 6 that transform the specification data between representations under the different meta-models.

Converters. In modification of the proposed approach in Subsection 1.4.1 (Figure 1.1), the scenario employs converters instead of built-in export and import interfaces of the involved tools. This change is due to the unavailability of tool programming interfaces and tool source codes. Thus, the interfaces are implemented as stand-alone programs that directly work on the tool databases and produce / consume data in the form of files. Being directly interfaced with the tools, the converters also take care of transforming the specification data from the

respective tool-internal representation to a representation under the meta-model of one of the desired specification techniques. For example, the Statemate activity charts are transformed into data-flow diagram representations before being forwarded to the Ecco application.

Data exchange files. The specification data is exchanged between the applications in the form of ISO 10303-21 conform files (short: part-21 files).

Statemate and Rose have been selected for the example data exchanges in this chapter, because their meta-models are close to the generic meta-models presented in Chapter 4. Furthermore, both are using en easy to understand clear text native file format.

Following the scenario, the Statemate specification data will first be converted from the Statemate-internal representation to a representation under the structured specification techniques presented in Chapter 4.3, depending of the respective type of Statemate diagram (see Subsection 7.3 for more details). The converted specification data is then stored in part-21 file, which is imported from the Ecco-generated application. The application transforms the specification data from its representation according to the meta-model of the respective specification technique to a representation under the common meta-model, following the respective mapping described in Chapter 6. For the import into the sink tool, the steps are analogously taken in reverse order. The representation under the common meta-model is first transformed within the Ecco-generated application to a representation of the desired sink specification technique's meta-model and stored again in a part-21 file. The file is then imported by the converter that turns the specification data into a Rose-internal representation, which then can be opened by Rose. In order to transfer the Rose specification back to Statemate, the above steps have to be performed analogously.

7.2 Example Outline

The specification data used for the data exchanges in this chapter is based on a real-world example taken from the SEDRES-2 project (see Subsection 1.3.3), provided by British Aerospace Systems. The specification data has been produced using Statemate MAGNUM. It specifies the control software necessary for the safe operation of an aircraft's nose gear landing system. The nose gear mainly consists of the nose leg, the nose wheel, two doors (port bay door and starboard bay door), and a number of sensors, actuators and locks, as illustrated in the schematic overview in Figure 7.2[1].

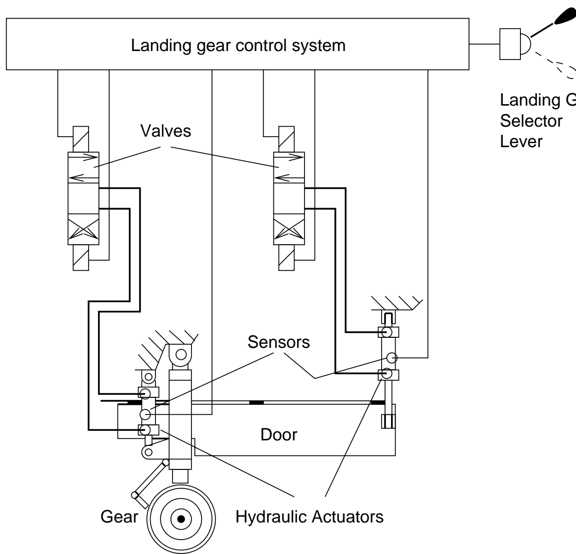

Fig. 7.2: Schematic overview of the landing gear system (adopted from [NTS99])

The landing gear control system monitors and processes commands from the cockpit landing gear panel, comprising the cockpit indicator panel. The cockpit indicator panel displays the current status of the landing gear and the landing gear selector lever, which is used by the pilot to command extension and retraction of the landing gear. The nose leg movement is accomplished by an actuator, which also is controlled by the landing gear control system. Proximity sensors and a weight-on-wheel sensor are monitored in order to determine the current status of the nose leg. Safety locks ensure

[1] Courtesy of Jan-Erik Strömberg, based on [NTS99]

that the nose leg is latched in fully extended respectively fully retracted position, which is commanded by the landing gear control system depending on the proximity sensors' values. During the operation of the nose leg, the bay doors protecting the nose leg in retracted position are operated in analogous fashion. For each door, there is one actuator controlling the movement of the door, proximity sensors for the current position of the door, as well as safety locks maintaining the door in fully opened respectively fully closed position.

The landing gear control system has been designed using the following four-layered pattern.

Equipment Interface. This layer comprises the digital and analog sensors and actuators and serves as interface to the remaining elements of the system.

Equipment Control and Monitor. This layer encapsulates monitoring and controlling major system components, e.g. monitoring the actual displacement of an element and taking actions in order to move it to its desired displacement.

Equipment Management. This layer specifies the control and monitor activities necessary to respond to the sequencing of the landing gear system and directs the voting between dependent systems, e.g. port and starboard bay door.

Operational Management. This layer specifies the sequencing of operations necessary to provide the high-level functionality of the system, e.g. extending or retracting the landing gear.

In the specification, this four-layered pattern is implemented through four Statemate activities, each representing one level of the design, as shown in Figure 7.3.

7.3 Example Specification Data Exchanges

This section shows how mappings, presented in Chapter 6, are applied to fragments of the design of the above presented landing gear control system in order to exchange specification data between Statemate MAGNUM and Rose 2001.

7.3.1 Landing Gear Control System

The example in this subsection illustrates, according to the proposed principle, all steps of a complete transformation of a specification from its representation in the source tool (Statemate MAGNUM) to its representation in the sink tool (Rose 2001).

In the scenario presented above, the transformation of the Statemate landing gear control system activity diagram (see Figure 7.3) to a representation in Rose comprises the following steps.

1. Conversion of the specification from its representation in Statemate to a representation under the meta-model of one of the specification techniques presented in Chapter 4. In this case, the landing gear control system diagram is a Statemate activity diagram, which is semantically closest related to the data-flow diagram. Hence, the first step comprises of converting the native Statemate representation to a representation as data-flow diagram.

2. Transformation of the specification from its data-flow diagram representation to a representation under the common meta-model, using the mapping description in Subsection 6.5.1.

3. Transformation of the specification from the common meta-model representation to a representation, which is semantically close to the Rose meta-model. In this case, the static structure diagram is closest as mostly classifications and associations, i.e. static structure elements, are employed in the example specification. Hence, this step comprises of transforming the specification according to the transformation described in Subsection 6.8.2.

4. Conversion of the specification from the static structure diagram representation to a native Rose representation. In this case, the conversion

is straightforward, because both Rose and the static structure diagram meta-model are based on the UML and have a similar meta-model.

The following paragraphs describe and illustrate the different representations of the specification under the meta-models used in each of the above presented steps.

Figure 7.3 shows the original specification as Statemate activity diagram.

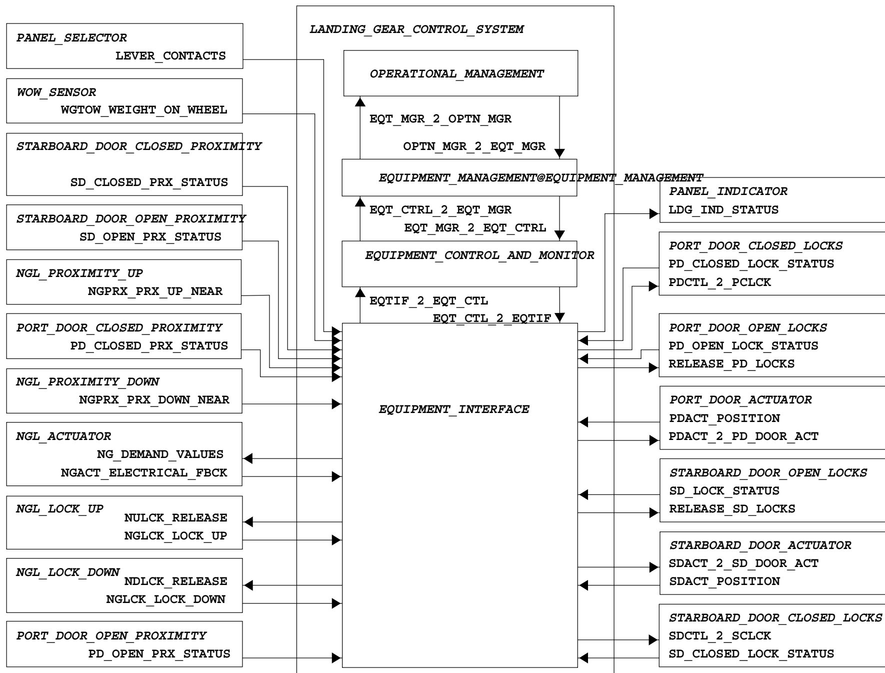

Fig. 7.3: Landing gear control system: Original Statemate representation

The specification models the sensors and actuators as well as the main components of the landing gear control system. Sensors, such as the panel selector (*PANEL_SELECTOR* activity) or the weight-on-wheel sensor (*WOW_SENSOR* activity), provide input to the equipment interface(*EQUIPMENT_INTERFACE* activity), depicted by directed arrows from the sensor to the equipment interface. Actuators, such as the nose gear lock actuator (*NGL_ACTUATOR* activity) or the nose gear upper lock (*NGL_LOCK_UP* activity), receive signals from the equipment interface, depicted by directed arrows from the equipment management to the respective actuator. The arrows are labeled with the type of data that flows between the related activities. All sensors

and actuators are considered to be external to the landing gear control system. The landing gear control system consists of four components, namely the equipment interface, the equipment control and monitor activity, the equipment management, and the operational management activity, which are all further refined through respective sub-charts.

Figure 7.4 shows a fragment of the serialized native file representation of the Statemate activity diagram in Figure 7.3. Note that attributes dealing with layout properties have for the greater part been manually removed in order to improve readability. Also, note that Figure 7.4 presents only a fragment of file representation of the specification in Figure 7.3.

The equipment interface activity is represented in lines 1 to 15 as internal activity (see type *INTERNAL*, line 3), i.e. it is considered to be an integral part of the system. The nose gear down lock is represented in lines 17 to 22 as external activity (see type *EXTERNAL*, line 19). Its occurrence in the diagram is specified in lines 24 to 33, i.e. Statemate distinguishes between the conceptual model elements and a representation of the model as diagram. Lines 35 to 49 specify the data-flow from the nose gear down lock to the equipment interface (through its activity occurrence).

Figure 7.5 illustrates the results of the first step of the transformation, namely the conversion from the native Statemate representation (see Figures 7.3 and 7.4) to a representation under the meta-model of data-flow diagrams (presented in Subsection 4.3.1). Note that Figure 7.5 only shows a fragment of the transformation of the original specification, namely the activities *PANEL_SELECTOR* and *LANDING_GEAR_CONTROL_SYSTEM* with its internal activities *EQUIPMENT_INTERFACE* and *EQUIPMENT_CONTROL_AND_MONITOR* as well as their inter-connections through data-flows.

The diagram itself is represented by a *dfd_diagram* entity, to which the contained elements refer through their *owner* attribute. External activities in the Statemate specification, such as the *PANEL_SELECTOR*, represent a data source or a data sink that is external to the system and not further refined in the specification. External activities match the concept of external entities of data-flow diagrams, and thus, are transformed into *dfd_external_entity* entities. Internal activities in the Statemate representation, such as the *EQUIPMENT_INTERFACE*, represent a function (or collection of functions) of the system and thus, match the process concept of data-flow diagrams. Hence, internal activities are transformed into processes, represented by *dfd_process* entities. Data-flows in the Statemate representation match the data-flow concept in data-flow diagrams. Hence, data-flows are transformed

```
1  activity :
2          name : EQUIPMENT_INTERFACE
3          type : INTERNAL
4          parent : LANDING_GEAR_CONTROL_SYSTEM
5          line width : 2
6          color : DARKORANGE WHITE OFF
7          name color : BLACK BLACK OFF
8          name font : Courier 14 BOLD ITALIC
9          name alignment : Left Bottom
10         termination : CONTROLLED_TERMINATION
11         name position :
12                 10.7488015000 7.5000000000
13         graphics coordinates :
14                 ...
15 end activity
16
17 activity :
18         name : NGL_LOCK_DOWN
19         type : EXTERNAL
20         parent : ACTIVITY#0
21         ...
22 end activity
23
24 activity occurrence :
25         name : ACTIVITY_OCCURRENCE#5
26         activity : NGL_LOCK_DOWN
27         ...
28         graphics coordinates :
29                 7.0050315946 1.9999984892
30                 7.0050315946 3.4956706523
31                 0.5000000000 3.4956706523
32                 0.5000000000 1.9999984892
33 end activity occurrence
34
35 arrow :
36         type : DATA_FLOW
37         source : ACTIVITY_OCCURRENCE#5 FROM
38         target : EQUIPMENT_INTERFACE TO
39         line width : 1
40         ...
41         graphics coordinates :
42                 ...
43         angles :
44                 ...
45         label : NGLCK_LOCK_DOWN
46         end label
47         label position :
48                 3.2500000000 2.2500000000
49 end arrow
```

Fig. 7.4: Landing gear control system: Fragment of the Statemate native file representation

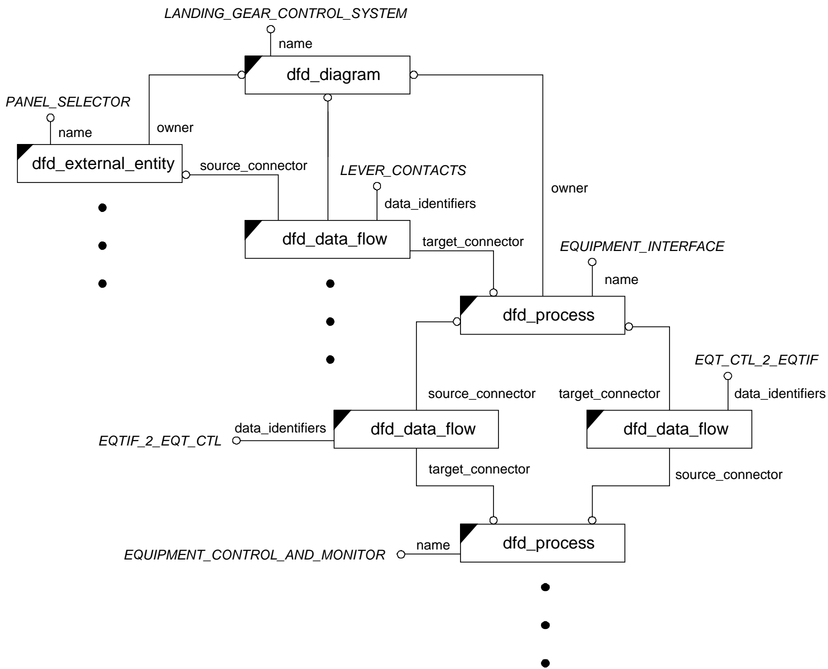

Fig. 7.5: Landing gear control system: Fragment of the data-flow diagram meta-model instantiation

in to representations by *dfd_data_flow* entities. The specification of the data-type of a data-flow is derived from the Statemate data-flow's label and stored in the *data_identifiers* attribute of the *dfd_data_flow* entity representing the data-flow.

Figure 7.6 illustrates the result of the second step, namely the transformation from the data-flow diagram representation to the representation under the common meta-model, according to the mapping description in Subsection 6.5.1. Note that Figure 7.6 only shows a fragment of the transformation result, namely the common meta-model representation of the data-flow diagram fragment shown in Figure 7.5. However, the principle illustrated in the figure applies to all elements of the complete specification.

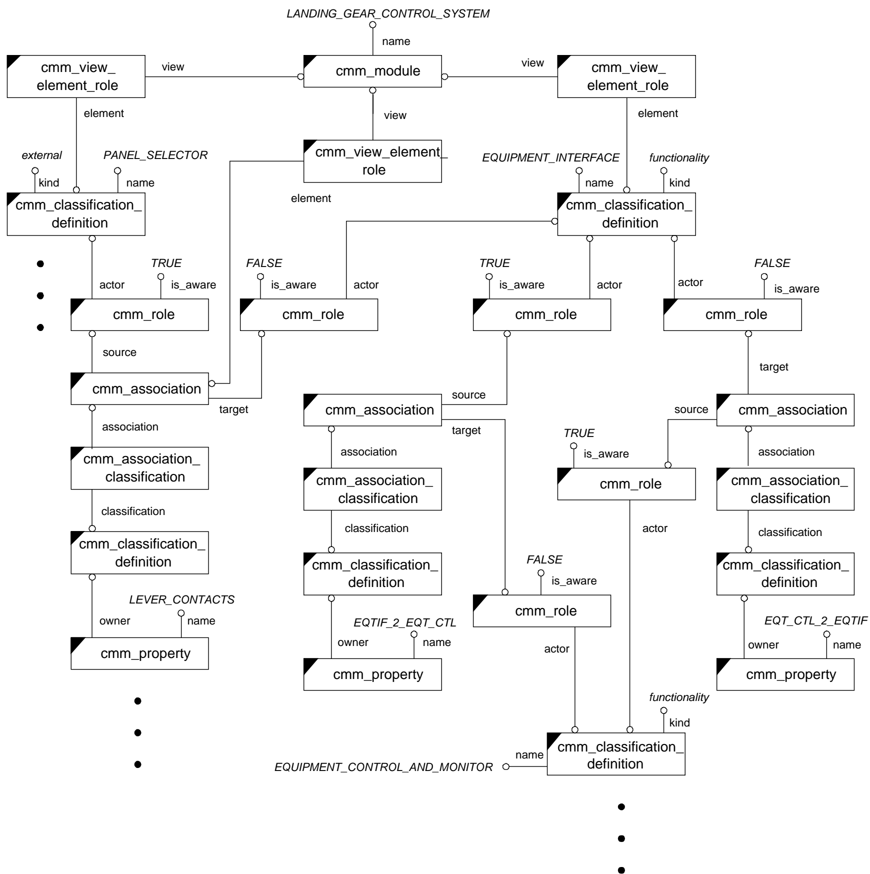

Fig. 7.6: Landing gear control system: Fragment of the common meta-model instantiation

The details of the mapping from data-flow diagram elements to representations under the common meta-model are described in Subsection 6.5.1. The corresponding EXPRESS-X mapping can be found in Section B.1 in Appendix B. Under the common meta-model, the diagram is represented by a module (*cmm_module*). The external entities are transformed into classification definitions (*cmm_classification_definition*), having their *kind* attribute set to *external*, depicting that they are external to the system. Data-flows are transformed into associations, represented by *cmm_association* entities, referencing the members of the association through *cmm_role* entities. The data-types of a data-flow is transformed into properties (*cmm_property*) of an association classification (*cmm_association_classification*) which is attached to the respective association. For example, the data-type *LEVER_CONTACTS* carried by the original data-flow between *PANEL_SELECTOR* and *EQUIPMENT_INTERFACE*, is represented by a *cmm_property* entity with the *name* attribute set to *LEVER_CONTACTS*. The property is owned by a classification definition (*cmm_classification_definition*), which in turn is associated with the respective association through a *cmm_association_classification* entity.

Figure 7.7 illustrates a fragment of the result of the third step, namely the transformation from the common meta-model conform representation to a representation under the static structure diagram meta-model.

The details of the mapping from elements of the common meta-model to elements of the static structure diagram meta-model are described in Subsection 6.8.2. The corresponding formal EXPRESS-X mapping can be found in Section B.2 of Appendix B. Due to the similarity of the common meta-model and the static structure diagram meta-model (with respect to the elements used in the example specification), the resulting static structure diagram representation is similar to the common meta-model representation. The diagram itself is represented by a *ssd_diagram* entity. Classifications are transformed into classes, represented by *ssd_class* entities. Associations and attached association classifications are transformed into the analogous association and association class concepts of the static structured diagram meta-model.

The result of the final step of the transformation from the Statemate representation to a Rose representation, namely the conversion from the representation as static structure diagram to a Rose native file representation is illustrated in Figures 7.8 and 7.9. Note that both figures again only show a fragment of the respective specification representation.

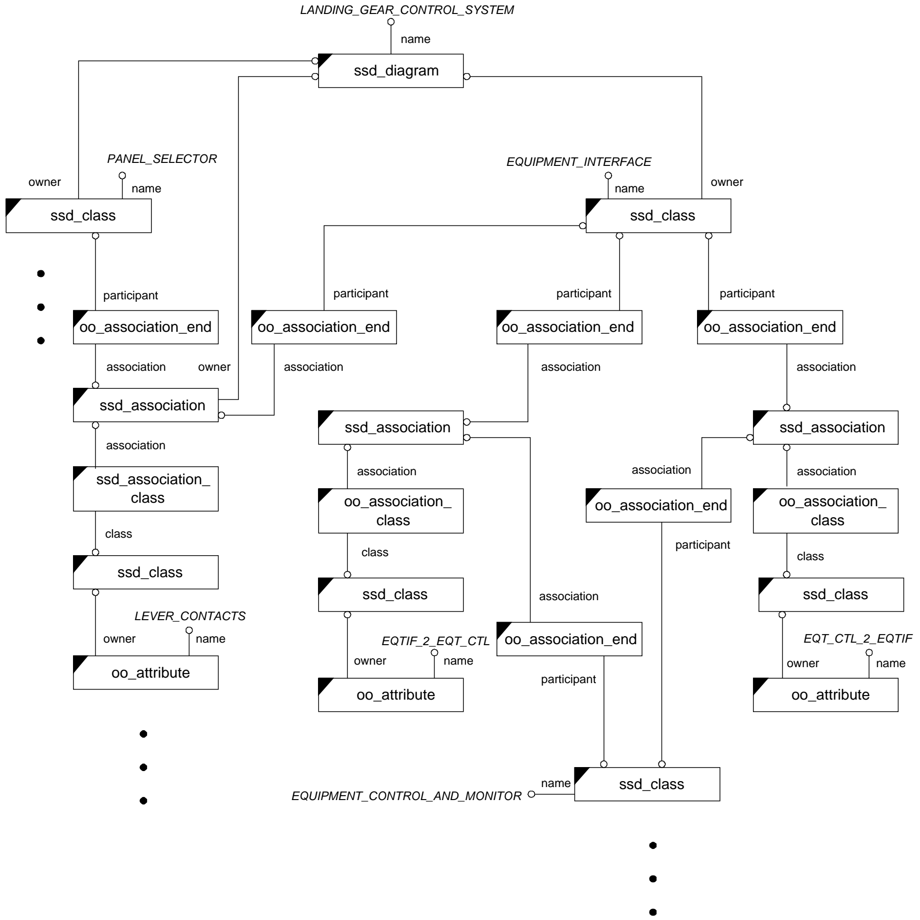

Fig. 7.7: Landing gear control system: Fragment of the static structure diagram meta-model instantiation

```
1              ...
2
3              (object Class_Category "LANDING_GEAR_CONTROL_SYSTEM"
4                  quid                "3D74A7C70236"
5                  exportControl          "Public"
6                  logical_models          (list unit_reference_list
7                      (object Class "$UNNAMED$4"
8                          quid                "3D74A8FC0220"
9                          class_attributes          (list class_attribute_list
10                             (object ClassAttribute "LEVER_CONTACTS"
11                                 quid                "3D74A8FC0221")))
12                     ...
13
14                     (object Class "PANEL_SELECTOR"
15                         quid                "3D74A7ED03CC")
16                     (object Class "EQUIPMENT_INTERFACE"
17                         quid                "3D74A8110002")
18
19                     ...
20
21             (object Association "$UNNAMED$1"
22                 quid                "3D74A8330296"
23                 roles               (list role_list
24                     (object Role "$UNNAMED$2"
25                         quid                "3D74A83400F3"
26                         supplier         "Logical View::
27                                          LANDING_GEAR_CONTROL_SYSTEM::
28                                          EQUIPMENT_INTERFACE"
29                         quidu               "3D74A8110002"
30                         is_navigable          TRUE)
31                     (object Role "$UNNAMED$3"
32                         quid                "3D74A83400F4"
33                         supplier         "Logical View::
34                                          LANDING_GEAR_CONTROL_SYSTEM::
35                                          PANEL_SELECTOR"
36                         quidu               "3D74A7ED03CC"))
37                 AssociationClass          "$UNNAMED$4")
38
39             ...
```

Fig. 7.8: Landing gear control system: Fragment of the Rose native file representation

Reading the part-21 file containing the specification according to the static structure diagram meta-model, the code generator transforms the specification into a native Rose file representation as shown in Figure 7.8. The static structure diagram meta-model (see Subsection 4.4.4) has many similarities with the Rose meta-model. Hence, the transformation of most elements of the static structure diagram representation into the Rose representation is straightforward. Classes are transformed into representations of the class concept of Rose, e.g. the *ssd_class* representing the original *PANEL_SELECTOR* activity is transformed into the class instantiation in lines 14 and 15 of Figure 7.8. Likewise, associations are transformed into the analogous Rose associations, including the association ends, which are represented as *Role* objects in the Rose native file (see lines 23 to 36 of Figure 7.8). Figure 7.9 shows a fragment of the graphical interpretation by Rose of the generated file. Note that layout information is not supported in the common meta-model. Hence, the layout in Figure 7.9 has been manually modified in order to arrange the diagram elements in a visually meaningful configuration.

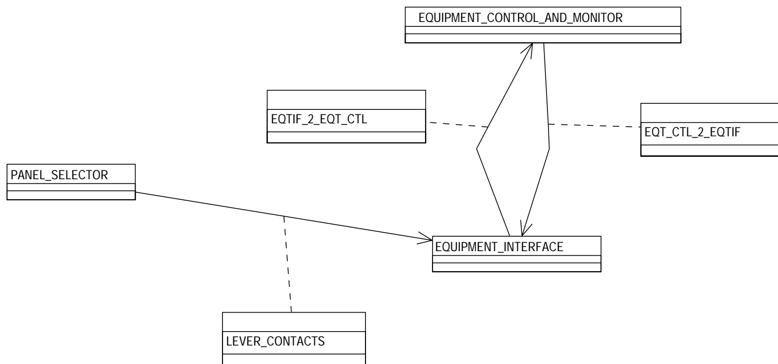

Fig. 7.9: Landing gear control system: Fragment of the Rose representation

7.3.2 Nose Gear Manager

This subsection 7.3.2 focuses on illustrating the support of the common meta-model for nesting levels of detail of a specification. Also, the principle of transforming specifications between structurally different meta-model representations is demonstrated.

Figure 7.10 provides an overview of the nose gear manager which controls the extension and retraction of the nose gear. The focus of this example is on the control activity *NG_MANAGER_CTL* and how it is internally connected with its statechart refinement shown in Figure 7.11.

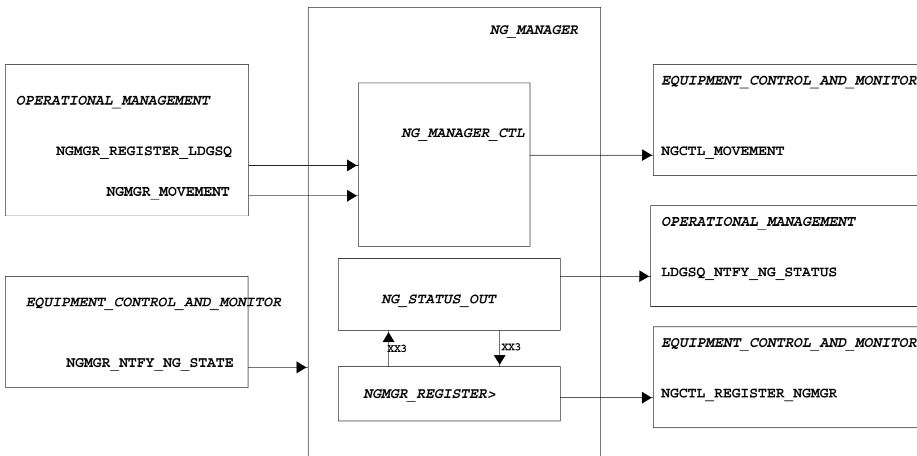

Fig. 7.10: Nose gear manager: Statemate representation

The single steps of the transformation of the original Statemate representation into a Rose representation are analogous to the transformation presented above in Subsection 7.3.1. First, the Statemate representation is converted into a state-transition diagram. Second, the state-transition diagram is transformed into a common meta-model representation, according to the transformation description in Subsection 6.5.4. The corresponding EXPRESS-X mapping for this transformation can be found in Section B.3, in Appendix B. Third, the common meta-model representation is transformed into an object-oriented statechart diagram, according to the description in Subsection 6.8.5, using the EXPRESS-X mapping of Section B.4, Appendix B.

The (intentional) lack of support for composite concurrent states in the object-oriented statechart diagram meta-model (see explanation in Subsec-

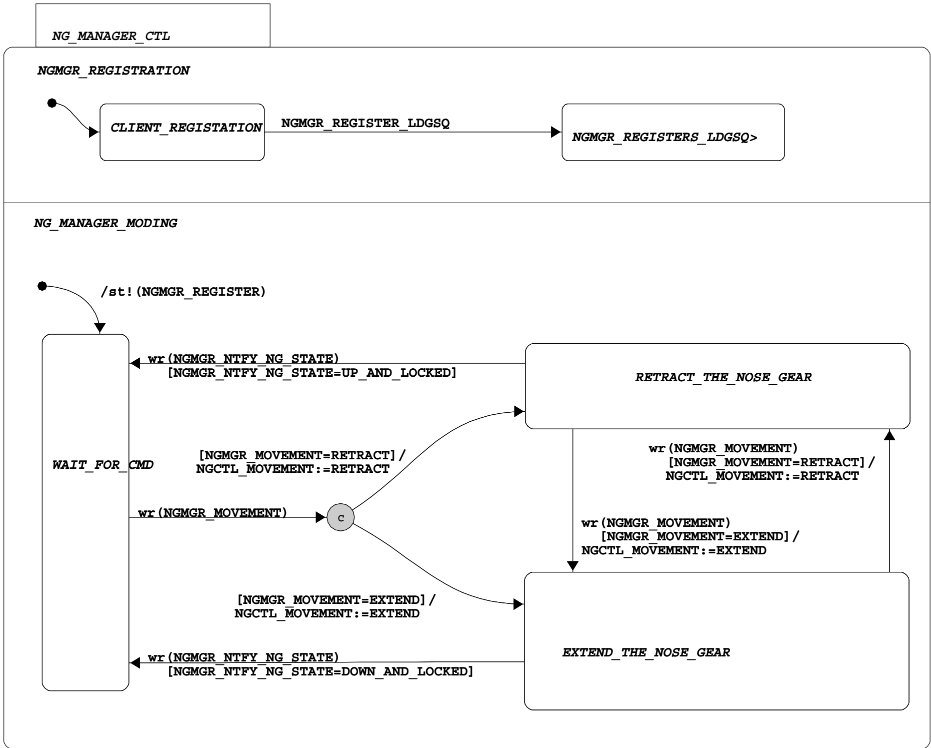

Fig. 7.11: Nose gear manager control: Original Statemate representation

tion 4.4.7) requires splitting the original *NG_MANAGER_CTL* state as described in Subsection 6.8.5. The splitting leads to three statechart diagrams: One representing the top-level statechart diagram with the *NG_MANAGER_CTL* state and two statechart diagrams each representing one of the concurrent sub-states of the *NG_MANAGER_CTL* state. The result of the last step of the transformation, i.e. the conversion from the static structure diagram to a Rose representation, is illustrated in Figures 7.12, 7.13 and 7.14.

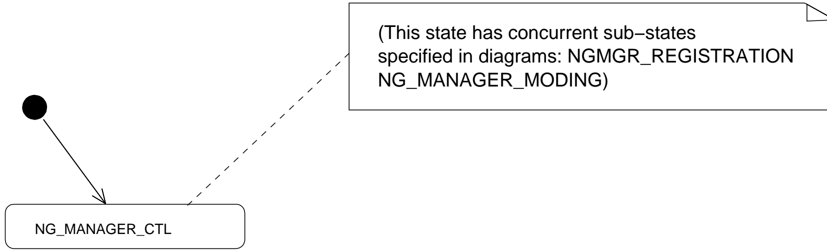

Fig. 7.12: Nose gear manager control: Rose representation, top-level statechart

Figure 7.12 shows the top-level statechart diagram. The note attached to the state *NG_MANAGER_CTL* indicates that the state is decomposed into the sub-diagrams *NGMGR_REGISTRATION* and NG_MANAGER_MODING, each representing a concurrent sub-state of *NG_MANAGER_CTL*. This note is generated from the *description* attribute of the *scd_state* entity representing the *NG_MANAGER_CTL* state. Furthermore, the concurrency is also indicated by a note in each sub-diagram (see Figures 7.13 and 7.14). These notes are generated from the *description* attribute of the respective *scd_diagram* entities representing the concurrent sub-states.

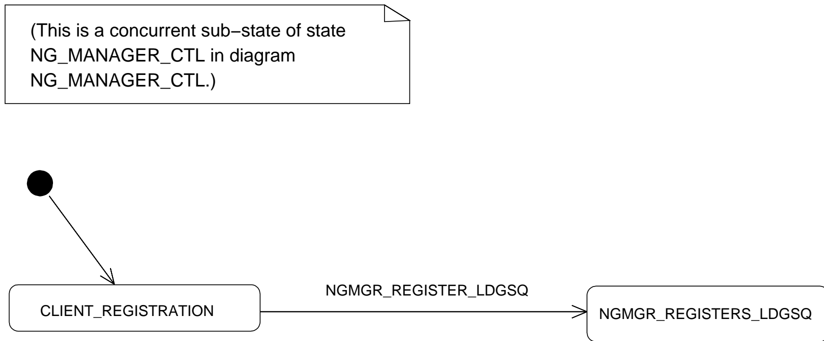

Fig. 7.13: Nose gear manager control: Registration statechart, Rose representation

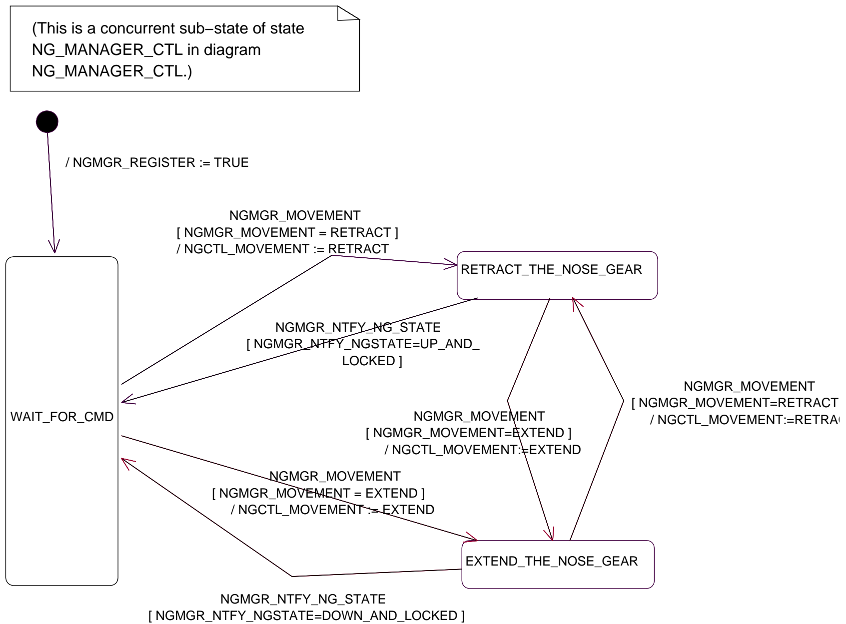

Fig. 7.14: Nose gear manager control: Moding statechart, Rose representation

7.4 Summary and Evaluation

The example transformations presented above show that the principle presented in this thesis can be implemented and used to enable data exchanges between structured and object-oriented specification tools. They also show that the principle is adaptive to structural differences between the underlying meta-models of the transformation source and sink representations.

However, conserving the semantics of a specification during transformations is strongly dependent on the semantic overlap between the source and the sink meta-models as well as on the transformations between the meta-models. For example, if two meta-models represent two different specification aspects, e.g. the entity-relationship diagram meta-model and the statechart diagram meta-model, then their semantic overlap is usually small. Hence, retaining the semantics of an entity-relationship diagram after a transformation into a representation as statechart diagram is difficult. In the example above, the semantic overlap between the data-flow diagram meta-model and the static structure diagram meta-model is also small. However, the major elements of the original representation can also be found in the sink representation.

Besides the semantic overlap of directly corresponding concepts, the quality of the transformation results depends also on how far the transformations succeed in converting a concept of the source meta-model into a semantically equivalent representation in the sink meta-model. In the example in Subsection 7.3.2, this has been illustrated by transforming the composite concurrent state concept of the common meta-model into a semantically similar construct, composed from elements of the object-oriented statechart diagram meta-model. However, replacing the original formal specification of concurrency (by using the dedicated concept of a composite concurrent state) by a formally weaker concept (informal notes attached to the split-up elements) has intrinsic consequences. Decoupling the concurrent parts and keeping them in different diagrams removes the obvious concurrent character of the specification. Removing the concurrency notes, e.g. by accident, would immediately remove the concurrency, which could not be as easily removed in the original representation. As a result, a transformation, even if it technically preserves the semantics of the single specification elements, may result in a representation that changes the meaning of the specification that is encoded in its original layout.

8. CONCLUSIONS

This chapter summarizes and evaluates the approach presented in this thesis. Also, possible extensions of the approach and future work directions are discussed.

8.1 Summary and Conclusions

This thesis presents a principle of enabling interoperability of different systems engineering tools on the basis of their underlying meta-models. It specifically focuses on the integration of structured and object-oriented specification techniques, in order to enable data exchanges between tools of both kinds (motivated in Chapter 1). Therefore, structured and object-oriented approaches have been contrasted (Chapter 2) in order to illustrate their relatedness as a basis for integration. Then, the concepts of representative structured and object-oriented (UML based) specification techniques have been identified (Chapter 3), meta-modeled (Chapter 4), and integrated in a common meta-model (Chapter 5). Mappings between the meta-models of the specification techniques and the common meta-model have been specified (Chapter 6) in order to allow for transformations of specification data between representations under the different meta-models. The principle presented has been implemented in a data exchange scenario (Chapter 7) in order to demonstrate the applicability of the approach for actual specification exchanges between a structured and an object-oriented specification tool.

The work presented in this thesis allows the following conclusions to be drawn as answers to the questions posted in Section 1.2.

First, as becomes clear in Chapter 2, object orientation is grounded on structured approaches and the major elements of structured approaches can be also found in object-oriented ones. Thus, meaningful mappings between structured and object-oriented specification techniques can be described

that allow transforming the major elements between different structured and object-oriented representations. However, object-oriented elements are more integrated than structured ones. Hence, not all object-oriented concepts can be fully represented by structured concepts. For example, the object-oriented class concept has no semantically equivalent representation in the structured approaches, it can only be represented by using several structured elements in combination.

Second, in order to allow for meaningful specification data exchanges between tools, the mappings between the underlying specification techniques have to be described at the level of their single concepts. Therefore, the considered specification techniques need to be analyzed and decomposed into their constituting concepts (Chapter 3) and the mappings need to be described at the level of single concepts (Chapter 6).

Third, the implementation of the proposed approach employs the STEP framework (ISO 10303), as motivated in detail in Chapter 4. STEP is widely known and used in systems engineering and it provides a set of auxiliary means for describing meta-models and mappings among them. Furthermore, a number of tools are available that allow for generating data exchange and translation applications from STEP conform descriptions, as shown in Chapter 7.

It can be concluded that, as described in Chapter 6, the transformation of specification data from a representation under a specific specification technique to a representation under the common meta-model is easier to accomplish than the inverse transformation. This lies in the fact that each of the examined specification techniques has contributed to the structure of the common integrated meta-model. As a result, the common meta-model comprises more and semantically richer concepts than each of the contributing specification techniques. Furthermore, it is noticeable that specifications under the common meta-model can only to a smaller extent be transformed into one of the structured specification techniques than into one of the object-oriented ones. This can be contributed to the fact that the common meta-model and the object-oriented meta-models, i.e. the UML meta-model, have similar characteristics, namely the integration of structural and functional aspects, whereas this is separated in the meta-models of the structured specification techniques. Generally, the transformation of specifications from structured representations into object-oriented ones (through the common meta-model) are more straightforward and complete than in the inverse direction. However, this thesis focuses more on the

transfer of structured specifications to object-oriented representations than vice-versa in order to allow engineers to reuse legacy (structured) specifications when employing object-oriented techniques.

In conclusion, the proposed principle enables interoperability between structured and object-oriented specification tools and provides the means for automated transformations of specifications between different tools. However, the degree of automation and thus, the effort that has to be further put into manual decisions and activities, is dependent on the semantic overlap between the meta-models underlying the respective tools.

8.2 Future Directions

The principle and the meta-models presented in this thesis provide opportunities for the following extensions.

First, the proposed approach is based on generic concepts rather than on actual meta-models of specific tools. Hence, two additional steps are necessary to implement the approach, namely converting the specification from the tool internal representation of the source tool into the representation under one of the generic meta-models and likewise, converting the representation back from a representation under another generic meta-model into the internal representation of the sink tool, as shown in Chapter 7. These additional conversions could be avoided if actual meta-models of specific tools were used instead of generic meta-models of specification techniques. However, this thesis aims at presenting a generic principle, independent of specific tools.

Second, the meta-models provided in Chapter 4 do, for the greater part, only cover the major concepts of the respective specification techniques. The meta-models could be refined in order to also support details of the respective specification techniques. For example, the object-oriented meta-models in Chapter 4 only support binary associations, whereas the UML allows for n-ary associations. Also, the different semantics of behavior specification techniques could be added, e.g. in order to support the fine-grained semantic differences between different statechart representations. Furthermore, this thesis discusses only concepts from analysis and design. In order to support also specifications of an instantiated system, respective techniques could be added, e.g. the implementation diagrams of the UML that are used to illustrate components of the system and their run-time deploy-

ment. Furthermore, the mappings in Chapter 6 currently allow for only one possible transformation of each concept. However, in some cases, several alternative mappings could be provided in order to let the user choose the most appropriate one.

Third, in order to maintain consistency of the data in the central repository (Chapter 7), it is necessary to introduce additional functionality in the central database management system that identifies specification elements and puts them into the correct location in the global specification. This may be either automatically or manually supported. For example, if a specification tool exports a part of the specification from the repository, the exported specification elements could be tagged with identifiers in order to be put back in the correct location within the global specification in the repository when re-imported.

Besides the above proposed extensions to the approach, the work could principally be useful for a number of considerations about integrating specification techniques of different domains, e.g. the ongoing AP-233 or SE DSIG activities described in Subsections 1.3.3 and 1.3.4, respectively. Furthermore, the approach could be employed to implement the theory of mappings described in literature, e.g. the mapping from Petri-nets to statechart representations described in [CL98].

Also, the Semantic Web will probably become the prevailing platform for describing, discovering, integrating and exchanging information over the internet. From a long-term perspective, the work presented in this thesis could be ported to the Semantic Web framework, as already proposed for parts of AP-233 in [AP02], in order to gain from the ongoing research in this area. One central aspect of the Semantic Web initiative is the use of ontologies for describing the semantics of data in order to create common understanding of information semantics and automate data exchanges between different instances. Expressed in the terminology of the Semantic Web, the work presented in this thesis can be described as creating ontologies (the meta-models of specification techniques and the common meta-model) and exchanging and transforming data between those ontologies, i.e. integrating or composing ontologies.

APPENDIX

A. TERMINOLOGY

This chapter provides definitions of terms used in this thesis. Note that the thesis makes strict use of UML terminology for most object-oriented concepts, e.g. classifier, association, or association class. See [Gro01] for the definition of those concepts.

Abstract element. Abstract elements are elements that are not intended to be instantiated, but used to model common features of its subelements in an inheritance hierarchy. For example, in order to make use of an abstract class, first a non-abstract subclass has to be derived from it.

Approach. Approach refers either to the solution approach used in this thesis (see Subsection 1.4.1) or to a family of methods and techniques, e.g. object orientation.

Aspect. An aspect represents a fragment of a specification. For example, the data aspect focuses on data structures and the function aspect considers the functional hierarchy.

Concept. A concept is an element of a specification technique. For example, class is a concept of the object-oriented class diagram.

Diagram. A diagram is the specific visualization of a specification technique, with defined semantics for the elements in the diagram. For example, the entity-relationship diagram is the visualization of entity-relationship modeling.

Feature. A feature is a part of a higher-level concept. Features can be structural or behavioral. For example, an object-oriented class may consist of attributes (structural feature) and operations (behavioral feature).

Inheritance. Inheritance describes the relationship between a more general and a more specific element, whereby the more specific element obtains (inherits) all features of the more general element and may extend

them with its own features.

Mapping. A mapping describes how one element of a meta-model is represented by elements of another meta-model. This term follows the naming in EXPRESS-X, where the transformations between different representations are described as mappings.

Object Orientation. When referring to object orientation, this thesis mainly refers to concepts and diagrams of the Unified Modeling Language UML [Gro01] if not stated otherwise.

Process. Process either refers to a process in a data-flow diagram (see Subsection 3.1.2) or to a engineering process, i.e. a specific sequence of activities and their interdependencies in order to produce a product.

Software Engineering. Software engineering is an engineering discipline with software as products. In this thesis, software engineering is often separated from systems engineering in order to distinguish the different approaches and requirements of both. However, both are closely related, and software engineering can be considered to be a more specific discipline within systems engineering.

Specification Technique. The term specification technique is used in this thesis to summarize a set of concepts together with a visualization, usually a diagram, that is employed for designing a certain aspect of a system.

Statechart / State-Transition Diagram. In this thesis, the term statechart is used to refer to the object-oriented statechart diagram (see UML specification, [Gro01]). The term state-transition diagram is used for the visualization of finite state machine as structured specification technique.

Structured. The term structured is used to refer to non-object-oriented specification techniques or concepts. In colloquial language, object orientation is also structured; however, in this thesis, the term structured is used to distinguish object-oriented and non-object-oriented techniques and concepts.

Super-class / Sub-class. A super-class is the more general element in a class inheritance (see inheritance), whereas sub-class refers to the more specific element.

Super-element / Sub-element. A super-element is an element of the com-

mon meta-model (see Chapter 5) that comprises the semantics of its constituents, the sub-elements.

Systems Engineering. Systems engineering deals with all aspects of a system and their inter-relationships. It is a high-level engineering discipline that comprises lower-lever disciplines, such as mechanical engineering, electrical engineering, or software engineering. In this thesis, the term systems engineering is especially used to distinguish the different requirements and approaches of engineering pure software systems (in software engineering) and systems in general.

Tool. A tool (or specification tool) is a computer program that is used to specify a system. A tool implements one or more specification techniques. For example, the tool Rose (by Rational) implements specification techniques of the Unified Modeling Language [Gro01].

Transformation. A transformation is a conversion of a specification representation under one meta-model to a representation under another meta-model, following a given set of mapping between the meta-models.

B. META-MODEL MAPPINGS

This chapter presents the complete formal EXPRESS-X mappings of the transformation descriptions in Chapter 6 that are employed in the examples in Chapter 7.

B.1 Data-Flow Diagram to Common Meta-Model

```
1  SCHEMA_MAP dfd_cmm;
2
3  (* Mapping from the data-flow diagram meta-model (dfd) *)
4  (* to the common meta-model (cmm) *)
5
6  REFERENCE FROM dfd_meta_model AS SOURCE;
7  REFERENCE FROM common_meta_model AS TARGET;
8
9
10 (* diagram *)
11
12 MAP diagram
13 AS m: cmm_module;
14 FROM d: dfd_diagram;
15 IDENTIFIED_BY d.name;
16 SELECT
17   m.name := d.name;
18   m.description := d.description + '(Genereated from DFD diagram.)';
19 END_MAP;
20
21
22 (* external entity *)
23
24 MAP external_entity
25 AS
26   c: cmm_classification_definition;
27   ver: cmm_view_element_role;
28 FROM e: dfd_external_entity;
29 IDENTIFIED_BY e.name;
30 SELECT
31   c.name := e.name;
32   c.description := e.description + '(Generated from DFD external entity.)';
33   c.kind := external;
34
35   ver.name := e.name;
36   ver.element := c;
37   ver.role_context := diagram(e.owner.name);
38 END_MAP;
39
40
41 (* data store *)
42
43 MAP data_store
44 AS
45   c: cmm_classification_definition;
46   ver: cmm_view_element_role;
47 FROM ds: dfd_data_store;
48 IDENTIFIED_BY ds.name;
49 SELECT
50   c.name := ds.name;
51   c.description := ds.description + '(Generated from DFD data store.)';
52   c.kind := general;
53
```

```
54    ver.name := ds.name;
55    ver.element := c;
56    ver.role_context := diagram(ds.owner.name);
57  END_MAP;
58
59
60  (* process *)
61
62  MAP process
63  AS
64    c: cmm_classification_definition;
65    ver: cmm_view_element_role;
66  FROM p: dfd_process;
67  IDENTIFIED_BY p.name;
68  SELECT
69    c.name := p.name;
70    c.description := p.description + '(Generated from DFD process.)';
71    c.kind := functionality;
72
73    ver.name := p.name;
74    ver.element := c;
75    ver.role_context := diagram(p.owner.name);
76  END_MAP;
77
78
79  (* control process *)
80
81  MAP control_process
82  AS
83    c: cmm_classification_definition;
84    ver: cmm_view_element_role;
85  FROM p: dfd_control_process;
86  SELECT
87    c.name := p.name;
88    c.description := p.description + '(Generated from DFD control process.)';
89    c.kind := controller;
90
91    ver.name := p.name;
92    ver.element := c;
93    ver.role_context := diagram(p.owner.name);
94  END_MAP;
95
96
97  (* function for converting a diagram element to a classification definition *)
98  (* used by data-flow *)
99
100 FUNCTION actor(dfc: dfd_data_flow_connector_selection)
101 :cmm_classification_definition;
102 LOCAL
103   a: cmm_classification_definition := ?;
104 END_LOCAL;
105   IF 'DFD_META_MODEL.DFD_PROCESS' IN TYPEOF(dfc) THEN
106     a := c@process(dfc.name);
107   ELSE
108     IF 'DFD_META_MODEL.DFD_EXTERNAL_ENTITY' IN TYPEOF(dfc) THEN
109       a := c@external_entity(dfc.name);
```

```
110      ELSE
111        IF 'DFD_META_MODEL.DFD_DATA_STORE' IN TYPEOF(dfc) THEN
112          a := c@data_store(dfc.name);
113        END_IF;
114      END_IF;
115    END_IF;
116    RETURN (a);
117  END_FUNCTION;
118
119
120  (* data-flow *)
121
122  MAP data_flow
123  AS
124    a: cmm_association;
125    sr: cmm_role;
126    tr: cmm_role;
127    ac: cmm_association_classification;
128    c: cmm_classification_definition;
129    p: AGGREGATE OF cmm_property;
130    ver: cmm_view_element_role;
131  FROM df: dfd_data_flow;
132  IDENTIFIED_BY df.name;
133  SELECT
134    sr.name := 'Source role';
135    sr.relationship := a;
136    sr.is_aware := TRUE;
137    sr.cardinality := 1;
138    sr.actor := actor(df.source_connector);
139
140    tr.name := 'Target role';
141    tr.relationship := a;
142    tr.is_aware := FALSE;
143    tr.cardinality := '*';
144    tr.actor := actor(df.target_connector);
145
146    a.name := df.name;
147    a.description := df.description + '(Generated from DFD data-flow.)';
148    a.SOURCE := sr;
149    a.TARGET := tr;
150
151    ver.name := df.name;
152    ver.element := a;
153    ver.role_context := diagram(df.owner.name);
154
155    c.name := sr.actor.name + '-' + tr.actor.name;
156    c.description := '(Association classification, generated from DFD data-flow.)';
157    c.kind := general;
158    ac.association := a;
159    ac.classification := c;
160    ac.description := c.description;
161
162
163    FOR EACH d IN df.data_identifiers INDEXING i;
164      SELECT
165        p[i].name := d;
```

```
166        p[i].owner := c;
167        p[i].cardinality := 1;
168        p[i].data_type := ?;
169        p[i].description := '(Data identifier, generated from data-flow.)';
170        p[i].encapsulation := public;
171        p[i].is_mandatory := TRUE;
172        p[i].is_constant := FALSE;
173        p[i].assigned_value := ?;
174
175  END_MAP;
176
177
178  (* incoming control flow *)
179
180  MAP incoming_control_flow
181  AS
182    t: cmm_control_transition;
183    e: cmm_event;
184    sr: cmm_role;
185    tr: cmm_role;
186  FROM cf: dfd_incoming_control_flow;
187  IDENTIFIED_BY cf.name;
188  SELECT
189    sr.name := 'Source role';
190    sr.relationship := t;
191    sr.is_aware := TRUE;
192    sr.cardinality := 1;
193    sr.actor := actor(cf.source_connector);
194
195    tr.name := 'Target role';
196    tr.relationship := t;
197    tr.is_aware := FALSE;
198    tr.cardinality := '*';
199    tr.actor := c@control_process(cf.target_connector);
200
201    e.name := 'Control event';
202    t.name := cf.name;
203    t.description := cf.description;
204    t.SOURCE := sr;
205    t.TARGET := tr;
206    t.trigger := e;
207    t.condition := '';
208    t.action := ?;
209  END_MAP;
210
211
212  (* outgoing control flow *)
213
214  MAP outgoing_control_flow
215  AS
216    t: cmm_control_transition;
217    e: cmm_event;
218    sr: cmm_role;
219    tr: cmm_role;
220  FROM cf: dfd_outgoing_control_flow;
221  IDENTIFIED_BY cf.name;
```

```
222 SELECT
223   sr.name := 'Source role';
224   sr.relationship := t;
225   sr.is_aware := TRUE;
226   sr.cardinality := 1;
227   sr.actor := c@control_process(cf.source_connector);
228 
229   tr.name := 'Target role';
230   tr.relationship := t;
231   tr.is_aware := FALSE;
232   tr.cardinality := '*';
233   tr.actor := actor(cf.target_connector);
234 
235   e.name := 'Control event';
236   t.name := cf.name;
237   t.description := cf.description;
238   t.SOURCE := sr;
239   t.TARGET := tr;
240   t.trigger := e;
241   t.condition := '';
242   t.action := ?;
243 END_MAP;
244 
245 
246 END_SCHEMA_MAP;
```

B.2 Common Meta-Model to Static Structure Diagram

```
SCHEMA_MAP cmm_ssd;

(* Mapping from the common meta-model (cmm) *)
(* to the static structure diagram meta-model (ssd) *)

REFERENCE FROM common_meta_model AS SOURCE;
REFERENCE FROM oo_meta_model AS TARGET;


(* diagram *)

MAP diagram
AS d: ssd_diagram;
FROM m: cmm_module;
IDENTIFIED_BY m.name;
LOCAL
  e: AGGREGATE OF ssd_element;
END_LOCAL;
SELECT
  d.name := m.name;
  d.description := m.description;
  FOR EACH me IN m.elements INDEXING i;
    SELECT
      e[i].owner := diagram(m.name);
END_MAP;


(* package *)

MAP package
AS p: ssd_package;
FROM m: cmm_module;
IDENTIFIED_BY m.name;
LOCAL
  r: cmm_view_element_role;
END_LOCAL;
WHERE m.owner <> ?;
SELECT
  r :=  QUERY(ver <* extent('COMMON_META_MODEL.CMM_VIEW_ELEMENT_ROLE')
  | ver.element = m)[1];

  p.name := m.name;
  p.description := m.description;
  p.content := diagram(m.name);
  p.owner := diagram(r.role_context.name);
END_MAP;


(* class *)

MAP class
```

```
52  AS c: ssd_class;
53  FROM cd: cmm_classification_definition;
54  IDENTIFIED_BY cd.name;
55  LOCAL
56    r: cmm_view_element_role;
57  END_LOCAL;
58  SELECT
59    r :=  QUERY(ver <* extent('COMMON_META_MODEL.CMM_VIEW_ELEMENT_ROLE')
60    | ('COMMON_META_MODEL.CMM_CLASSIFICATION_DEFINITION' IN TYPEOF(ver))
61    AND (ver.element = cd))[1];
62
63    c.name := cd.name;
64    c.description := cd.description;
65    c.owner := diagram(r.role_context.name);
66  END_MAP;
67
68
69  (* attribute *)
70
71  MAP attribute
72  AS a: oo_attribute;
73  FROM p: cmm_property;
74  SELECT
75    a.name := p.name;
76    a.owner := class(p.owner.name);
77    a.multiplicity := p.cardinality;
78    a.visibility := IF p.encapsulation = cmm_encapsulation_kind.public THEN
79      oo_visibility_kind.public;
80    ELSIF p.encapsulation = cmm_encapsulation_kind.PRIVATE THEN
81      oo_visibility_kind.public;
82    ELSIF p.encapsulation = cmm_encapsulation_kind.protected THEN
83      oo_visibility_kind.protected;
84    END_IF;
85  END_MAP;
86
87
88  (* operation *)
89
90  MAP OPERATION
91  AS o: oo_operation;
92  FROM f: cmm_function;
93  IDENTIFIED_BY f.name;
94  SELECT
95    o.name := f.name;
96    o.owner := class(f.owner.name);
97    o.visibility := IF f.encapsulation = cmm_encapsulation_kind.public THEN
98      oo_visibility_kind.public;
99    ELSIF f.encapsulation = cmm_encapsulation_kind.PRIVATE THEN
100     oo_visibility_kind.public;
101   ELSIF f.encapsulation = cmm_encapsulation_kind.protected THEN
102     oo_visibility_kind.protected;
103   END_IF;
104   o.return_data_type := ?;
105   o.specification := f.specification.specification;
106 END_MAP;
107
```

```
108
109  (* operation parameter *)
110
111  MAP parameter
112  AS op: oo_operation_parameter;
113  FROM fp: cmm_function_parameter;
114  SELECT
115    op.name := fp.name;
116    op.data_type := ?;
117    op.owner := OPERATION(fp.owning_function.name);
118  END_MAP;
119
120
121  (* association *)
122
123  MAP association
124  AS
125    sa: ssd_association;
126  FROM ca: cmm_association;
127  IDENTIFIED_BY ca.name;
128  LOCAL
129    r: cmm_view_element_role;
130    sae: oo_association_end;
131    tae: oo_association_end;
132  END_LOCAL;
133  SELECT
134    r :=  QUERY(ver <* extent('COMMON_META_MODEL.CMM_VIEW_ELEMENT_ROLE')
135    | ('COMMON_META_MODEL.CMM_ASSOCIATION' IN TYPEOF(ver))
136    AND (ver.element = ca))[1];
137
138    sa.name := ca.name;
139    sa.description := ca.description;
140    sa.reading_direction := tae;
141    sa.owner := diagram(r.role_context.name);
142  END_MAP;
143
144
145  (* association end *)
146
147  MAP association_end
148  AS ae: oo_association_end;
149  FROM r: cmm_role;
150  SELECT
151    ae.role_name := r.name;
152    ae.participant := class(r.actor.name);
153    ae.aggregation := none;
154    ae.association := association(r.relationship.name);
155    ae.multiplicity := r.cardinality;
156  END_MAP;
157
158  (* association class *)
159
160  MAP assocation_class
161  AS sac: oo_association_class;
162  FROM cac: cmm_association_classification;
163  SELECT
```

```
164   sac.association := association(cac.association.name);
165   sac.class := class(cac.classification.name);
166 END_MAP;
167
168
169 (* generalization *)
170
171 MAP generalization
172 AS g: oo_generalization;
173 FROM i: cmm_inheritance;
174 SELECT
175   g.child := class(i.SOURCE.actor.name);
176   g.parent := class(i.TARGET.actor.name);
177 END_MAP;
178
179
180 END_SCHEMA_MAP;
```

B.3 State-Transition Diagram to Common Meta-Model

```
SCHEMA_MAP std_cmm;

(* Mapping from the state-transition diagram meta-model (std) *)
(* to the common meta-model (cmm) *)

REFERENCE FROM std_meta_model AS SOURCE;
REFERENCE FROM common_meta_model AS TARGET;


(* diagram *)

MAP diagram
AS m: cmm_control_view;
FROM d: std_diagram;
IDENTIFIED_BY d.name;
SELECT
  m.name := d.name;
  m.description := d.description + '(Generated from STD diagram.)';
END_MAP;


(* function for generating a control view for a sub-state *)
(* used by state mapping *)

FUNCTION subdiagram(o: cmm_state; d: std_diagram): cmm_control_view;
LOCAL
 m: cmm_control_view;
END_LOCAL;
  m := diagram(d.name);
  m.description := d.description + '(Generated for sub-state of state '
  + o.name + '.)';
  m.owner := o;
  RETURN (m);
END_FUNCTION;


(* state *)

MAP state
AS
  cs: cmm_state;
  ver: cmm_view_element_role;
FROM ss: std_state;
SELECT
  cs.name := ss.name;
  cs.description := ss.description;
  cs.kind := controller;
  cs.state_kind := IF ss.kind = std_state_kind.initial THEN
    cmm_state_kind.initial;
  ELSIF ss.kind = std_state_kind.normal THEN
    cmm_state_kind.normal;
  ELSIF ss.kind = std_state_kind.final THEN
```

```
53      cmm_state_kind.final;
54    END_IF;
55    cs.active_elements := ?;
56    cs.concurrent_substates := FOR EACH s IN ss.concurrent_substates
57      RETURN subdiagram(cs, s);
58    cs.description := '';
59
60    ver.name := ss.name;
61    ver.element := cs;
62    ver.role_context := diagram(ss.owner.name);
63  END_MAP;
64
65
66  (* transition *)
67
68  MAP transition
69  AS
70    ct: cmm_control_transition;
71    e: cmm_event;
72    sr: cmm_role;
73    tr: cmm_role;
74    ts: cmm_textual_specification;
75  FROM t: std_transition;
76  IDENTIFIED_BY t.source_state, t.target_state;
77  SELECT
78    sr.name := IF EXISTS(t.event_expression) THEN
79      t.event_expression.specification + ' source';
80    ELSE
81      'Transition source';
82    END_IF;
83    sr.actor := cs@state(t.source_state);
84    sr.description := 'Role of ' + sr.name + '.';
85    sr.cardinality := 1;
86    sr.is_aware := TRUE;
87    sr.relationship := ct;
88
89    tr.name := IF EXISTS(t.event_expression) THEN
90      t.event_expression.specification + ' target';
91    ELSE
92      'Transition target';
93    END_IF;
94    tr.actor := cs@state(t.target_state);
95    tr.description := 'Role of ' + tr.name + '.';
96    tr.cardinality := '*';
97    tr.is_aware := FALSE;
98    tr.relationship := ct;
99
100   e.name := IF EXISTS(t.event_expression) THEN
101     t.event_expression.specification;
102   ELSE
103     '';
104   END_IF;
105
106   ts.language := '(unknown language)';
107   ts.specification := IF EXISTS(t.action) THEN
108     t.action.specification;
```

```
109    ELSE
110      ?;
111    END_IF;
112
113    ct.name := IF EXISTS(t.event_expression) THEN
114      t.event_expression.specification;
115    ELSE
116      '';
117    END_IF;
118    ct.description := t.description;
119    ct.SOURCE := sr;
120    ct.TARGET := tr;
121    ct.trigger := e;
122    ct.condition := IF EXISTS(t.guard_expression) THEN
123      t.guard_expression.specification;
124    ELSE
125      '';
126    END_IF;
127    ct.action := IF t.action = ? THEN
128      ?;
129    ELSE
130      ts;
131    END_IF;
132  END_MAP;
133
134
135  END_SCHEMA_MAP;
```

B.4 Common Meta-Model to Statechart Diagram

```
1  SCHEMA_MAP cmm_scd;
2
3  (* Mapping from the common meta-model (cmm) *)
4  (* to the (object oriented) statechart diagram meta-model (scd) *)
5
6  REFERENCE FROM common_meta_model AS SOURCE;
7  REFERENCE FROM oo_meta_model AS TARGET;
8
9
10 (* diagram *)
11
12 MAP diagram
13 AS d: scd_diagram;
14 FROM cv: cmm_control_view;
15 IDENTIFIED_BY cv.name;
16 SELECT
17   d.name := cv.name;
18   d.description := cv.description;
19   d.initial_state := ?;
20 END_MAP;
21
22
23 (* initial state *)
24
25 MAP initial_state
26 AS is: scd_initial_state;
27 FROM cs: cmm_state;
28 LOCAL
29   r: cmm_view_element_role;
30   d: scd_diagram;
31 END_LOCAL;
32 WHERE cs.state_kind = initial;
33 SELECT
34   r := QUERY(ver <* extent('COMMON_META_MODEL.CMM_VIEW_ELEMENT_ROLE')
35   | ver.element = cs)[1];
36
37   d := diagram(r.role_context.name);
38   is.owner := d;
39   is.description := cs.description;
40
41   d.initial_state := is;
42 END_MAP;
43
44
45 FUNCTION serialize_texts(a: AGGREGATE OF global.text): global.text;
46 LOCAL
47   st: global.text := '';
48 END_LOCAL;
49   REPEAT i := LOINDEX(a) TO HIINDEX(a);
50     st := st + ' ' + a[i];
51   END_REPEAT;
52   RETURN (st);
53 END_FUNCTION;
```

```
54
55  (* state *)
56
57  MAP state
58  AS s: scd_state;
59  FROM cs: cmm_state;
60  IDENTIFIED_BY cs.name;
61  LOCAL
62    r: cmm_view_element_role;
63    d: scd_diagram;
64    dn: AGGREGATE OF global.text;
65    ct: global.text;
66  END_LOCAL;
67  WHERE cs.state_kind = normal;
68  SELECT
69    r := QUERY(ver <* extent('COMMON_META_MODEL.CMM_VIEW_ELEMENT_ROLE')
70    | ver.element = cs)[1];
71
72    s.name := cs.name;
73    dn := FOR EACH subs IN cs.concurrent_substates RETURN (subs.name);
74    ct := IF HIINDEX(cs.concurrent_substates) > 1 THEN
75     'This state has concurrent sub-states specified in diagrams:'
76     + serialize_texts(dn);
77    ELSE
78      '';
79    END_IF;
80    s.description := cs.description + ct;
81    s.owner := diagram(r.role_context.name);
82
83    FOR EACH subs IN cs.concurrent_substates INDEXING i;
84      SELECT
85        d := diagram(subs.name);
86        d.description := subs.description
87        + '(This is a concurrent sub-state of state ' + s.name
88        + ' in diagram ' + s.owner.name + '.)';
89  END_MAP;
90
91
92  (* final state *)
93
94  MAP final_state
95  AS fs: scd_final_state;
96  FROM cs: cmm_state;
97  LOCAL
98    r: cmm_view_element_role;
99  END_LOCAL;
100 WHERE cs.state_kind = final;
101 SELECT
102   r := QUERY(ver <* extent('COMMON_META_MODEL.CMM_VIEW_ELEMENT_ROLE')
103   | ver.element = cs)[1];
104
105   fs.owner := diagram(r.role_context.name);
106   fs.description := cs.description;
107 END_MAP;
108
109
```

```
110 (* transition *)
111
112 MAP transition
113 AS
114   t: scd_transition;
115   a: scd_action;
116   e: expression;
117 FROM ct: cmm_control_transition;
118 LOCAL
119   ss: cmm_state := ct.SOURCE.actor;
120   ts: cmm_state := ct.TARGET.actor;
121 END_LOCAL;
122 SELECT
123   a.name := IF EXISTS(ct.action) THEN
124     'Transition action, specified in: ' + ct.action.language;
125   ELSE
126     'Transition action';
127   END_IF;
128   a.specification := IF EXISTS(ct.action) THEN
129     ct.action.specification;
130   ELSE
131     ?;
132   END_IF;
133
134   e.language := 'boolean';
135   e.specification := ct.condition;
136
137   t.EVENT := IF EXISTS(ct.trigger) THEN
138     ct.trigger.name;
139   ELSE
140     ct.name;
141   END_IF;
142   t.guard := e;
143   t.action := a;
144
145   t.description := ct.description;
146   t.source_state := IF ss.state_kind = initial THEN
147     initial_state(ss);
148   ELSIF ss.state_kind = normal THEN
149     state(ss.name);
150   ELSIF ss.state_kind = final THEN
151     final_state(ss);
152   END_IF;
153   t.owner := t.source_state.owner;
154   t.target_state := IF ts.state_kind = initial THEN
155     initial_state(ts);
156   ELSIF ts.state_kind = normal THEN
157     state(ts.name);
158   ELSIF ts.state_kind = final THEN
159     final_state(ts);
160   END_IF;
161 END_MAP;
162
163
164 END_SCHEMA_MAP;
```

BIBLIOGRAPHY

[Ala88] B. Alabiso. Transformation of data flow analysis models to object-oriented design. In *OOPSLA 1988 Conference Proceedings, Special Issue of SIGPLAN Notices*, volume 23 (11), pages 335 – 353, November 1988.

[AP02] Dario Aganovic and Asmus Pandikow. Towards enabling innovation processes for dynamic extended manufacturing enterprises. In *Proceedings of the 2002 Digital Enterprise Technology Conference*, 2002.

[Axe00] Jakob Axelsson. Unified modeling of real-time control systems and their physical environments using uml. In *Proceedings of the Eigth IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2000.

[Bal96] H. Balzert. *Methoden der objektorientierten Systemanalyse.* Spektrum Akademischer Verlag, Heidelberg, 1996. (in German).

[BBM90] M. Bewtra, S. C. Balin, and J. M. Moore. An ada design and implementation toolset based on object-oriented and functional programming paradigms. In *Proceedings of the Seventh Washington Ada Symposium*, pages 213 – 226, June 1990.

[BD89] R. J. Brown and V. Dobbs. A method for translating functional requirements for object-oriented design. In *Proceedings of the Seventh Annual National Conference on Ada Technology*, pages 589–599, March 1989.

[BDR84] D. Boehm-Davis and L. S. Ross. Approaches to structuring the software development process. Technical Report GEC/DIS/TR-84-BI V-I, General Electric Company, October 1984.

[BJ66] C. Böhm and G. Jacopini. Flow diagrams, turing machines and

languages with only two formation rules. *Communications of the ACM*, 9(5):366 – 371, May 1966.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21:61 – 72, May 1988.

[Boo91] Grady Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings, Redwood City, 1991.

[Boo94a] G. Booch. The evolution of the booch method. *Report on Object Analysis and Design*, pages 2 – 5, May 1994.

[Boo94b] G. Booch. *Object-Oriented Analysis and Design With Applications, Second edition.* Benjamin/Cummings, Redwood City, 1994.

[BR95] G. Booch and J. Rumbaugh. Unified method, version 0.8. Rational Software Corporation, Santa Clara, Internet at http://www.rational.com, 1995.

[BRJ96] G. Booch, J. Rumbaugh, and I. Jacobson. The unified modeling language for object-oriented development, version 0.91. Rational Software Corporation, Santa Clara, Internet at http://www.rational.com, 1996.

[Che76] P. Chen. The entity-relationship model - towards a unified view of data. In *ACM Transactions on Database Systems*, volume 1 (1), March 1976.

[CL98] Jordi Cortadella and Luciano Lavagno. Deriving petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8), August 1998.

[CS95] M. A. Cusamano and R. W. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People.* The Free Press / Simon and Schuster, New York, 1995.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design.* Yourdon Press, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[DeM78] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New York, 1978.

[Dij69] E. W. Dijkstra. Structured programming. In J.N. Buxton and B. Randell, editors, *Software Engineering Techniques.* NATO Science Committee, Brussels, Belgium, 1969.

[Dor02] Dov Dori. *Object-Process Methodology.* Springer-Verlag, 2002.

[DSI02] OMG SE DSIG. Homepage of the se dsig, systems engineering domain special interest group, at the omg, object management group. Internet at http://syseng.omg.org, 2002.

[FK92] R. G. Fichman and C. F. Kemerer. Object-oriented and conventional analysis and design methodologies: Comparison and critique. *IEEE Computer*, 25(10):22 – 39, October 1992.

[Gil88] T. Gilb. *Principles of Software Engineering Management.* Addison-Wesley, Wokingham, UK, 1988.

[GJK93] L. R. Garceau, E. G. Jancura, and J. Kneiss. Object-oriented analysis and design: A new approach to systems development. *Journal of Systems Management*, 44(1):25 – 32, 1993.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Massachusetts, 1983.

[Gra87] L. Gray. Procedures for transitioning from structured methods to object-oriented design. *Proceedings of the Conference on Methodologies and Tools for Real-Time Systems IV*, pages R–1 – R–21, September 1987. National Institute for Software Quality and Productivity, Washington, D.C.

[Gra88] L. Gray. Transitioning from structured analysis to object-oriented design. *Proceedings of the Fifth Washington Ada Symposium*, pages 151 – 162, 1988. Association for Computing Machinery, New York.

[Gra91] I. Graham. *Object Oriented Methods.* Addison-Wesley, 1991.

[Gra94] R. B. Grady. Successfully applying software metrics. *IEEE Computer*, 27:18 – 25, September 1994.

[Gro99] Object Management Group. Omg unified modeling language specification, version 1.3. Internet at http://www.rational.com/uml, 1999.

[Gro00] Object Management Group. Omg meta-object facility. Internet at http://www.omg.org/cgi-bin/doc?formal/00-04-03, April 2000.

[Gro01] Object Management Group. Omg unified modeling language specification, version 1.4. Internet at http://www.omg.org/technology/documents/formal/uml.htm, 2001.

[Gro02] Object Management Group. Omg website. Internet at http://www.omg.org, 2002.

[GS79] C. Gane and T. Sarson. *Structured Systems Analysis*. Prentice Hall, 1979.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, Chapter 10:231 – 274, June 1987.

[Hei95] Sandra Heiler. Semantic interoperability. *ACM Computing Surveys*, 27:271 – 273, 1995.

[Hol01] Jon Holt. *UML for systems engineering*. IEE, The Institution of Electrical Engineers, 2001.

[HS91] B. Henderson-Sellers. Hybrid object-oriented/functional decomposition methodologies for the software engineering life-cycle. *Hotline on Object-Oriented Technology*, 2(7):1 – 8, May 1991.

[HSC91] B. Henderson-Sellers and L. L. Constantine. Object-oriented development and functional decomposition. *Journal of Object-Oriented Programming*, 3(5):11 – 17, January 1991.

[HSE90] B. Henderson-Sellers and J. M. Edwards. The object-oriented life cycle. *Communications of the ACM*, 33:142 – 159, September 1990.

[ISO94] International Standardization Organization ISO. Step, iso 10303-1, standard for the exchange of product model data. Internet at http://www.iso.ch/, 1994.

[ISO01] International Standardization Organization ISO. Publicly available specification 20542. Internet at http://www.iso.ch/, 2001.

[ISO02] International Standardization Organization ISO. Website of

the international standarization organization. Internet at http://www.iso.org, 2002.

[Jac75] M. A. Jackson. *Principles of Program Design.* Academic Press, New York, 1975.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. "Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[JH01] Julian Johnson and Erik Herzog. The data standard ap-233: An invigorator for global systems engineering. In *Proceedings of the 11th Annual International Symposium of the International Council on Systems Engineering*, 2001.

[Kam87] G. R. Kampen. An eclectic approach to specification. In *Proceedings of the Fourth International Workshop on Software Specification and Design, Monterey*, pages 178 – 182, April 1987.

[Kel87] J.C. Kelly. A comparison of four design methods for real-time systems. In *Proceedings of the 9th International Conference on Software Engineering*, pages 238 – 252, March 1987.

[Kha89] G. K. Khalsa. Using object modeling to transform structured analysis into object-oriented design. In *Proceedings of the Sixth Washington Ada Symposium*, pages 201– 212, June 1989.

[Lan85] K. E. Lantz. *The Prototyping Methodology.* Prentice Hall, Englewood Cliffs, New York, 1985.

[Li91] X. Li. Integration of structured and object-oriented programming. In *Focus On Analysis and Design*, pages 54 – 60. SIGS Publications Inc., New York, 1991.

[Luk91] J. T. Lukman. Transforming the 2167a requirements definition model into an ada-object-oriented design. In *Proceedings of the Ninth Annual National Conference on Ada Technology*, pages 200 – 205, March 1991.

[Mar88] J. Martin. *Information Engineering, Book I: Introduction.* Prentice Hall, Englewood Cliffs, New York, 1988.

[Mey87] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4, chapters 6 and 7:50 – 64, 1987.

[NTS99] Simin Nadjm-Tehrani and Jan-Erik Strömberg. Proving dynamic properties in an aerospace application. *Formal Methods in System Design*, 2:135 – 169, March 1999.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058, December 1972.

[Pen89] J. Pendley. Using information engineering and ada object-oriented design methods in concert - a case study. In *Proceedings of the Sixth Washington Ada Symposium*, pages 11 – 19, June 1989.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Germany, 1962. (in German).

[PHT00] Asmus Pandikow, Erik Herzog, and Anders Törne. Integrating systems and software engineering concepts in ap-233. In *Proceedings of the 10th Annual International Symposium of the International Council on Systems Engineering*. INCOSE, 2000.

[Pre00] R. S. Pressman. *Software Engineering - A Practitioner's Approach - European Adaptation by Darrell Ince, Fifth Edition*. McGraw Hill, 2000.

[PT01a] Asmus Pandikow and Anders Törne. Integrating modern software engineering and systems engineering specification techniques. In *Proceedings of the 2001 International Conference on Software and Systems Engineering and its Applications*, 2001.

[PT01b] Asmus Pandikow and Anders Törne. Software engineering at system level. In *Proceedings of the First Swedish National Conference on Software Engineering Research and Practice*, 2001.

[PT01c] Asmus Pandikow and Anders Törne. Support for object-orientation in ap-233. In *Proceedings of the 11th Annual International Symposium of the International Council on Systems Engineering*. INCOSE, 2001.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Ros77] D. Ross. Structured analysis (sa): A language for communicat-

ing ideas. *IEEE Transactions on Software Engineering*, 3(1), January 1977.

[Roy89] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 11th International Conference on Software Engineering*, pages 328 – 338, May 1989. (reprint from original publication in the Technical Papers of the Western Electronic Show and Convention, Los Angeles, August 1970).

[SC93] R.C. Sharble and S. S. Cohen. The object-oriented brewery: A comparison of two object-oriented development methods. *ACM SIGSOFT Software Engineering Notes*, 18(2):60–73, 1993.

[Sch99] Steven R. Schach. Classical and object-oriented software engineering with uml and java, forth edition. In *McGraw Hill*, 1999.

[SED99] SEDRES. Sedres website. Internet at http://www.ida.liu.se/projects/sedres/, 1999.

[SED02] SEDRES. Sedres website. Internet at http://www.sedres.com, 2002.

[Sha94] Mary Shaw. Comparing architectural design styles. *IEEE Software*, pages 27 – 41, November 1994.

[Shl92] S. Shlaer. *A Comparison of OOA and OMT*. Project Technology Inc., 1992.

[SM88] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis - Modeling the World in Data*. Yourdon Press, Prentice Hall, 1988.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115 – 139, May 1974.

[SW94] Douglas Schenck and Peter Wilson. *Information Modeling: The EXPRESS Way*. Oxford University Press, 1994.

[VJBCS97] Pepjijn R. S. Visser, Dean M. Jones, T. J. M. Bench-Capon, and M. J. R. Shave. An analysis of ontological mismatches: Heterogeneity versus interoperability. In *AAAI 1997 Spring Symposium on Ontological Engineering*, Stanford, USA, 1997.

[War74] J.-D. Warnier. *Logical Construction of Programs.* Van Nostrand Reinhold Company, New York, 1974.

[War89] P. T. Ward. How to integrate object orientation with structured analysis and design. *IEEE Software*, pages 74 – 82, March 1989.

[WBW89] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: A responsibility-driven approach. *OOPSLA 1989 Conference Proceedings, Special Issue of SIGPLAN Notices*, 24(10):71 – 76, October 1989.

[Wir71] N. E. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221 – 227, April 1971.

[WWW02] World Wide Web Consortium WWWC. Semantic web. Internet, http://www.w3.org/2001/sw/, 2002.

[YC79] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice Hall, Englewood Cliffs, New York, 1979.

[You75] E. Yourdon. *Techniques of Program Structure and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[You89] E. Yourdon. Object-Oriented Observations. *American Programmer*, 2(7-8):3 – 7, 1989.

[You96] E. Yourdon. *Rise and Resurrection of the American Programmer.* Yourdon Press, New York, 1996.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented

Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering
- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.