

VRIJE UNIVERSITEIT

From Smallest Software Particle to System Specification

MuDForM: Multi-Domain Formalization Method

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van Doctor of Philosophy aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. J.J.G. Geurts,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op dinsdag 12 november 2024 om 11.45 uur
in een bijeenkomst van de universiteit,
De Boelelaan 1105

door

Robertus Theodorus Christianus Deckers

geboren te 's-Hertogenbosch

promotoren: prof.dr. P. Lago
 prof.dr. W.J. Fokkink

promotiecommissie: prof.dr. M. van den Brand
 prof.dr. P. Avgeriou
 dr. R. Eramo
 dr. E. Woods
 dr. I. Malavolta



SIKS Dissertation Series No. 2024-34

The research reported in this thesis has been carried out under the auspices of SIKS,
the Dutch Research School for Information and Knowledge Systems.

Promotiecommissie:

prof.dr. Mark van den Brand, (Eindhoven University of Technology, The Netherlands)

prof.dr. Paris Avgeriou, (University of Groningen, The Netherlands)

dr. Romina Eramo (University of Teramo, Italy)

dr. Eoin Woods (Endava, United Kingdom)

dr. Ivano Malavolta (Vrije Universiteit Amsterdam, The Netherlands)

ISBN/EAN: 978-94-6510-164-4

DOI: <https://doi.org/10.5463/thesis.831>

Copyright © 2024, Robert Deckers

All rights reserved unless otherwise stated

Cover design by Nieck van Zoggel and Robert Deckers

Printed by Proefschriftmaken.nl

Declaration

I, Robert Deckers, hereby declare that the thesis titled "*From Smallest Software Particle to System Specification*" and its content are the result of my own work.

I confirm that:

- This work was primarily conducted during my pursuit of a research degree at this university.
- If any portion of this thesis has been previously submitted for an academic degree or any other qualification at this university or any other educational institution, I have explicitly disclosed this information.
- I have consistently acknowledged the sources of published works by other authors that I consulted.
- In cases where I have included excerpts from the work of others, I have consistently provided proper source attributions. Aside from these quotations, the entirety of this thesis represents my independent effort.
- I have acknowledged all significant sources of assistance.
- In cases where this thesis draws upon collaborative efforts with others, I have provided a clear distinction between the contributions made by collaborators and my own individual contributions.

September 1, 2024

I dedicate this thesis to my Mother, who unconditionally supported this academic journey, and my late Father, who would have been proud.

Acknowledgements

I am deeply grateful to my parents for their belief in me and the warmth with which they supported me. They have been a motivating force in my academic pursuit.

I am very grateful to my dedicated supervisor, Patricia Lago, for her invaluable guidance and unwavering support throughout the past nine years, and for the way she handled my directness and frustrated reactions to the author's nightmare called \LaTeX . I always found it impressive how she could pin the errors in my work, suggest improvements, and set priorities.

Special thanks go out to Bonne van Dijk, Jennek Geels, Hans van Hemmen, Arjan van Krimpen, and Eric Suijs for their reviews and many discussions, not only about the content of the work, but also about my struggles with the quirks of the academic publication world. I also thank Wan Fokkink, Jan Schoonderbeek, Aksel de Vries, Frank Benders, Marc Hamilton, Ivano Malavolta, Dennis van den Brand, Arash Saberi, and Daan Pasmans for their reviews and discussions.

My love goes out to all my friends, to whom I have spitted my heart out, when I was frustrated with some BS during this bumpy ride. I hope they will be there on my next journey.

Abstract

Software development can be seen as a process in which the knowledge and decisions of a wide variety of people are integrated to produce a machine-readable specification. The majority of today's specification and programming languages are based on computer-oriented and mathematical concepts. Those languages are difficult for people that are not a software professional.

Domain models and domain-specific languages are a way to capture knowledge from people who have no software expertise, while also resulting in specifications that can be used as software. This thesis reports on the creation of a domain-oriented method to systematically transform knowledge and decisions of all people involved in a development process, into unambiguous specifications.

A systematic literature review revealed that existing domain-oriented techniques have a low engineering maturity, are barely connected to human communication and thinking, and lack methodical support for applying domain models and domain-specific languages to create system specifications.

This thesis describes how these three problems are addressed by a method called MuDForM. 25 years of modeling and specification experience was captured in a method definition and gradually improved, driven by the outcome of case studies and shortcomings of existing methods. A reflection on the method from the perspective of cognitive science and linguistics concludes this thesis, demonstrating the feasibility of cognition-based specifications.

The result is a method with a uniform metamodel with precise modeling concepts, a process that guides the path from knowledge elicitation to engineered model, and a set of principles that enable implementation and improvement of the method. The method can be seen as a step in transforming a computer-centered and mathematics-oriented discipline, into one that facilitates knowledge and decision specification for all people involved in software development.

Samenvatting

Softwareontwikkeling is een proces waarin kennis en beslissingen van verschillende mensen worden geïntegreerd om een specificatie te produceren die door een machine gelezen kan worden. De meeste hedendaagse specificatie- en programmeertalen zijn gebaseerd op computergeoriënteerde en wiskundige concepten. Deze talen zijn moeilijk voor mensen die geen softwareprofessional zijn.

Domeinmodellen en domeinspecifieke talen zijn een middel om kennis vast te leggen van mensen zonder software-expertise, en resulteren in specificaties die als software kunnen worden gebruikt. Dit proefschrift rapporteert over de creatie van een methode waarmee kennis en beslissingen van alle betrokkenen in een ontwikkelingsproces omgezet kunnen worden in eenduidige specificaties.

Een systematische literatuurstudie toonde aan dat bestaande domeingeoriënteerde methodes technisch vrij onvolwassen zijn, nauwelijks verbonden zijn met menselijke communicatie en denken, en geen methodische ondersteuning bieden voor het toepassen van domeinmodellen en domeinspecifieke talen om systeemspecificaties te creëren.

Dit proefschrift beschrijft hoe deze drie problemen worden aangepakt in een methode genaamd MuDForM. 25 jaar modelleer- en specificatie-ervaring zijn vastgelegd in een methodedefinitie, waarna deze geleidelijk verbeterd is, gedreven door resultaten uit case studies en tekortkomingen in bestaande methoden. Een reflectie op de methode vanuit het perspectief van de cognitieve wetenschap en linguïstiek concludeert dit proefschrift, waarbij de haalbaarheid van op cognitie gebaseerde specificaties wordt beschouwd.

Het resultaat is een methode met een uniform metamodel met precieze modelleerconcepten, een proces dat het pad van kenniselicitatie tot een geëngineerd model begeleidt, en een reeks principes die de implementatie en verbetering van de methode mogelijk maken. De methode is een stap in het transformeren van een computergerichte en wiskundige discipline naar een die kennis- en beslissingsspecificatie mogelijk maakt voor alle betrokkenen bij softwareontwikkeling.

Prologue

And you may ask yourself,
“Well, how did I get here?”

Once in a Lifetime,
Talking Heads, 1980

Why is there a difference between specifying and programming?

It is 1990, and I am working on my Master thesis Computing Science. My assignment is to compare two formal specification languages: LOTOS and ExSpecT. Both differ from the imperative and functional programming languages that I am taught during the past four years. The two languages are independent of software aspects like memory size, memory management, pointers, data types, and other execution platform characteristics. As such, both languages are not built on top of computer primitives, but are based on mathematical theories (respectively process algebra and Petri Nets). I guess that is why my teachers call them specification languages instead of programming languages. They lead to unambiguous specifications, but just like any programming language, they are not suited for most people as a means to express their requirements, design decisions, or generic domain knowledge. They have a steep learning curve and there is not much hands-on methodical support. How can I make unambiguous specifications that people can understand?

After that, during my post-master education in Software Technology, I am taught object-oriented modeling. Object Orientation (OO) is closer to human reasoning than imperative programming, functional programming, Petri Nets, or process algebra, because it uses categories as the main language element. In fact, it is based on set theory, which was taught to me at the age of twelve via Venn-diagrams. But also, OO does not offer language primitives that allow people to express knowledge and decisions in the way that they do in daily life. Furthermore, OO makes behavior

subordinate to state, which differs significantly from natural language, where verbs are at the center of sentences. My conclusion is that programming languages, formal specification languages, and modeling languages are not suited for *non-software* people. They are mainly suited for telling computers what to do.

Combine state, behavior, and natural language

Equipped with all these formalisms and languages, I start my first job at a company that exploits its own development method: the KISS method. Its language combines Jackson Structured Design (a derivative of process algebra) with object-oriented design. It uses natural language for the interface between specifications and stakeholders. The result is that specifications can be used to communicate with business managers, domain experts, and users; anyone who speaks natural language. Furthermore, the built models are fed to a code generator or a model interpreter, which results in minimal programming of system-specific code. Today, this would be called a low-code environment.

Application specific source code is a waste

Before the burst of the internet bubble in 2000, my colleagues and I have developed a model interpreter consisting of components that covered software design aspects like user interaction, business logic, persistence, workflow control, reporting, and timing control. A model of an application domain forms the input for creating a software system that is based on the interpreter. The architecture of the interpreter is such that components for design aspects can be added gradually, and be replaced to meet domain-specific non-functional requirements when needed. The functionality is derived from the model, and the non-functional properties are programmed. The latter is in practice not a big issue, because non-functional properties are often more stable than functionality.

The tyranny of functionality

I start my next job at the esteemed software architectures group of Philips Research in December 1999. I discover that most software development activities are still dominated by machine-oriented languages like C, C++, Java, and C#. For me, it is clearly a step back in the way software is developed. I decide to shift my focus to architecture, and observe that development projects mostly aim to realize software functionality, and seldom to create reusable knowledge that is independent of source code. The business opportunity to create reusable solutions for non-functional properties is hardly utilized.

Functionality is not the main issue

It becomes clear to me that an application domain is not only characterized by domain specific concepts and terminology, but also by its own demands for certain system qualities. Software architecture must focus on the quality attributes of the system. Functionality is just one of those qualities, and often not the most important one. I learn to analyze and specify quality requirements, and apply design patterns to implement them. The problem is that it is still manual, and thus, error-prone work, and that the development process suffers from a lot of implicit rework. I think, and still do, that the root cause of this problem is that code structure often reflects the structure of the application domain and system functionality. This structure hinders the realization of other qualities. The result is that in each software project, the design decisions to achieve quality have to be woven into the functional decomposition of the system. This is a huge waste of development effort, and it requires expensive knowledge building in many developers. In hindsight, I understand perfectly why a project that developed the software of a photo copier, ended up in a software stack where 85% of the code was for error handling, and not for copying functionality. The project should have developed an error handler and then added the copying functionality, instead of the other way around. In almost all the organizations that I have been working for, the systems have a functional decomposition as the foundation of their architecture. This is wrong. They get away with it, because the competition is not doing any better.

Reuse knowledge, instead of source code

My conclusion is that systems must be decomposed along the most important design aspects. These aspects can be specified separately, and treated as distinct knowledge domains. A system specification is then the integration of a set of specifications for different aspects. These specifications do not find their roots in computer software, but in the minds of people. Therefore, it is logical that languages and methods for software development should be close to how people think and communicate. I present this idea in my book on software architecture in 2009, when I am working at Sogeti. It starts to itch.

This thesis is the beginning

It is 2013, and I definitely want to bring the idea to life, but I cannot find a company that wants to hire me for it. So, I quit my job and become self-employed. I divide my time between paid architecture consultancy and teaching, and the engineering of method and tools for the idea. In 2014, I give a presentation at the National

Architecture Conference (LAC). Professor Patricia Lago is in the audience, and likes my story. She asks me if I want to start a PhD research project to investigate how the idea can be realized. I explain it requires the creation of a specification method, a supporting modeling tool set, a runtime engine, and some demonstrators. I can not oversee if this is achievable in the scope of such a research project, but it will give me a carrier to work on the idea, and, more important, someone that guides me and keeps me on track. So, I say yes. The PhD journey starts in April 2015.

The smallest software particle

So, then I find myself with the megalomaniac task to come up with an approach in which all the decisions and knowledge of people in a development activity can be expressed and integrated. In other words, an all-encompassing method for software development artifacts. A theory that is rooted in the human mind, and not based on primitives that are derived from computing theory. I call it the search for the smallest software particle.

Revolution through evolution

I find people that collaborate with me on the method, the modeling tool set, the runtime engine, and the demonstrators. It is all incredibly inspiring, but too much. After three years, I pick my battle, and decide that the method is the first thing to work on. This thesis is the culmination of that work.

Apparently, this thesis is just the beginning

Robert Deckers
September 1, 2024.

Contents

1	Introduction	1
1.1	Background	2
1.1.1	Vision: Software Development is a Human Task	2
1.1.2	Research Goal	5
1.1.3	Method Objectives	5
1.2	Research Questions	8
1.3	Research Methodology	9
1.4	Thesis at a Glance	10
1.5	Research Contribution and Authorship	12
2	State-of-the-Art Specification Techniques	15
2.1	Introduction	17
2.2	Background: MuDForM and Domain Modeling	20
2.2.1	What is a Domain Model?	20
2.2.2	The MuDForM Vision	21
2.2.3	MuDForM Objectives	22
2.3	Study Design and Execution	22
2.3.1	Research Questions	23
2.3.2	Search Queries	27
2.3.3	Selection Criteria	29
2.3.4	Study Execution	30
2.3.5	Classification Framework	31
2.4	Study Results	36
2.4.1	Publication Trends	37
2.4.2	Application Scope	39
2.4.3	Method Engineering	40
2.4.4	MuDForM Specific	43

Contents

2.5	Discussion	44
2.5.1	RQ1.1: No Fully Engineered Method	45
2.5.2	RQ1.2: No Methodical Support for Applying a Created DSL or DM	45
2.5.3	RQ1.3: No Integral Support for Multiple Domains	47
2.5.4	Behavior is Mostly Ignored and Poorly Integrated with Structure	47
2.5.5	Minimal Interface with Natural Language	48
2.5.6	The Terminology around Domain Models is not Unified	48
2.6	Related Work	49
2.7	Threats to Validity	51
2.8	Conclusions and Future Work	52
3	Cognition-based Method Engineering of MuDForM	55
3.1	Introduction	58
3.1.1	Problem Statement	59
3.1.2	Contribution and Target Audience	60
3.2	Research Methodology	60
3.3	Method Definition Concepts	62
3.4	Cognitive Aspects	63
3.5	Method Definition	68
3.5.1	Construction Patterns	68
3.5.2	Main Model Structure: Domain, Feature, Context	69
3.5.3	Metamodel	71
3.5.4	Method Flow	74
3.5.5	Guidance	75
3.5.6	Viewpoints	77
3.6	Examples from a Case Study	78
3.6.1	Specification Spaces and Dependencies of Printing Jobs	79
3.6.2	The Interaction View of the Domain Job printing	80
3.6.3	Object Lifecycle of Domain Class Job	81
3.7	Related Work	83
3.7.1	Domain-oriented Specification Methods	83
3.7.2	Generic Method Engineering Techniques	84
3.7.3	Method Engineering from a Cognition-based Perspective	85
3.8	Discussion	86
3.8.1	Method Ingredients and their Application	86
3.8.2	Relation between the Cognitive aspects and Method Definition	87
3.9	Conclusion and Future Work	88
3.A	Detailed Explanation of Construction Patterns	89
3.A.1	Named Elements	89

Contents

3.A.2 Embedding Formalisms	90
3.A.3 Structures	90
3.B All Major Modeling Concepts	93
3.C All Method Steps	98
3.D All Viewpoints	105
4 From Text to Model with MuDForM	109
4.1 Introduction	111
4.2 Background: MuDForM Development	113
4.3 Research Methodology	114
4.4 MuDForM Overview	116
4.4.1 MuDForM Modeling Process	116
4.4.2 MuDForM Model Structure	118
4.4.3 MuDForM Model Elements	118
4.5 Grammatical Analysis	119
4.6 Text-to-Model Transformation	120
4.7 A Case Study: System Behavior Description of the History Feature	122
4.7.1 Case Study Overview and Execution	122
4.7.2 Grammatical Analysis of the History SBD	123
4.7.3 Text-to-Model Transformation for the History Domain Model	125
4.8 Discussion	125
4.9 Related Work	128
4.10 Conclusion and Future Work	130
5 Domain-based Feature Modeling with MuDForM	131
5.1 Introduction	134
5.1.1 Problem Statement	135
5.1.2 Contribution and Audience	136
5.2 MuDForM Vision and Terminology	137
5.2.1 MuDForM Objectives	138
5.2.2 Domain and Feature	138
5.3 Research Methodology	139
5.4 MuDForM Foundation	142
5.4.1 MuDForM Modeling Process	143
5.4.2 MuDForM Model Structure	146
5.4.3 MuDForM Specification Elements	147
5.5 Feature Modeling	151
5.5.1 From Modeling Initiation to Initial Model	151
5.5.2 Engineer Feature	154

Contents

5.6	A Case Study: Modeling the Processes of ISO26262	157
5.6.1	Introduction to the Case	158
5.6.2	Case Study Overview and Execution	159
5.6.3	From Modeling Initiation to Initial Model	160
5.6.4	Specify Function Lifecycles	167
5.6.5	Specify Function Signatures	173
5.6.6	Specify Feature Structure	173
5.7	Discussion	174
5.7.1	Support for Domain-based Specifications	175
5.7.2	Bridging the Gap between Feature Trees and Domain Models . .	178
5.7.3	Feature Modeling is Part of MuDForM	180
5.7.4	Reflection on Using UML for MuDForM	182
5.7.5	Reflection on Case-specific Objectives	183
5.8	Threats to Validity	184
5.9	Related Work	186
5.10	Conclusion and Future Work	187
5.A	Guidelines Pertaining to Feature Modeling	189
6	Cognition Perspective on MuDForM	197
6.1	Introduction	199
6.1.1	Problem Statement and Contribution	201
6.2	Method Creation Context	201
6.3	Research Method	202
6.4	Reflection on the Research Background	204
6.4.1	Use Domain-oriented Models to Convey Knowledge	204
6.4.2	Meaningful Cognition-based Models	205
6.5	Reflection on the Method Definition	207
6.5.1	Separation of Syntax and Semantics in Mind and Method . . .	207
6.5.2	Supporting Analysis, Design, and their Context	208
6.5.3	Model Engineering through Method Engineering	210
6.5.4	Match Reality with Useful Categories	213
6.5.5	Time is Perceived through Events	215
6.5.6	Building Modeling Concepts from Natural Language	216
6.5.7	More Cognitive Aspects and Derived Modeling Concepts . . .	218
6.6	Discussion	220
6.6.1	Cognition-based Method Definition	220
6.6.2	Different Types of Reflections	222
6.7	Related work	222
6.8	Limitations	223

Contents

6.9	Conclusion and Future work	224
7	Conclusion and Future Work	227
7.1	Introduction	229
7.2	Research Questions Answered	229
7.2.1	RQ1: State-of-the-Art Specification Techniques	230
7.2.2	RQ2: Cognition-based Method Engineering	230
7.2.3	RQ3: From Text to Model	231
7.2.4	RQ4: Domain-based Feature Modeling	232
7.2.5	RQ5: Cognition Perspective on MuDForM	232
7.3	Classification of MuDForM	233
7.3.1	Application Scope	233
7.3.2	Method Engineering	235
7.3.3	MuDForM Specific	239
7.4	Future Work	241
7.4.1	Method Ingredients	241
7.4.2	Method Engineering	242
7.4.3	Putting MuDForM in Practice	243
7.5	Main Research Goal	244
A	Cognition-based Reflection Snippets	245
A.1	Research Background: Vision, Research Goals, and Method Objectives	248
A.1.1	Use unambiguous models to convey knowledge	248
A.1.2	People care about their domain. Computers do not know what they are doing	248
A.1.3	Specifications must mean something to people	249
A.1.4	Method steps and viewpoints focus the modeling process	250
A.1.5	Modeling is not an extra activity	251
A.1.6	Focus on domain-oriented methods	251
A.1.7	Exploit the lexical hypothesis	252
A.1.8	Language awareness is innate	252
A.1.9	Cognition is more than language	252
A.1.10	Modeling concepts should be based on Mentalese	253
A.1.11	Focus on mental concepts; not on the brain or reality	254
A.1.12	On the evolution of thought	255
A.2	Separation of Syntax and Semantics In Mind and Method	257
A.2.1	Concepts have meaning	257
A.2.2	There are no colorless green ideas that sleep furiously	257
A.2.3	Meaningless symbols refer to meaningful concepts	258

Contents

A.2.4	The mind has representations of concepts too	258
A.2.5	A concept can have multiple representations	259
A.2.6	Eliminate homonyms and synonyms, but not always	259
A.2.7	Involved experts decide about model element names, not the modeler	261
A.2.8	Resembling word forms in models	261
A.3	Supporting Analysis, Design, and their Context	262
A.3.1	Support analysis and design for multiple domains	262
A.3.2	Analysis and design reflect lexical defining and real defining . .	263
A.3.3	Domain models give meaning to other specifications	264
A.3.4	Context models form a foundation for contracts with other parties	265
A.3.5	Context models support subjectivity	265
A.3.6	Separate what must happen from what makes it happen	266
A.3.7	Formal propositions vs. propositions about something	267
A.4	Model Engineering through Method Engineering	267
A.4.1	Support different types of defining a term	267
A.4.2	Support extensional and intensional defining	269
A.4.3	Support principles of vocabulary learning	270
A.4.4	Avoid different ways of modeling the same meaning	271
A.4.5	There should be guidelines for determining the right granularity of model elements	272
A.4.6	Guidelines for deciding which concepts become a model element	273
A.4.7	Guidelines for distinguishing model elements	273
A.4.8	Models represent situation-independent knowledge	274
A.4.9	Support knowledge elicitation from both semantic memory and episodic memory	275
A.4.10	Traceability helps to prevent falsities	276
A.4.11	Traceability and multiple viewpoints help to detect biases . . .	276
A.5	Match Reality with Useful Categories	277
A.5.1	Categories enable inference of properties	277
A.5.2	Prevent mismatches between reality and model	278
A.5.3	Classifiers must represent functional categories	279
A.5.4	Support multiple functional categories for one instance	279
A.5.5	A model is a complete specification of the world it describes .	280
A.5.6	It must be unambiguously determinable if an object belongs to a category	281
A.5.7	Different objects in the real world can correspond with one model instance	282
A.5.8	Objects differ if and only if they are distinguishable	282

Contents

A.5.9 Objects are not tied to one category	283
A.5.10 There are no exact category definitions in the real world	285
A.6 Time is Perceived through Events	286
A.6.1 Events are ordered in time, and have a duration	286
A.6.2 The past is determined by the events that have happened	286
A.6.3 Lifecycles enable stepping through time	287
A.7 Building Modeling Concepts from Natural Language	288
A.7.1 Language is common, but there is no common language	288
A.7.2 Support for words that are common in all languages	288
A.7.3 Support part of speech concepts	289
A.7.4 Support modeling verbs, and their temporal contour	290
A.7.5 Support to be, to have, to do, and to go	291
A.7.6 Support different meanings of to be	292
A.7.7 Support specification of location, force, agency, and causation .	293
A.7.8 Support different types of speech acts	294
A.7.9 Support Subject-Verb-Object sentences	295
A.7.10 Compound names correlate with hierarchy between concepts .	295
A.7.11 Attributes and steps support indexicals	296
A.7.12 Support word definitions and word usage	297
A.8 More Cognitive Aspects and their Derived Modeling Concepts	297
A.8.1 Support possible connections between ideas	297
A.8.2 Support the specification of hierarchy between concepts	299
A.8.3 Support making abstractions	300
A.8.4 Support the specification of analogies	300
A.8.5 Any model fragment can be seen as a pattern	301
A.8.6 Distinguish performative from constative speech acts	302
B Lingo, Neuro, Psycho	305
Bibliography	307

1 Introduction

“If you can’t change the world,
change yourself.
And if you can’t change yourself,
change your world”

Lonely Planet, The The, 1993

Contents

1.1	Background	2
1.1.1	Vision: Software Development is a Human Task	2
1.1.2	Research Goal	5
1.1.3	Method Objectives	5
1.2	Research Questions	8
1.3	Research Methodology	9
1.4	Thesis at a Glance	10
1.5	Research Contribution and Authorship	12

Welcome to the first chapter of this thesis, which sets the outline for the obtained results. We start by explaining the background in the next section, which is the starting point for our research. Section 1.2 presents the research questions of this thesis and Section 1.3 briefly clarifies the research methods used to address them. Section 1.4 gives an overview of the chapters, and Section 1.5 lists the publications that contributed to them.

1.1 Background

This section¹ describes the context of the research in this thesis. It explains our vision on software development, the main research goal, and the derived method objectives.

1.1.1 Vision: Software Development is a Human Task

Software development can be seen as the integration of knowledge and decisions from multiple people who are responsible for different aspects. Software languages are mostly based on computer-based primitives, which was logical when computers were expensive in comparison to developing costs. Today's software costs are dominated by the cost of development personnel. In other words, people are more

This section is based on sections of the publications R. Deckers and P. Lago, “Engineering MuDForM: A Cognition-based Method Definition”, Software and Systems Modeling, Springer, 2024, Under submission [39], R. Deckers and P. Lago, “Systematic Literature Review of Domain-oriented Specification Techniques”, In Journal of Systems and Software, Elsevier Science Inc., Vol 192, 2022 [37], and R. Deckers and P. Lago “Specifying Features in Terms of Domain Models: MuDForM Method Definition and Case Study”, Journal of Software: Evolution and Process, Wiley, 2023 [38].

1.1 Background

important for the success of a development activity than computers. A development method should reflect this. Our vision is that software development is intrinsically a cooperative human task, which should be supported by a method that is based on (human) cognitive concepts, instead of computer-based concepts.

The vision is based on Chapter 2 of the book DYASoftware [40] and elaborated in our position report [31]. The writing of that chapter led to the need to investigate what specification method could realize the vision. The vision is formed by four pillars, which each address an important issue in today's software development practice. These issues frame the research goal that we want to address.

A development method should enable people to communicate and reason in their own domain language. Software development has started as a specialism, in which software engineers perform every development task, and need to have all the required knowledge to do so. Since then, software systems have become larger and the number of stakeholders and aspects to deal with during their development has increased. Software development involves people in many areas of expertise. These people each speak their own 'language' with their idiom, and sometimes with their own grammar and syntax. Nevertheless, the practice is that we still use (development) languages that are an aggregation of computer-based primitives. A development method should support the languages of multiple domains, which involves capturing a domain language, as well as using it.

A development method should help to guard the consistency of all specifications in the development process. Tools and methods, each with their own notations and artifacts, are already available and used for some domains in the development process, e.g., requirement management, project management, modeling, or code checking. These tools and methods are mostly not integrated in a coherent chain, despite their overlap in concepts and their interdependencies. For example, a requirements engineer uses a requirement management tool, and a tester uses a test tool. Although a test tool might allow you to refer to requirements, there is typically no support to guard the consistency between the requirements specification and the (related) test specifications. Integral specification consistency is often not managed explicitly, let alone guaranteed. A development method should have an integral mechanism to relate specifications.

Development methods should support the explicit specification of any quality. In the early days of software development, a computer with software offered functionality in a way that was not possible before. Nowadays, functionality of software is only unique for a short time, and is quickly offered by multiple suppliers. The business

Chapter 1. Introduction

success factor of software has shifted from functionality to system qualities like ease of use, availability, security, and sustainability, and to development qualities like time-to-market, development cost, and project predictability. In spite of that, development methods and tools are still focused on delivering software functionality instead of guaranteeing quality. Architecture approaches related to quality, like ATAM [91] and architectural tactics [15], systematize the process, but not the specification of the involved requirements, designs, and their relations. Moreover, the desired quality is often not specified at all. The effect is that personal beliefs and experience of developers determine the quality of developed software. When a development team consists of people with different backgrounds, this results in inconsistent quality throughout a system and its development. This situation clearly contradicts the just stated business importance of quality. Any system property, including quality, cannot be proven, guaranteed, or engineered consistently, when it is not specified. A development method should be independent of any domain, while supporting various standard specification aspects, *e.g.*, state, structure, behavior, and agency, and be extendible with other aspects.

Development methods should support the reuse of any decision made in a system's lifetime: from idea to requirement, design, test, and system evolution. A software system is the result of a series of development decisions. Making good decisions costs significant time and money. One must learn the involved domains and consider trade-offs between all kinds of concerns. The reuse of a good decision in another context increases the return on investment in that decision. To make a decision reusable, it should at least be specified. Most reuses of decisions go via artifacts that capture the result of a set of decisions, *e.g.*, a piece of source code. Those artifacts can be part of the working system like a GUI library, a DBMS, or a rule engine, or can be part of any intermediate product during development, like a standard set of security requirements, process models, or design patterns. Also reusable transformations, *e.g.*, via a code generator, can be artifacts that capture development decisions. This kind of decision reuse is common practice in today's software development. However, it is often unclear which decisions exactly are reused, and whether they should have been reused at all. The result can be that expensive development knowledge is wasted, because the decisions are not reusable. The reuse of decision results starts with their specification. A development method should produce specifications that have a defined meaning, and that do not have implicit dependencies (on their context).

The story above must be seen as a vision of how system development could be. This thesis intends to pave the path towards it.

1.1.2 Research Goal

Some of today's techniques solve part of the stated research problems, *e.g.*, like natural language processing, domain modeling, domain-specific languages, model driven development, method engineering, design patterns, product line architectures, and IDEs. Furthermore, many formalisms, *e.g.*, set theory, calculus, and process algebra, may be used to enable automated reuse of specified decisions. But these techniques and formalisms are not integrated, and not applied to every aspect of a system's lifecycle.

To bring software development close to the thinking and communication of all stakeholders, we want to connect it to cognitive sciences, linguistics, and philosophy. Our research will realistically not be able to cover all potentially relevant knowledge in these fields. We mostly want to break ground for approaching software engineering as a process of knowledge capturing, more than a process of realizing executable software functionality. We understand that functionality is relevant. But we advocate that it should not be the foundation and focus of a development approach. A method and language based on cognitive concepts instead of machine primitives, and that is independent of any domain, should be the foundation of a system's lifecycle that does not suffer from the issues described in the previous section. This leads to our main research question (RQ):

What specification method enables all people involved in system development to unambiguously specify knowledge and decisions in a way that is close to how they think and communicate?

1.1.3 Method Objectives

Based on the vision and our experience with domain modeling, architecture, and model driven development, we have defined a set of objectives for our method – called MuDForM (Multi-Domain Formalization Method). They are not intended as a complete set of method requirements, but they form the rationale for the method definition, and for the method engineering approach. They also form a yardstick for evaluating existing specification techniques as potential content for the definition of MuDForM. Table 1.1 lists the number and name of each method objective, which will be used for reference throughout the thesis.

Chapter 1. Introduction

Number	Name
O1	Distinct domains
O2	Domain-independent
O3	Self-contained specifications
O4	Distinct prescriptive and descriptive specifications
O5	Integrated state and change
O6	Based on natural language and cognition
O7	Engineered method
O8	Related to architecture

Table 1.1: Number and name of the method objectives

The explanation of the objectives is as follows:

- O1 In any software development process, there are people that have concerns about different aspects. People use terminology from the domain that they are knowledgeable about, when they express their knowledge and decisions. A development method should support the **distinction of multiple domains** in one specification, to deal with the multitude of aspects in a development process. Moreover, a specification method should offer multiple mechanisms, *e.g.*, composition, consistency rules, transformation, or weaving, to integrate domain specifications, and domain-based specifications.
- O2 There is no limitation to what kind of aspects can be relevant in software development. A specification method should therefore be **independent of any domain** or system, and a method user (modeler) should not need any prior knowledge about the domain or system that is being specified. In practice, domain modeling is mostly used for the application domains of targeted systems, or for the design aspects of software. We think domain modeling should also be applicable to quality domains, such as reliability, security, usability, or functional safety, like in the case study of Chapter 5.
- O3 Knowledge about particular aspects can be seen independently of any specific (software) system, and is potentially usable in more than one system. A specification method should reflect this and produce **self-contained specifications** that are independent of their application in a specific system. To use specifications in different contexts, *i.e.*, to build other specifications, they should be composable, interpretable, and translatable.
- O4 Specifications can be made to analyze what is known about a domain, *i.e.*, specify what can happen and what can exist, and to design what should be true

1.1 Background

for a domain, *i.e.*, specify what shall happen and what shall exist. Therefore, a specification method should support the **distinction between descriptive specifications and prescriptive specifications**, and their relations.

- O5 Most domains and systems are not only about entities with a state, but also about change. A method should therefore support both the **specification of state** at a certain moment, and the **specification of change** of state over time. In other words, specifications should address things that exist, things that happen, and how those things are related. This perspective is similar to the notion of structural (static) properties and behavioral (dynamic) properties in UML [120].
- O6 Almost all people, including domain experts, use natural language to convey their knowledge and decisions. It is used in many documents in any software development process. A specification method should support the **transformation of knowledge** stated in **natural language** into specifications in an unambiguous language. Preferably, such specifications should themselves be translatable into natural language. The purpose of this support is to minimize loss of semantics and better mutual understanding in the communication between modelers and domain experts. The use of natural language as entry point for modeling is a stepping stone for the realization of the MuDForM vision to have method concepts that are close to human cognition. Natural language is a starting point, but probably does not cover all relevant cognitive concepts, because it evolved for communication and not specifically for thinking.
- O7 A specification **method should be engineered**, which means it should support specification consistency and completeness, elicitation of knowledge, and traceability of decisions, and it should have a comprehensive definition. According to Kronlöf [97] a method definition should contain the following ingredients:
 - An underlying model, *e.g.*, metamodel, core model, or abstract syntax, which defines the modeling concepts, *i.e.*, the elements of the modeling language, and the semantics of a specification.
 - An explicit notation, possibly used in different viewpoints. All the viewpoints of the method should be defined in terms of the concepts of the underlying model.
 - Explicit method steps that guide the modeler, from eliciting knowledge from documents and experts, to making a consistent and complete model.

Chapter 1. Introduction

- Guidance for taking steps and making specification decisions.
- O8 The purpose of a specification is mostly to (partially) realize (part of) a system. Hence, the transition from a set of created (domain-oriented) specifications into a working system should be feasible. In other words, there should be a **clear relation between specification method and architecture**.

1.2 Research Questions

To guide the research work, we have defined five sub-questions (RQ1-RQ5), which evolved in pursuit of the main RQ. First, we investigate what existing techniques are available to address the main RQ, which led to the following research question:

RQ1: What are existing techniques to create domain-oriented specifications, and to what extent do they support the method objectives?

The answer to RQ1 potentially offers content for our specification method and enables us to provide an understanding of the existing research gaps. Three other research questions are identified from these gaps. One of the identified gaps is that the available methods for domain-oriented specifications lack in their definition maturity level. Given that we aim to create a cognition-based specification method, this led to the following research question:

RQ2: How to define a specification method that is explicitly cognition-based?

Another identified gap between the method objectives and the state-of-the-art domain-oriented specification techniques is the lack of methodical support for using natural language in the specification process. This led to the following research question:

RQ3: What methodical support can be given for the conversion of text into a domain-oriented model?

Another identified gap between the method objectives and the available domain-oriented specification techniques is the lack of support for using domain models as the terminology for other specifications, and feature specifications in particular. This led to the following research question:

RQ4: What methodical support can be given for the specification of a feature, such that it is defined in terms of domain models?

1.3 Research Methodology

To fully address RQ2, we also needed input from cognitive science literature on what concepts should be implemented in the method definition. We decided to read books to look for knowledge that could play a role in the creation of MuDForM. This led to the following research question:

RQ5: What are common elementary cognitive concepts that people use to think and communicate, and how can they be applied in engineering a specification method?

1.3 Research Methodology

In light of the diversity of the research questions, we have chosen to apply several research methods. Below, we provide an exploration of each method. This information serves to contextualize the research questions and their associated findings, as summarized in Section 1.4, and represented in Figure 1.1.

Systematic Literature Review (SLR): This research method serves as an evidence-based technique, employed to examine well-established research topics. It allows for an objective approach to filtering academic literature, conducting data extraction and analysis, and deriving conclusions from the gathered data. Our utilization of this method primarily addresses RQ1. In Chapter 2, we apply this method to identify useful method content and research gaps. The outcome of is a collection of primary studies, and an analysis of how they fit the method objectives.

Action Research: This is a research method that aims to simultaneously investigate and solve an issue. In other words, as its name suggests, action research conducts research and takes action at the same time. We followed the description by Petersen *et al.* [121], resulting in the phases of Diagnosis, Action Planning, Action Taking, Evaluation, and Specifying Learning. Chapter 3 explains how action research is applied to address RQ2. Chapter 5 explains how action research is applied to address RQ4. Similarly, action research is applied to address RQ3 in Chapter 4, but it is not explained in much detail there.

The Action Taking phases of the research presented in Chapters 3, 4, and 5, were actually one action. We defined an initial version of the method, based on our more than 25 years of experience in creating, using, and maintaining models of various domains and systems, in the context of business analysis, requirements engineering, functional design, software architecture, process modeling, and test specification. The result was used as starting point for the mentioned chapters.

Chapter 1. Introduction

Qualitative Data Analysis: This research method involves examining nonnumerical data such as texts, images, or observations to uncover patterns and meanings [81]. The main steps typically include data collection, data organization, coding the data, analyzing the data to draw conclusions, and finally reporting of insights. This method is applied to address RQ5 in Chapter 6.

1.4 Thesis at a Glance

Figure 1.1 provides an overview of this thesis. Chapter 1 introduces the vision, the research goals, and the derived method objectives, which form the starting point of the research. It introduces the main research question and the derived research questions, which are addressed in the other chapters.

Chapter 2 answers RQ1 via a systematic literature review (SLR) of 53 primary studies in domain-oriented specification techniques. It compares the techniques on their applicability, their level of method engineering, and their suitability for domain-oriented modeling. The SLR has identified several gaps in current literature, of which the following are addressed in this thesis: 1) there is no fully engineered method, *i.e.*, existing ones lack a metamodel, notation, process, or guidance, 2) natural language is supported minimally, and 3) there is no methodical support for creating specifications in terms of a domain model or domain-specific language. These gaps have respectively led to RQ2, RQ3, and RQ4, which are addressed via an action research approach, and answered in Chapters 3, 4, and 5, respectively.

Chapter 3 answers RQ2. It clarifies how the method contributes to the vision that software development should be based on concepts from human cognition. It explains the process and framework for developing our MuDForM. It provides a coherent and detailed method definition, and explains how cognitive concepts are implemented in the method. The definition comprises method construction patterns, a metamodel, viewpoints, modeling steps, and guidelines. Parts of the method are demonstrated with fragments of an industry case study.

Chapter 4 answers RQ3. It presents the part of MuDForM that deals with the transformation of text into an initial model. It presents the metamodel, analysis steps, and guidelines for that transformation, and demonstrates them with fragments from an industry case study. The guidelines are the result of over 25 years experience with grammatical analysis, combined with content from primary studies in the SLR. The chapter also explains how the text-to-model transformation can be gradually improved to deal with more grammatical constructs.

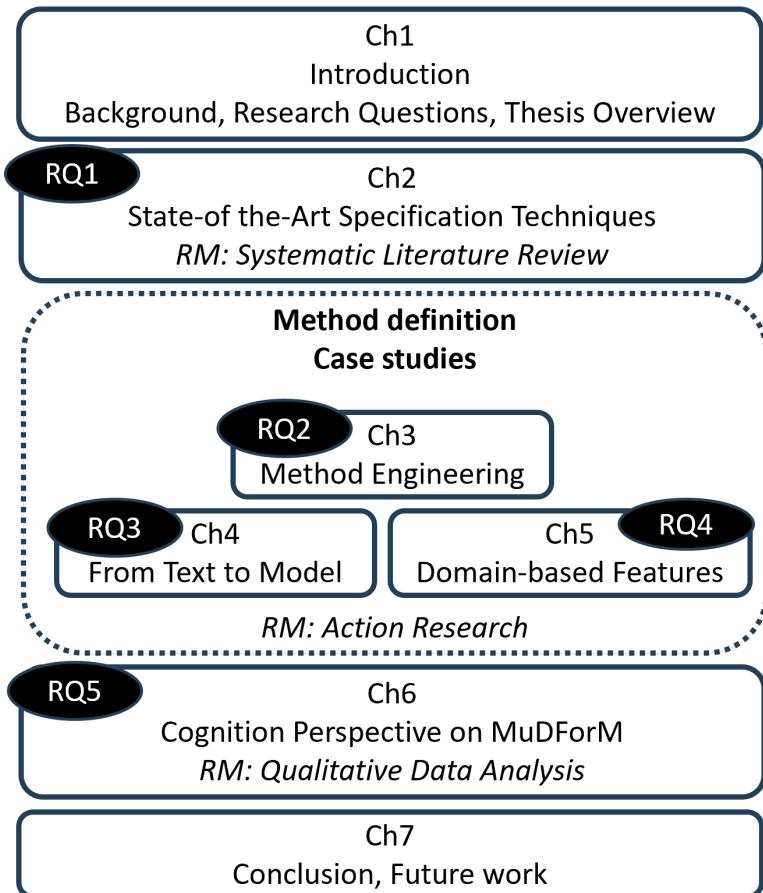


Figure 1.1: Thesis overview: chapters (Ch), research questions (RQ), and research methods (RM)

Chapter 5 answers RQ4. It presents the part of MuDForM that deals with the specification of features in terms of domain models. It presents the modeling concepts and modeling steps, and provides an extensive list of guidelines. All those method ingredients are demonstrated and discussed via a case study from industry, in which a model is made from the ISO/IEC26262 standard for functional safety in automotive systems. It is an example of modeling a quality attribute.

Chapter 1. Introduction

In parallel to the work for the other chapters, we performed a qualitative data analysis of literature in cognitive science, philosophy, and linguistics. It produced findings for the complete scope of MuDForM and its development, and analyses that underpin some method design choices, reflect on method design choices, or suggest improvements. The main result is that we break ground for making software development cognition-based, and explored how it can be done. This chapter is not technical. It illustrates how the fields of software engineering and cognitive science can be combined. Chapter 6 presents the results, which answer RQ5.

Chapter 7 concludes this thesis. It shortly discusses the research questions, evaluates MuDForM via the classification framework of the SLR in Chapter 2, suggests future work, and reflects on the main research goal.

1.5 Research Contribution and Authorship

This section lists the publications on which this thesis is based, and some earlier publications, in which the method presented in this thesis is also (partially) applied.

Chapter 1, and in particular Section 1.1, is based on the introduction and background sections of:

- R. Deckers and P. Lago, “*Engineering MuDForM: a Cognition-based Method Definition*”, Software and Systems Modeling, Springer, 2024, Under submission [39].
- R. Deckers and P. Lago, “*Systematic Literature Review of Domain-oriented Specification Techniques*”, In Journal of Systems and Software, Elsevier Science Inc., Vol 192, 2022 [37].
- R. Deckers and P. Lago “*Specifying Features in Terms of Domain Models: MuDForM Method Definition and Case Study*”, Journal of Software: Evolution and Process, Wiley, 2023 [38].

Chapter 2 is based on:

- R. Deckers and P. Lago, “*Systematic Literature Review of Domain-oriented Specification Techniques*”, In Journal of Systems and Software, Elsevier Science Inc., Vol 192, 2022 [37].

Chapter 3 is based on:

- R. Deckers and P. Lago, “*Engineering MuDForM: a Cognition-based Method Definition*”, Software and Systems Modeling, Springer, 2024, Under submission [39].

1.5 Research Contribution and Authorship

Chapter 4 is based on:

- 〔 R. Deckers and P. Lago, 〔 R. Deckers and P. Lago “*Methodical Conversion of Text to Models*”, CEUR Workshop Proceedings of the PoEM 2022 Forum, 15th IFIP Working Conference on the Practice of Enterprise Modeling 2022, November 23–25, pages 113–127, CEUR-WS.org, 2022 [36].

Chapter 5 is based on:

- 〔 R. Deckers and P. Lago, “*Specifying Features in Terms of Domain Models: MuDForM Method Definition and Case Study*”, Journal of Software: Evolution and Process, Wiley, 2023 [38].

All these chapters and publications have been authored by Robert Deckers and Patricia Lago. Robert Deckers performed the conceptualization, research methodology definition, investigation, data curation, and writing. Patricia Lago took care of supervision and reviewing.

While conceptualizing and creating the method that forms the core of this thesis, *i.e.*, MuDForM, Robert Deckers guided the modeling process and co-authored the presented domain-oriented models in the following publications:

- 〔 R. de Boer, R. Farenhorst, V. Clerc, J. van der Ven, R. Deckers, P. Lago, and H. van Vliet, “*Structuring Software Architecture Project Memories*”, 8th International Workshop on Learning Software Organizations (LSO), Rio de Janeiro, Brazil, p39–p47, Springer, 2006 [29].
- 〔 R. Farenhorst, R. de Boer, R. Deckers, P. Lago, H. van Vliet, “*What's in Constructing a Domain Model for Sharing Architectural Knowledge?*”, In Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE), pages 108–113. Knowledge Systems Institute Graduate School, 2006 [56].
- 〔 F. Moghaddam, R. Deckers, G. Procaccianti, P. Grossi, and P. Lago, “*A Domain Model for Self-adaptive Software Systems*”, Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, p16–p22, ACM, 2017 [114].
- 〔 Chapter 3 of the PhD thesis “*Functional Safety: A New Architectural Perspective: Mode l-Based Safety Engineering for Automated Driving Systems*” by A. Khabbaz Saberi, Eindhoven University of Technology, 2020 [93].

2 Systematic Literature Review of Domain-oriented Specification Techniques

“We get these pills to swallow
How they stick in your throat?
Tastes like gold”

No one knows,
Queens of the Stone Age, 2002

This chapter is based on:

✉ R. Deckers and P. Lago, “*Systematic Literature Review of Domain-oriented Specification Techniques*”, In Journal of Systems and Software, Elsevier Science Inc., Vol 192, 2022 [37].

Chapter 2. State-of-the-Art Specification Techniques

This chapter addresses RQ1.

Context. The popularity of domain-specific languages and model driven development has made the tacit use of domain knowledge in system development more tangible. Our vision is a development process where a (software) system specification is based on multiple domain models, and where the specification method is built from cognitive concepts, presumably derived from natural language.

Goal. To realize this vision, we evaluate and reflect upon the existing literature in domain-oriented specification techniques.

Method. We designed and conducted a systematic literature review on domain-oriented specification techniques. The approach uses a classification framework that is derived from the method objectives from Section 1.1.3.

Results. We identified 53 primary studies, populated the classification framework for each study, and summarized our findings per classification aspect. We found many approaches for creating domain models or domain-specific languages. Observations include: (i) most methods are defined incompletely; (ii) none offers methodical support for the use of domain models or domain-specific languages to create other specifications; (iii) there are specification techniques to integrate models in general, but no study offers methodical support for multiple domain models.

Conclusion. The results indicate which topics need further research and which can instead be reused to realize our vision on system development.

Contents

2.1	Introduction	17
2.2	Background: MuDForM and Domain Modeling	20
2.2.1	What is a Domain Model?	20
2.2.2	The MuDForM Vision	21
2.2.3	MuDForM Objectives	22
2.3	Study Design and Execution	22
2.3.1	Research Questions	23
2.3.2	Search Queries	27
2.3.3	Selection Criteria	29
2.3.4	Study Execution	30
2.3.5	Classification Framework	31
2.4	Study Results	36
2.4.1	Publication Trends	37
2.4.2	Application Scope	39
2.4.3	Method Engineering	40
2.4.4	MuDForM Specific	43
2.5	Discussion	44
2.5.1	RQ1.1: No Fully Engineered Method	45
2.5.2	RQ1.2: No Methodical Support for Applying a Created DSL or DM	45
2.5.3	RQ1.3: No Integral Support for Multiple Domains	47
2.5.4	Behavior is Mostly Ignored and Poorly Integrated with Structure	47
2.5.5	Minimal Interface with Natural Language	48
2.5.6	The Terminology around Domain Models is not Unified	48
2.6	Related Work	49
2.7	Threats to Validity	51
2.8	Conclusions and Future Work	52

2.1 Introduction

Since the 1980s, several approaches for domain modeling have been developed and published [143, 148, 96]. Domain modeling is a technique to capture knowledge from

Chapter 2. State-of-the-Art Specification Techniques

domain experts and domain literature into a model. A domain model can be used in various ways, *e.g.*, as a basis for requirements specification [58], as a basis for software design [53], or as a language for functional specifications [96].

The purpose of this systematic literature review (SLR) is to investigate which specification techniques exist in the context of domain modeling. These are both techniques to *create* domain models, and techniques to *use* a domain model *as a language* for other specifications, *e.g.*, the specification of a feature, application, or system aspect. We call specifications that are expressed in terms of a domain model, *domain-based specifications*. In order to create system specifications, we are also interested in techniques to *integrate* domain models and to integrate specifications expressed in terms of domain models. We call all these techniques together *domain-oriented specification techniques*. The outcome of this study is used as input for the definition of MuDForM in Chapters 3, 4, and 5.

Domain models are intended to capture knowledge about the application domains of systems [95, 55]. But we are also interested in applying domain modeling to other domains, and quality domains in particular. Most methods related to domain modeling focus on the structural (state) properties of a domain, *e.g.*, [156, 63, 7, 173, 130, 73, 72, 25, 24, 82, 62]. Though, we are especially interested in methods that facilitate the specification of behavioral (dynamic) properties, and in such a way that they are well integrated with the specification of structural properties.

This chapter addresses **RQ1** of this thesis:

What are existing techniques to create domain-oriented specifications, and to what extent do they support the objectives?

For this SLR, RQ1 is detailed in three sub-questions that investigate what specification techniques exist to (**RQ1.1**) make specifications of a domain (called *domain specifications*), (**RQ1.2**) make other specifications in terms of a domain specification (called *domain-based specifications*), and (**RQ1.3**) integrate several domain specifications and domain-based specifications in one specification (called *integration specifications*). These different types of specification are explained further in Section 2.3.1. Per identified technique, we answer several questions. First, to which domains is it applicable, and is it applicable to quality domains? Second, how methodical is the technique? Third, how does it address a number of aspects that are specific for domain-oriented modeling?

2.1 Introduction

Abbreviation	Meaning
DM	domain model
DO	domain-oriented
DS	domain specification
DSL	domain-specific language
SMS	systematic mapping study
SLR	systematic literature review

Table 2.1: Frequently used abbreviations in this thesis

We have identified three **contributions of this SLR** together with the **related target audiences**. The first contribution is an overview of the state-of-the-art in specification techniques to create domain models (DMs) and domain-specific languages (DSLs), and their use in the creation of other (domain-based) specifications. (Table 2.1 lists frequently used abbreviations.) This SLR discusses how well those techniques are engineered, by analyzing the conciseness and clarity of the specification language and specification process. We also discuss how well the existing techniques cover the aspects that are derived from the specific objectives (introduced in Section 2.2.3) that we envision for our own research, *i.e.*, for the definition of MuDForM. Potential users, developers, and researchers of domain-oriented (DO) specification techniques, can use the overview to select techniques in order to apply them in their own context.

The second contribution is the identification of shortcomings in the existing literature on domain-oriented specification techniques. We identified topics that need to be researched in the future, such as the support for working with multiple domains, the integration of modeling concepts for structural and behavioral properties, and having fine-grained guidance for modeling decisions. Another gap in the literature is the incompleteness of most method descriptions. This is not a new research topic, but rather a lack in method engineering of those specification techniques. Researchers and developers of domain-oriented specification techniques may use the identified topics as a starting point for their research and method engineering activities.

The third contribution is that we have defined a reusable approach for comparing methods, which is an extension to the guidelines described by Kitchenham [94]. First, we have made a conceptual model of the domain of method engineering. This model is reusable for other method comparisons. Second, we have created a conceptual model of the application domain of the targeted methods, *i.e.*, *domain-oriented specifications*. The concepts from both models are used to define the research questions, the search queries, and the classification framework. The classification framework

Chapter 2. State-of-the-Art Specification Techniques

consists of three parts, which can be applied to any method comparison. Furthermore, the use of concept models of the method engineering domain, and of the application domain of the targeted methods, leads to a more consistent study design and execution. Researchers who also want to compare methods, possibly through a literature review, can benefit from the approach we followed.

The remainder of this chapter is structured as follows. Section 2.2 introduces some background knowledge. Section 2.3 describes the study design and execution. All the data produced during study execution is available via the replication package [35]. Section 2.4 presents the study results, *i.e.*, the extracted data from the primary studies that we included. Section 2.5 discusses the results from the perspective of the research questions, while Section 2.6 discusses related works. Section 2.7 addresses the threats to the validity of this study. Section 2.8 concludes this chapter, and identifies topics for future research.

2.2 Background: MuDForM and Domain Modeling

This section provides the background information of this SLR. This study is carried out as the starting point of our research, in which we work on an integral method for system specification via multiple domain models, *i.e.*, MuDForM¹. We mention our MuDForM research program in this SLR, because its objectives are the main reason for the RQs and the aspects in the classification framework.

Accordingly, the following explains our perspective on domain modeling, and the objectives we aim to achieve with MuDForM. These objectives will be used for the definition of the classification framework in Section 2.3.5, and as a yardstick in the discussion (Section 2.5) of the data extracted from the selected primary studies.

2.2.1 What is a Domain Model?

We found two different notions of DM in the literature. The notion that we use is that of a specification space, analogous to a domain in the mathematical sense. The term “domain” refers to an area of knowledge or activity and a DM describes what can happen (behavior) and what can exist (state and structure) in a domain, or in other words, what can be controlled and managed in a domain. A DM is the foundation for a shared lexicon in communication between stakeholders, and can

¹A MuDForM is used to shape tacit and “muddy” data into knowledge building blocks.

2.2 Background: MuDForM and Domain Modeling

serve directly as a structured vocabulary for making other specifications, or form the underlying model of a DSL. For example, a model of the banking domain expressed in a UML class diagram, can be used directly in other UML diagrams, or can serve as the abstract syntax of a DSL. A DM is not intended to express what should happen, does happen, is likely to happen, or has always happened in the domain, because we assign those aspects to different types of specifications, like a system, application, or feature specification. The knowledge captured in a DM is not limited to a specific way of working in the domain or to a specific system that operates in the domain. Approaches for DSLs, such as [3, 61, 92], comply with this notion of DM.

The other notion in the literature is that a domain is a collection of related systems. Accordingly, a DM defines a set of (system) features that are common in the domain. This notion is used, for example, by FODA [90]. According to this notion, a DM can only be made with a set of systems or features in mind, while in the notion that we adhere to, one can talk about the concepts in a domain independently of any feature or system.

2.2.2 The MuDForM Vision

We envision software development as a process in which the involved people make decisions in their own area of knowledge, *i.e.*, *domain*. Those decisions must be integrated, and finally result in a machine-readable specification. That is why our research focuses on an integral method for creating DMs, for using DMs as a language to create other (domain-based) specifications, and for integrating multiple DMs and domain-based specifications. It is the ultimate intention for a system to be completely defined in domain-oriented specifications, and that if other kinds of specifications are used, then they are also explicitly integrated.

We envision that a major difference between MuDForM and most other methods is that, in addition to modeling the objects in a domain, MuDForM also considers domain actions to be first-class domain concepts, and MuDForM integrates objects and actions. Domain actions describe the atomic changes in their domain. They are elements for the creation of composite behavioral specifications, *e.g.*, processes, scenarios, and system functions. Another difference comes from our notion of DM: DMs are descriptive and the result of analysis, and system specifications, *e.g.*, feature models, are design artifacts and prescriptive. We foresee that the third major difference is the extensive use of natural language processing in the modeling process.

2.2.3 MuDForM Objectives

The main motivation for this SLR is to identify and characterize what solutions the existing literature provides for the method objectives presented in Section 1.1.3, in order to use those solutions in the development of MuDForM. The objectives are the starting point for the design of the classification framework described in Section 2.3.5.

We, the authors of this SLR, have a background in software architecture, domain modeling and model driven development. When we started to work on MuDForM, we already knew of specification techniques from several books on domain modeling and domain-specific languages [53, 96, 92, 164, 107, 61]. We observed that those books did not address all the objectives. Especially, the use of natural language processing and dealing with multiple models are topics that are hardly addressed. We did a preliminary informal literature scan on these topics and found some useful studies [24, 102, 82, 137, 62, 52, 161, 2], but they were mostly not containing relevant content for answering our research questions. Hence, this SLR.

2.3 Study Design and Execution

This section describes the design and execution of our SLR. We follow the guidelines described by Kitchenham [94]. The purpose of this SLR is to investigate what specification techniques exist in the context of domain modeling, and compare them on their applicability, degree of method engineering, and how well they support the aspects that are derived from the MuDForM objectives. We are especially interested in techniques for analysis of natural language, techniques for handling multiple domains, and guidelines for modeling decisions. Another goal is to identify research topics related to the method objectives, based on shortcomings and gaps that we detected in the existing literature.

Section 2.3.1 explains the research questions. From those questions and the inclusion/exclusion criteria (Section 2.3.3), we derive the search queries (Section 2.3.2) and the classification framework (Section 2.3.5). Section 2.3.4 describes the search process, *i.e.*, study execution. Based on the data extracted from the search results (described in Section 2.4), Section 2.5 discusses the answers to our research questions.

Of course, the results of this SLR, and the references to the found specification techniques in particular, can also be used by researchers and practitioners that investigate and develop domain-oriented specification methods.

2.3.1 Research Questions

This section elaborates on the research questions (see Section 2.1) of this SLR. In order to formulate RQ1.1-RQ1.3, the derived search queries, and the classification framework, in a coherent and unambiguous way, we created a conceptual model of the research domain of this SLR, *i.e.*, the domain of domain-oriented specification techniques. This model is presented in four class diagrams throughout this section.

We are interested in three categories of *specification techniques*²: *domain specifications*, *domain-based specifications*, and *domain-oriented integration specifications*, which we all classify as *domain-oriented specifications* (see Figure 2.1). To be clear, this doesn't mean that the specification techniques themselves have to be explicitly domain-oriented. We are interested in all specification techniques that can be used to create domain-oriented specifications. Figure 2.1 also depicts that each category corresponds to a research question that starts with the phrase "What are specification techniques to create ...?". We will now explain each research question.

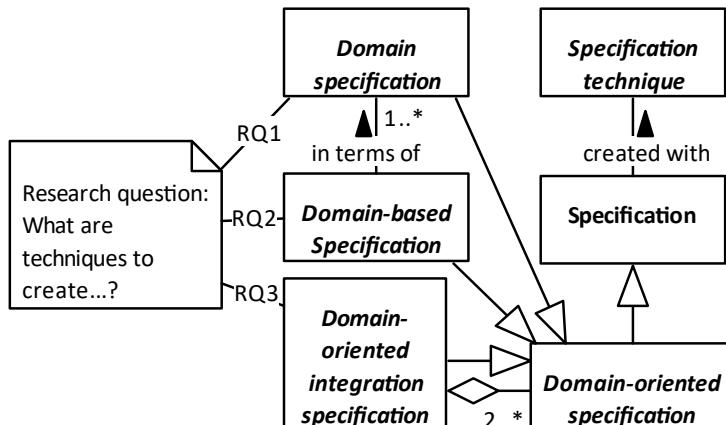


Figure 2.1: Positioning the research questions

Domain specifications (RQ1.1): *What are techniques to create specifications of a domain?*

We are interested in structured specifications of a *domain*, such as *domain models*, *domain-specific languages*, and *domain ontologies*. We found several

²The italic words in the text refer to elements in the models.

Chapter 2. State-of-the-Art Specification Techniques

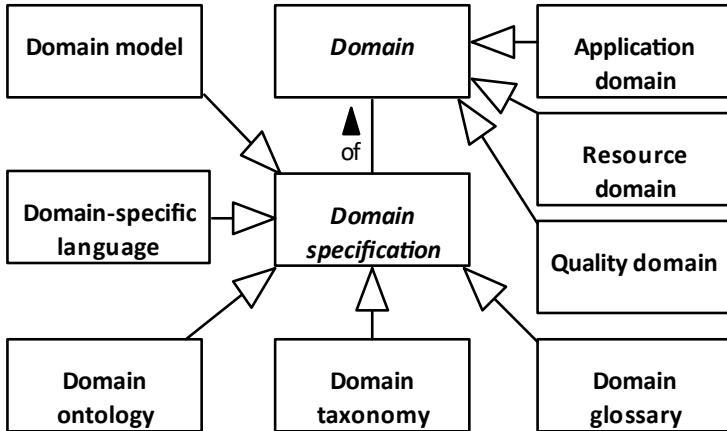


Figure 2.2: Concepts pertaining RQ1.1

types of *domain specifications* (see Figure 2.2). But we are not looking for techniques that result in just an enumeration of concepts in a domain, such as *domain glossaries* or vocabularies, because they have no explicit structure. *Taxonomies* sometimes have a hierarchical structure, but typically do not provide insight in how concepts at one hierarchy level relate to each other.

The most common use of domain modeling techniques is for *application domains*. But we are also interested in techniques for other types of domains, in particular *quality domains*. There are many quality domains, denoted by quality attributes such as security, usability, or maintainability. We are not looking for specifications of these quality domains, but for techniques to create their specification.

Software design and programming can also be seen as a domain, *i.e.*, the *resource domain*. This large domain can be divided into subdomains (often called design aspects), such as user interaction, logging, persistence, rule checking, error handling, encryption, component deployment, load balancing, system decomposition, data communication, resource usage, and so on. We are not looking for specifications of these subdomains, but for techniques to create their specification.

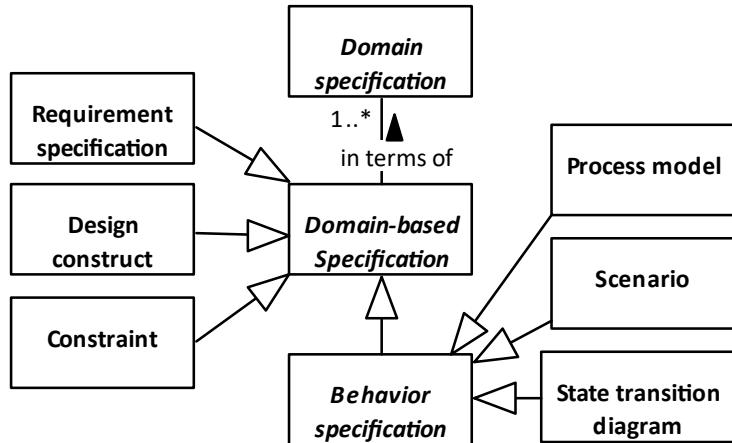


Figure 2.3: Concepts pertaining RQ1.2

Domain-based specifications (RQ1.2): *What are techniques to create specifications in terms of domain specifications?*

We are looking for techniques to create *domain-based specifications*, i.e., specifications in terms of a *domain specification*³. Figure 2.3 shows some examples of types of specifications that could be domain-based: *(quality) requirements*, *constraints*, *design constructs*, or *behavior specifications* like *process models*, *scenarios*, or *state transition diagrams*. Besides using a DS as terminology, they are also written in terms of a language that is specific for their type of specifications, such as a requirements language, constraint language, or process modeling language. Preferably, such a language is a DSL by itself, or at least specified via a DM.

To clarify how a domain-independent method could support the creation of a specification in terms of a DS, we describe three examples of specification techniques that address this RQ. First, if the domain elements that describe the behavior in a domain are used to specify processes steps, i.e., they are the types of process steps, then it is possible to detect overlap between processes regarding sequences of steps that occur in multiple processes. In such case, a method guideline can be given to identify sub-processes in a set of process models, e.g., “Define a separate process for those sequences of process steps that occur in multiple processes”. Second, if requirements or constraints are

³From hereon we will use DS (domain specification) instead of 'DM and/or DSL'.

Chapter 2. State-of-the-Art Specification Techniques

specified in terms of a DS, then method guidelines can be given to detect inconsistencies between requirements, e.g., “Check requirements that use the same domain class”. Third, if domain classes are used to specify the object structures in a system, then guidelines can be given for how to do this top-down, e.g., “Start specifying functions for domain classes that are not a part of a composition or aggregation”. More examples can be found in Chapter 5.

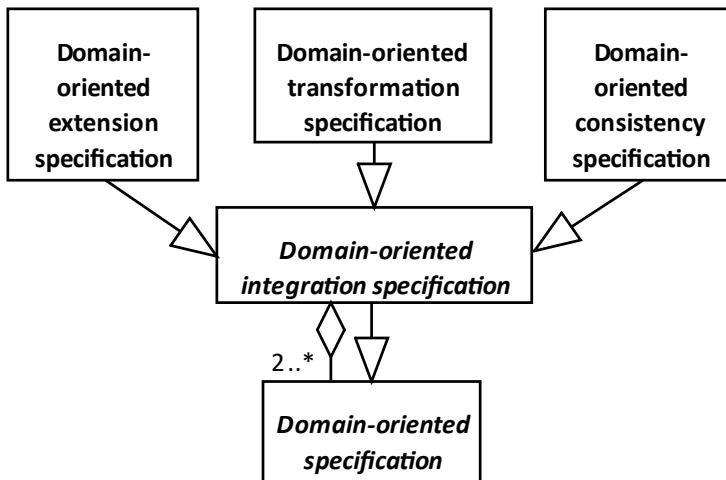


Figure 2.4: Concepts pertaining RQ1.3

Specification integrations (RQ1.3): *What are techniques to create specifications of integrations between domain specifications, and between domain-based specifications?*

We are looking for specification techniques to create *DO integration specifications* of two or more *DO specifications* (*domain specifications* or *domain-based specifications*) via explicit integration methods, languages, models, or other mechanisms. We distinguish at least techniques for *transformation*, *extension*, and *consistency* between DO specifications (see Figure 2.4). We see correspondence between specifications, as for example in the IEEE42010 standard for architectural descriptions [85], as a form of *consistency specification*. We see merge and composition, as for example in [48], weaving, as for example in [146], and other ways to combine two or more specifications into a new specification, as a form of *extension specifications*.

2.3.2 Search Queries

This section explains the creation of the search queries. The term specification techniques, which is used in all three research questions, is not commonly used in the literature as a denominator. Therefore, using this term will not give adequate results. That is why we first scanned through the literature we were familiar with, and made a model of the used terminology. Figure 2.5 shows different types of *specification techniques* that we found. It is not a complete lexicon, but a summary of the most common categories. The most occurring techniques are Specification (or Modeling) language, and Specification (or Modeling) method. Specification languages can be based on a Metamodel, (which could be defined in a metamodeling language). A Metamodel can also be the underlying model of a Specification method. Method definitions may also contain Method steps and Guidelines, and distinguish different Method viewpoints which use a Specification language as their notation. As such, Metamodel, Method step, Guideline, and Method viewpoint can be seen as partial Specification techniques, and are of interest for this study.

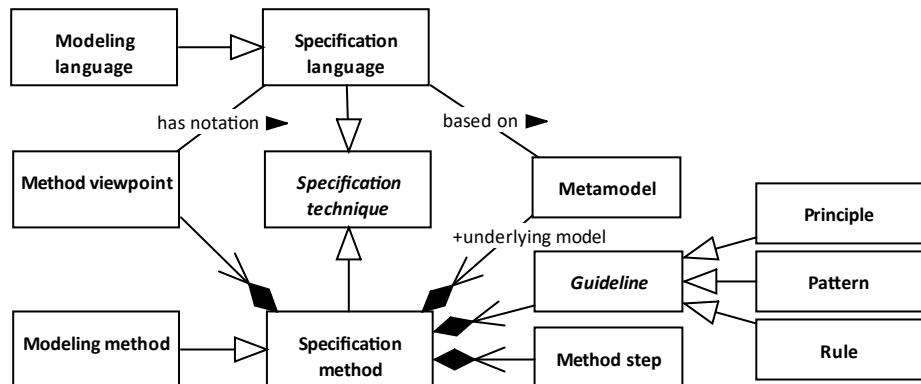


Figure 2.5: Examples of different specification techniques and their aspects.

It is not useful to just search for all types of techniques that we identified in the model, because this yields an unwieldy amount of results (more than 500,000 hits on Google Scholar). The search string was thus narrowed down in several steps to come to a manageable set of results:

1. We omitted *guidelines* (*principles*, *patterns*, *rules*), *method steps*, *viewpoint*, and *metamodel* because they should always be defined in the context of a method. We will still use these terms as a denominator in the data extraction.

Chapter 2. State-of-the-Art Specification Techniques

2. We omitted *ontology*, because ontology languages are in general less expressive than domain modeling languages. Namely, they are mostly limited to just capturing terms from the domain and relations between the terms. Some ontology languages go a bit further and distinguish classes, attributes, and different types of relations between classes. Though, almost all domain modeling languages also have those concepts. Though, if a study uses ontologies as an ingredient of a domain modeling approach, then we will include it if it matches the criteria.
3. Some authors use the term *approach*, mostly because they find the term method too specific. We do not want to ignore those studies. Thus, we include the term approach as a possible generalized and more informal term for method.
4. We use the term *language* instead of *modeling language* or *specification language*, because we always search for it in combination with specification or model.

Given RQ1.1 and the explanations of Figure 2.2, and after alternative terms that express the same semantics, we obtain the following search string:

```
((method) OR (approach) OR (methodology) OR (methods) OR  
(approaches) OR (methodologies)) AND ((domain-model) OR  
(domain-specific-language) OR (domain-models) OR (domain-modeling)  
OR (domain-modelling) OR (domain-specific-languages))
```

This still led to more than 17,000 hits on Google Scholar. Therefore, we decided to limit the search string to the title of the publication and compensate this limitation with snowballing.

Further, because RQ1.2 and RQ1.3 include specification techniques as well as domain specifications, we reckon that the defined search string also covers these questions. So, RQ1.2 and RQ1.3 are not framed in two distinct search strings. Instead, we address them in the classification framework with their own classification aspect, as explained in Section 2.3.5.

We run our search queries on Google Scholar, because it covers all well-known scientific publication sources, such as ACM, Springer, and IEEE.

2.3 Study Design and Execution

2.3.3 Selection Criteria

This section addresses the inclusion/exclusion criteria that are used to select the primary studies.

The inclusion criteria are:

- (I1) Research publications subject to scientific peer review. Studies that were not submitted to scientific peer review might have claims that are not objectively verified on credibility. So, journal papers, PhD theses, and papers in conference or workshop proceedings, are considered. Also books and technical reports issued by respected institutes or authors are taken into account. But white papers, or articles in commercial magazines, are discarded.
- (I2) Studies written in English.
- (I3) Studies available online as full text. Exceptions can be made for well-known books on the subject.

The exclusion criteria are:

- (E1) Studies that do not contribute any specification technique for DO specifications, which includes DMs, DSLs, domain-based specifications, and DO integration specifications. For example, we exclude studies in which specifications, like requirements or DSL definitions, are only used as an example, while they do not explain how to make them.
- (E2) Studies that focus on techniques for testing, reviewing, or checking specifications. We are looking for techniques in the context of system development, *i.e.*, the creation and maintenance of domain-oriented specifications.
- (E3) Studies that focus on techniques for human behavior or on how to organize the specification process. For example, SCRUM prescribes the specification of all work items, but it does not address how to specify them or how to apply them correctly.
- (E4) Secondary and tertiary studies, like systematic literature reviews, and surveys. It is important to note that, though secondary studies are excluded, we may use them for precisely scoping the contribution of this SLR and for checking the completeness of the set of selected primary studies.
- (E5) Studies that describe an approach for creating a DS and that do not comply with our notion of DM as explained in Section 2.2.1. As such, we exclude studies that consider a domain as a set of systems or applications. We are interested in

Chapter 2. State-of-the-Art Specification Techniques

studies that see a domain specification as a language, and not as a framework to specify applications and systems. Of course, we will include a study if parts of it offer specification techniques that comply with our notion of DM.

(E6) Studies that focus mainly on implementing DSs in a target environment without using an explicit DS of that environment. Transformation specifications from a source DS to a target DS are in scope. But transformations from a source domain to program code, without an explicit DS of the targeted software environment, are excluded.

2.3.4 Study Execution

As depicted in Figure 2.6, we followed the guidelines described by Kitchenham [94], leading to the following steps and search results:

1. The *initial search* took place on June 15, 2020, and led to 602 unique studies.
2. Then we *applied the criteria* in three exclusion stages: based on the title, based on the abstract, and based on the full text. This resulted in the inclusion of 20 primary studies. Besides those, we also kept 9 of the excluded studies for snowballing, because we found relevant citations during reading them.
3. We applied *snowballing* (as described by Jalali and Wohlin [88]) based on the citations in the already included studies, and in the studies we kept for references. This led to the selection of 125 extra references.
4. By *applying the criteria* to those, we selected 19 extra studies, bringing the total to 39 studies.
5. As indicated in Section 2.2.3, we added several relevant *books and studies in an informal search*, namely [24, 102, 82, 137, 62, 52, 161, 2, 53, 96, 92, 164, 61, 107], and double-checked them against the selection criteria.
6. Finally, we *extracted the data* and performed the analysis on a total of 53 primary studies, of which 7 are books and the rest are articles in journals, conferences, workshops, and reports published by well known academic institutes.

We care to note that most papers were excluded because they did not contribute a specification technique (E1). The majority of them were about a specific DS and its

2.3 Study Design and Execution

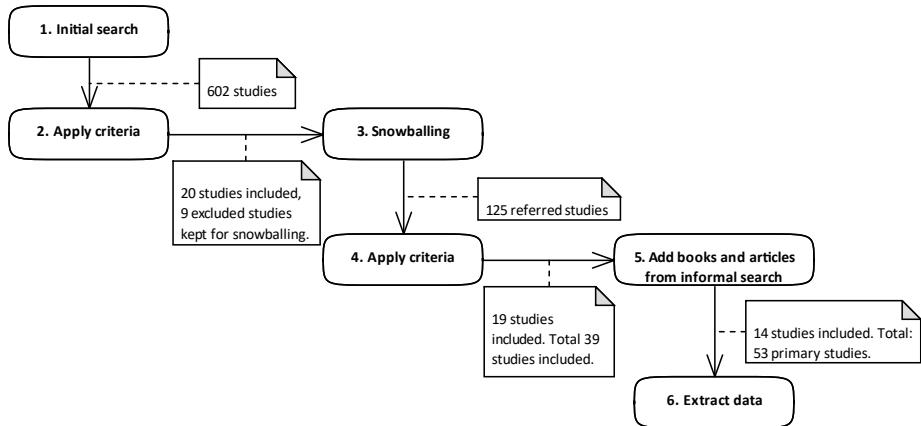


Figure 2.6: Study execution

usage, and did not offer an explanation of the used specification technique to create or use that DS.

Many other studies were excluded because they were about code generation without considering the target environment as a domain, and thus not treating code generation as a form of domain integration (E6).

We found a few PhD theses on the topic of domain-oriented specifications, but we did not find articles that were part of, or derived from those theses, and none of the theses actually explained the specification techniques used to create or to use the DSs. But we kept theses in the process for snowballing their references, because they had relevant citations, as mentioned in step 2 above. The details of the study execution are available in the replication package [35].

2.3.5 Classification Framework

This section discusses the aspects that we use to analyze and compare the primary studies. We distinguish the aspects in three classification categories (see Table 2.2): application scope of the technique, method engineering level, and contribution to the MuDForM objectives from Section 1.1.3.

Chapter 2. State-of-the-Art Specification Techniques

Each aspect is explained below, and an indication of the possible values is given. All aspects have a default value of “not addressed” which means that the study does not cover the aspect at all. Another possible value is “mentioned”, which means that the aspect is recognized and possibly discussed, but that no clear contribution or solution is given.

Besides explaining all classification aspects, the next sections also state for each aspect (i) which research questions it helps to answer, and (ii) which MuDForM objectives it serves. A summary overview is also given in Table 2.2.

Table 2.2: Relation between classification framework, research questions, and MuDForM objectives

CLASSIFICATION CATEGORY	CLASSIFICATION ASPECT	HELPS TO ANSWER			SERVES OBJECTIVE						
		RQ1.1	RQ1.2	RQ1.3	O1	O2	O3	O4	O5	O6	O7
Application scope	Domain dependence Quality domains Architecture	■ ■ ■	■ ■ ■	■ ■ ■	□ □ ■	■ ■ ■	■ □ ■	□ □ □	□ □ □	□ □ □	□ □ ■
Method engineering	Assuring consistency Provide traceability Detect incompleteness Definition completeness Underlying model Notation Method steps Guidance Formalness	■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■	□ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □	□ □ ■ ■ ■ ■ ■ ■ ■
MuDForM specific	Domain-based Structural and behavioral Multiple domains Natural language As input Translatable back into text	□ ■ □ ■ ■ ■	■ ■ □ ■ ■ ■	□ ■ ■ ■ ■ ■	□ □ ■ □ □ □	□ □ □ □ □ □	□ □ □ □ □ □	■ □ □ □ □ □	□ □ □ □ □ □	□ □ □ □ □ □	□ □ □ □ □ □

Application scope This category considers the context in which the specification technique is applicable, and helps to answer all three RQs. We classify the techniques on:

1. **Domain dependence:** We want to see if there are limitations to the domains to which the technique is applicable. This classification aspect is added to see how well existing techniques serve MuDForM objectives O2 and O3.
Possible values: no specific domain, the name of a specific domain, or characteristics of the targeted domains. Although a technique might not be specific for a domain, we will also extract the domains of the examples in the study.
2. Their suitability for **quality domains**. We want to see if literature exists that shows how to apply domain modeling techniques to the domain of quality,

2.3 Study Design and Execution

and if this requires specific modeling concepts or modeling steps. This serves objective O2. Keep in mind that we are not looking for concepts that enable dealing with quality as a topic in the development process. For example, the distinction between functional requirements and nonfunctional requirements enables to deal with them separately, but just the distinction does not help to specify them differently. The literature might offer solutions for particular quality attributes or other classifications of nonfunctional requirements. These must be considered if they provide a specification technique.

Possible values: any quality, a specific quality domain (*e.g.*, from ISO/IEC25010 [86]), explicitly mentioned characteristics of dealing with quality (*e.g.*, quality attribute scenarios as explained by Bass *et al.* in [15]).

3. Their usefulness in the definition of the **architecture** of a system. How does the technique fit in the context of architecture activities and architecture artifacts? We are specifically interested in how DO specifications are used to create a system in the targeted software environment, *i.e.*, the software technologies and platforms that the system is supposed to operate on and connect to. Of course, this aspect helps to cover MuDForM objective O8. But it also serves O1 and O3.

Possible values: an explicit architecture approach, a specific (possibly partial) match with ISO/IEEE42010, specific matches with architecture elements.

Method engineering We also classify studies on how well their approach or method is described and on how systematic it is. The aspects below are not specific for domain-oriented methods, but are relevant for all specification methods. This classification category serves MuDForM objective O7 and is relevant for answering all three RQs. We classify techniques on:

1. Support for **assuring the consistency** of a DO specification, or between DO specifications. This means, not just testing if a set of specifications is consistent, but defining specifications such that their consistency level is known at any moment, and that it is clear what must be done to achieve consistency. Mechanisms to prevent inconsistency are also contributing to this goal.
Possible values: a specific mechanism to assure consistency (*e.g.*, fully based on a DSL or DM, or detection of elements that are used in several specifications).
2. How well they **provide traceability** from (intermediate) specifications back to the input. It must be possible to trace the decisions that led to a specification.

Chapter 2. State-of-the-Art Specification Techniques

Possible values: on model/document level, on smallest specification element level, an indication of somewhere in between, or a specific mechanism to provide traceability.

3. How well it helps to **detect incompleteness** in the targeted specification, and in the used input. A method should offer guidance in gathering knowledge about the (to be) modeled entity, for example by the use of standard types of questions for the involved (domain) experts, or questions that are an entry point for the analysis of input documents.

Possible values: specific guidelines or steps for detecting and acquiring missing input information.

4. The **definition completeness** of the specification technique. According to Kronlöf [97] a method definition should provide:

- (a) An **underlying model**, *e.g.*, metamodel, core model, or abstract syntax, of the specification technique, which forms the foundation for the semantics of a specification.

Possible values: a specific metamodel, set of concepts of the underlying model, a specific (meta)modeling language.

- (b) An explicit **notation**, possibly used in different viewpoints. All the viewpoints of the method should be defined in terms of the concepts of the underlying model.

Possible values: specific viewpoints, notation descriptions, a specific language (like UML)

- (c) Explicit **method steps** that go through the viewpoints and that have clear entry criteria and exit criteria.

Possible values: list or model of steps, comments about the relation to the viewpoints and/or about the granularity of the steps.

- (d) **Guidance** for taking steps and making specification decisions.

Possible values: (reference to) a set of guidelines. These may be specific for each step or viewpoint.

5. **Formalness**: The degree to which the technique delivers formal specifications, and how it combines formal and informal specifications, *i.e.*, semiformal specifications. A formal language has formal semantics and can potentially be processed in an automated way.

Possible values: not formal, explicit formalism, via formal metamodel, indication of hybridity, via model consistency rules, via unambiguous semantics.

2.3 Study Design and Execution

MuDForM specific The MuDForM objectives defined in Section 2.2.3 could potentially be met by existing specification techniques. We discuss how well the identified studies serve the objectives and classify them on the following aspects:

1. **Domain-based:** The degree to which a specification technique uses a domain specification to define specifications in terms of that domain specification. DMs and DSLs are both considered as domain specifications that can be used to make other specifications. This aspect is added to the framework to serve objective O4 and to answer RQ1.2.
Possible values: specification uses DS as terminology, specification is instance of DS, specific mechanisms to integrate specifications written in the same DSL or DM.
2. The degree to which the specification technique supports the specification of **structural** (static) properties, **behavior** (dynamic) properties, and their relation. Most techniques just cover either structural properties or behavioral properties. So, this classification aspect is particularly interesting when a study actually covers the integration of structural and behavioral properties. This aspect is added to evaluate objective O5.
Possible values: static, dynamic, dynamics of statics, statics of dynamics, dynamics structures, possibly with additionally mentioned specification concepts for those. For example:
 - (a) Static: classes, objects, entities, attributes, class associations, specializations.
 - (b) Dynamic: activities, events, use cases, functions.
 - (c) Dynamics of statics: operations of classes, activities per class, functions of a system.
 - (d) Statics of dynamics: activity parameters, classes per activity, classes per use case, parameters of functions.
 - (e) Dynamics structures: flows, process models, activity diagrams, state transition diagrams, Petri nets.
3. Suitability for working with **multiple domains**. This aspect is added to the framework to address RQ1.3 and serves the evaluation of objective O1.
Possible values: specific mechanisms for dealing with multiple domains. For example, to specify transformation/synchronization/consistency between domains, or to structure domains into new domains, e.g., via extension, merge, composition, or decomposition.

Chapter 2. State-of-the-Art Specification Techniques

4. Support for **natural language**. This aspect is added to serve objective O6.
 - (a) The degree to which they support texts in natural language **as input** for the specification process.
Possible values: specific mechanisms to deal with concepts found in natural language texts. For example:
 - i. Setting context for (domain) terminology, like books or reports in a series, articles, chapters, sections, and paragraphs. These are potential namespaces for the elements in specifications.
 - ii. The processing of grammatical concepts, such as Subject, Noun, Predicate, Possessive case, Preposition, Phrase, Object, Number (amounts, singular, plural), Direct object, Gerund, Indirect object, Case, Collective noun, Comparative, Conjunctive, Infinitive, Imperative mood, Ordering events (in time), Adjectives, Adverbs, Appositive, Modifier, Classification, etc..
 - (b) The degree to which DO specifications are **translatable back into text** in natural language, and how well that text is still consistent with the original input text.
Possible values: specification mechanisms for translation of specification elements into text, indication of the degree to which semantics are lost in the translation.

2.4 Study Results

This section uses the classification framework described in Section 2.3.5 to organize and present our major observations. To this end, we first extracted the data from each of the 53 primary studies through the perspective of each classification aspect. Then, we made a summary per aspect, as reported in Sections 2.4.2–2.4.4. We collected all the extracted data in one spreadsheet, which is part of the replication package [35]. For easy reference, at the end of this chapter we have provided the List of Primary Studies with reference numbers [1] through [53]. Hence, in the following, the first 53 references indicate primary studies. First, we discuss our observation regarding the publication trends.

Table 2.3, which has the same structure as Table 2.2, shows which primary studies address each aspect.

2.4 Study Results

Table 2.3: Primary Studies per Classification Aspect

CLASSIFICATION CATEGORY	CLASSIFICATION ASPECT	Addressed by PRIMARY STUDIES
Application scope	Domain dependence Quality domains Architecture	- [156, 8, 173, 53, 153, 148, 157, 92, 96, 138, 70, 105, 109, 21, 102, 61, 164, 107]
Method engineering	Assuring consistency	[96, 25, 137, 52]
	Provide traceability	[8, 7, 130]
	Detect incompleteness	[133, 134, 96]
	Definition completeness	[141, 3, 27, 63, 133, 135, 134, 8, 163, 7, 173, 130, 73, 4, 64, 157, 92, 25, 41, 109, 98, 24, 137, 62, 164]
	Underlying model	[63, 133, 134, 8, 163, 7, 71, 173, 130, 4, 53, 72, 64, 157, 143, 92, 96, 25, 47, 138, 70, 41, 24, 102, 82, 161, 62, 61, 164, 52]
	Notation	[141, 156, 63, 135, 8, 129, 163, 71, 173, 130, 20, 73, 72, 143, 92, 80, 96, 25, 47, 138, 70, 105, 21, 82, 137, 2, 62]
MuDForM specific	Method steps	[27, 135, 8, 163, 7, 130, 53, 153, 143, 92, 96, 47, 138, 51, 102, 82, 161, 2, 61, 164, 52]
	Guidance	[146, 73, 64, 80, 138, 41, 50, 24, 107]
	Formalness	[133, 135, 134, 73, 157, 96, 107] [134, 53, 64, 143, 96, 47, 138, 161, 2, 107]
	Domain-based Structural and behavioral	[3, 27, 146, 135, 130, 20, 4, 153, 157, 80, 25, 70, 105, 41, 48, 109, 50, 98, 160, 51, 24, 102, 137, 62, 164, 52]
MuDForM specific	Multiple domains	[129, 7, 96, 47, 138, 82, 161, 2]
	Natural language As input Translatable back into text	[129, 96, 161]

2.4.1 Publication Trends

Figure 2.7 shows the total number of included studies per publication type (in the y-axis) and their distribution over time according to their year of publication (in the x-axis). We observe an increase after 2002, with peaks in 2004 and 2009. Studies before 2000 are about DM approaches and not about DSLs. An explanation is suggested by Czech *et al.* [28], because they state that the term DSL did not exist before 2000. However, Kosar *et al.* say that there was a Usenix conference in 1997 on DSLs [95]. Prieto-Díaz states in his 1990 paper [128] that a domain language, preferably formal, is one of the outputs of domain analysis. Nascimento *et al.* [44] say that the idea of DSLs was already published in 1965, but that the term domain or DSL was not used. After 2000, the studies cover the whole DSL development process, in which the creation of a DM is positioned as one of the DSL development phases. Consequently, DM creation receives less attention than before 2000. Though, Chaudhuri *et al.* [21] state in their 2019 paper that there is still not much literature about creating the abstract syntax of a DSL. In the last decade, the included studies' topics have also shifted towards issues related to multiple domains, and to multiple models in general.

Concerning the types of publications, Figure 2.7 shows that most studies are peer-reviewed scientific works (41/53 are conference-, workshop-, or journal papers). Several books (7) and technical reports (5) provided useful information too.

We also looked (in Figure 2.8) at the publication trends with respect to the coverage of the aspects over the years. The Figure emphasizes three clusters around 2004, 2009 and 2015. All are centered around aspects of method completeness and multiple domains; with a growing attention for guidance. We notice that the topics of notation,

Chapter 2. State-of-the-Art Specification Techniques

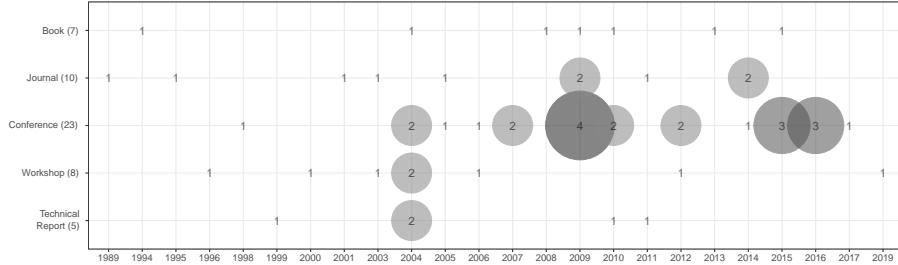


Figure 2.7: Publication Trends – Venues of the Years

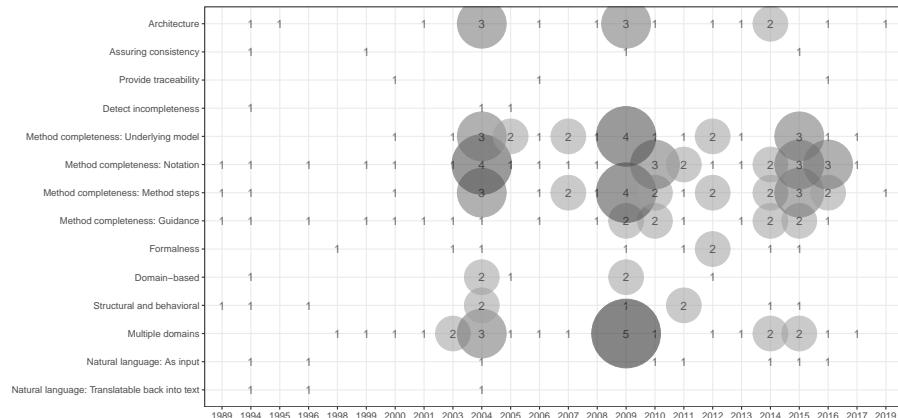


Figure 2.8: Publication Trends – Aspects over the Years

guidance, and multiple domains are addressed throughout the entire time span.

We also looked at the application domains of the examples or case studies in the studies. We found that only six studies [8, 4, 25, 41, 164, 92] have real case studies or examples. None of the studies mentions the business or organization in which the techniques are used. Several studies [72, 80, 70, 161, 129, 153, 148, 50, 160, 82] do not have a demonstrative example of how their specification techniques are used. The rest has either small illustrative examples or a running example throughout the study.

We found that a banking example is used the most (six times) in the 53 studies. But the examples are all slightly different. It might be a good idea having a reference (banking) case description that can be used by over and over in different studies. This would save time in case development and in understanding the application of whatever concept is the topic of research. Examples about processes for reserving, ordering, paying, and delivering products or services are also used regularly. Another category of examples concerns the software domain itself, like the example about components and deployment in [48], or the transformation from Petri Nets to Statecharts in [98] and [102]. The examples in [134, 21, 143] and some examples in [92] are more about, or closely related to, embedded software.

We also looked for correlations between the domains and the other aspects of the classification framework, but we found no significant ones.

2.4.2 Application Scope

Domain dependence. All approaches and methods of the selected studies are domain independent. Though, some studies [148, 25, 146, 157, 143, 21] explicitly limit themselves to the specification of software systems, by seeing a DSL as a system specification language, or as a programming language [61]. All studies aim at specifying the application domain or at the functionality that the system should provide for the application domain.

None of the studies addresses resource domains or quality domains. Though, some have examples covering the resource domain, like the example about component deployment and hardware in [48].

Suitability for quality domains. There is no method that is specifically targeted at the specification of quality. All studies demonstrate their technique via specifications of the application domain, *e.g.*, banking, or the system domain, *e.g.*, components and interfaces. It seems that the explicit specification of quality is simply ignored or avoided. Kelly and Tolvanen [92] mention that domain-specific modeling is leading to higher quality, but they do not explain how or provide specification techniques for quality. We have found some examples of DMs for a specific quality, like the security DM by Firesmith [58], which is used to specify security requirements. However, those examples are excluded, as they do not contribute any specification technique.

Relation to architecture. Most studies do not address the relation to (software) architecture, and none of them deals with the architecture of systems that are specified via

Chapter 2. State-of-the-Art Specification Techniques

multiple DSs. Some studies, *e.g.*, [156], distinguish an explicit step from a DS to its implementation in a targeted software environment, but they do not explain how to specify the transformation, what detailed steps to take, or what guidelines to follow.

Some studies use explicit mechanisms to transform DO specifications into a specification that can be executed in the target (software) environment. We found the following types of mechanisms:

- Some approaches, *e.g.*, [141, 135], are UML based and, as such, can build upon the use of UML for modeling a software design. These approaches use classes as the main modeling concept, and suggest that a system design is based on the modeled class structure.
- Some approaches, *e.g.*, [53], model the application domain and prescribe that the derived software system has elements that correspond with elements of the created DS. Sagar and Abirami [138] describe a specification technique for functional requirements, and they suggest that those requirements correspond with functions of the system. This kind of use of a DS can be seen as an architectural style or design pattern.
- Some approaches, *e.g.*, [96, 92], give examples of transformations from a DS to a target environment, such as a relational database or user interface library. The study of Zhang *et al.* [173] follows the structure of model driven architecture [117], which has a step for the transformation of a platform independent (domain) model to a platform specific model.
- Some approaches, *e.g.*, [157, 21], focus on DSLs that specify parts of a system design. A clear example is the DSL for component and interface specification in [153]. In these cases, the DSL itself can be seen as a design pattern for a software system, because the system structure follows the structure of the DSL.

In general, methods with an underlying (meta)model with clear (formal) semantics, *e.g.*, [62, 63] are easier to embed in an architecture (pattern), because they enable a formal transformation of the DS into software.

2.4.3 Method Engineering

Support for assuring consistency. Some studies, *e.g.*, [25, 92, 105], mention consistency, but do not provide mechanisms to achieve it. Evans [52] provides different

approaches for consistency across domains, which can serve as techniques to make DO integration specifications on top of DSs. Romero *et al.* [137] describe an approach for achieving consistency across viewpoints, which is based on the use of correspondences, similar to the concept of correspondence in the ISO/IEC 42010 standard [85].

The KISS method [96] proactively supports consistency via method steps and guidelines that iterate over the different views on one model. The guidelines prescribe how existing views are used as a starting point for creating a new view, and how changes in one view impact other views, without providing an explicit metamodel.

Traceability (to input). None of the studies addresses traceability explicitly. But studies that provide fine-grained method steps, an underlying (meta)model, or guidelines for modeling decisions, offer support implicitly. Namely, to achieve a traceable process, changes to the specification must be logged, preferably with a rationale. First, if the method provides method steps at the level of specification changes, then these steps can be used as the type of the changes. Second, if a metamodel is given, then changes can be defined in terms of create, update, delete actions of instances of the metamodel. Third, guidelines can serve as rationales for logged specification decisions.

Detect incompleteness. None of the studies explicitly addresses the detection of incompleteness in the used input. The approaches that see a DM as an abstraction of a set of systems, *e.g.*, [134], help in achieving completeness by explicitly checking if all DM elements are used in a specific application model. Some studies prescribe the presence of multiple viewpoints in one model, *e.g.*, [96]. This might result in the detection of missing information in the input of the modeling process, which may lead to requests for extra input from the domain experts, and as such contributes to the completeness of a specification.

Method completeness: underlying model. Most studies use UML or MOF as their underlying model. Some have their own metamodel, like KerMeta [62], or MEMO ML [63, 64]. The major observation here is that most metamodels limit themselves to structural concepts, like classes, attributes, and relations between classes. ORM [129], Normalized Systems [107], and the KISS method [96] offer also behavioral concepts, but do not provide a metamodel.

Method completeness: notation (syntax and viewpoints). In the selected studies, UML is used most often as a graphical notation. Its usage is mostly limited to a class diagram, sometimes extended with OCL for specifying rules on top of the classes.

Chapter 2. State-of-the-Art Specification Techniques

Some studies, *e.g.*, [24, 27, 52, 51], use packages and package diagrams to model relations between domains.

Only graphical DSLs sometimes offer more than one viewpoint. In case of a textual DSL, only one viewpoint is presented. This view is the complete textual specification of a model in terms of the DSL. For example, they do not even distinguish header files and implementation files as different viewpoints.

Most studies show a notation in their examples, or prescribe a step for explicitly defining a notation. Chaudhuri *et al.* [21] do not address notation, because they explicitly restrict themselves to the abstract syntax or metamodel.

Method completeness: method steps. Many studies provide a step for making a DS. The steps of most approaches are coarse grained, and reflect phases or stages in the specification process. The only study that offers fine-grained steps for making a model is from Ibrahim and Ahmad [82]. We did not find any other approach that has steps at the fine-grained level of modeling concepts like class or attribute. The KISS method [96] provides fine-grained steps for grammatical analysis on an input text to come to an initial model. But after grammatical analysis, the model engineering phase is defined by steps at the level of viewpoints. As mentioned before in the observations about traceability, a metamodel offers implicit modeling steps via the creation, update, deletion of modeling concept instances.

Some DSL approaches, *e.g.*, [141, 156, 63, 135, 163, 72, 143, 92, 21], have a step for creating a domain model, metamodel, or abstract syntax. But they do not go into detail on how to do that, or how to derive a DSL from the created model.

Method completeness: guidance. The studies about patterns for DSL design [153], for DSL implementation [61], for DM integration [52], and for model transformations [102] all provide guidelines for choosing between patterns. Remarkable is that Frank *et al.* [63] state that the creation of a domain language is a demanding task, mostly performed by highly specialized experts, and good guidance is currently missing. This contradicts slightly with the elaborate guidelines given by the studies that are about the processing of natural language to make a (domain) specification [2, 7, 82, 138, 96, 47, 161].

None of the studies offers guidelines for all modeling steps or for all prescribed views. This means that the predictability of a process that follows such an approach is low because it strongly depends on the expertise and domain knowledge of the modeler.

Formalness. Most studies do not mention how formal their specification techniques are. The approaches that use UML as their underlying model can be seen as partially formal, depending on the part of UML that is used.

Simos and Anthony [146] and Frank [64] present languages with a formal metamodel. Kelly and Tolvanen [92] state that all DSLs must be formal, so they can be parsed and used for generating code. Of course, all DSs that are used to generate software, must be unambiguously parsable.

Golra *et al.* [71] explicitly choose informal modeling in their approach for developing DSLs. Though, they state that a DSL itself implicitly has formal semantics via its implementation in software.

2.4.4 MuDForM Specific

Domain-based specifications. Most studies that discuss the creation of domain-based specifications, consider an application model to be an instance of a DS. We are not interested in these types of approaches because of the same reasons as given in exclusion criterion E5. None of the studies that see a DS as a language to define other specifications, provides steps and guidance to do so. The KISS method [96] and the Normalized systems approach [107] use the DM to specify functions, processes, or workflows. They both provide examples, but no explicit steps and guidelines are given.

Some approaches [7, 164] show examples of requirements specifications in terms of a DSL, but they do not provide steps and guidelines for creating them.

Integrated structural and behavioral properties. Most approaches only cover structural properties (classes, attributes, relations between classes). Some approaches [134, 53, 64, 143, 47, 138, 2] also provide behavioral concepts of structural elements (operations of classes). Reinhartz-Berger *et al.* [133] focus purely on behavior, via activity diagrams. Some studies [8, 173, 73, 53, 72] offer behavioral concepts and structural concepts, but they do not explain their coherence. These are mostly UML based studies, which use classes for structural properties, and use cases or activities for behavioral properties, but do not explain how to relate them to each other.

Only the KISS method [96] and the Normalized systems approach [107] offer autonomous modeling concepts, method steps, and guidance for specifying structural properties, behavioral properties, and the relationship between them.

Chapter 2. State-of-the-Art Specification Techniques

Suitability for working with multiple domains. We did not find a study that offers methodical support for working with multiple domains. Many studies [153, 105, 3, 27, 130, 20, 4, 80, 70, 41, 48, 109, 50, 98, 160, 51, 24, 62] offer mechanisms for dealing with multiple models and their integration, but those mechanisms are not explicitly merged with the specification technique that was used to create the original models. A method to create a DS or domain-based specification could provide certain model properties which make the models easier to integrate, *e.g.*, being in the third normal form, or being context free. The found studies mostly offer techniques based on relations between packages, such as merge, refine, reference, specialization, assembly, instantiation, and unification.

We did not find studies that use consistency rules or correspondence rules as a technique to specify domain-oriented integrations. All studies either combine two DSs into a new DS, or define transformations from a source DS to a target DS.

Natural language as input. Several studies [96, 138, 2, 161, 82, 47] offer explicit steps and guidelines for transforming natural language text into model elements, but most studies do not. Some studies mention the involvement of domain experts and that their “words” become terms in a model, *e.g.*, [129, 163, 53, 92], but they do not explain how to do this systematically, *i.e.*, with clearly defined method steps and guidelines for eliciting knowledge from domain experts.

Translatable to natural language. Most studies do not address the translation of specification into natural language. The study by Hoppenbrouwers *et al.* [80] offers the paraphrasing of all model parts in natural language. Of course, any specification written in a defined language can be spoken in natural language by simply reading the specification in terms of the specification language literals. Specification languages that have concepts that are close to natural language, like in the KISS method [96] and the Normalized System approach [107], are more suitable, because they offer an easier transition from text to model and back.

2.5 Discussion

In the following we discuss our observations in relation to the three research questions (Sections 2.5.1–2.5.3) followed by additional observations that emerged from the results (Sections 2.5.4–2.5.6).

2.5.1 RQ1.1: No Fully Engineered Method

We did not find any method that was engineered in full, *i.e.*, with a good answer to all the method engineering aspects of the classification framework, or even for just the aspects of method completeness.

Most of the studies do not address consistency. As such, one cannot assume anything about the well-formedness and consistency of a made specification. This could be acceptable when specifications are only used informally. But if specifications are used to create other specifications, then consistency rules, and how well a specification meets them, are important. None of the studies provides explicit guidance for detecting incompleteness of specifications. The effect is that the completeness of a specification depends on either the proactive attitude of the involved (domain) experts to bring in missing information, or that the modeler herself has complementary domain knowledge that allows her to ask the right questions. Method steps could provide guidance for asking the right questions to the involved experts, or for searching the input documents for a specific type of information.

None of the found methods has a complete definition (*i.e.*, covering underlying model, notation, steps, and guidance). Some approaches provide a metamodel and a notation, and others provide high level steps, sometimes guidance, and sometimes the use of an existing language, such as UML. The lack of an underlying model makes it hard to have an unambiguous well-defined interpretation of models, and difficult to separate the semantics from the syntax. If there is a language defined without modeling steps and guidance, then modelers must be very experienced, and the specification process becomes unpredictable. Moreover, it is impossible to create a tool that guides the modeler through the modeling process. And, if such a tool is made, then the tool builder has (implicitly) decided about the steps and guidance, which might result in a suboptimal specification process.

2.5.2 RQ1.2: No Methodical Support for Applying a Created DSL or DM

The approaches in the studies about DO specification techniques, are mostly limited to creating a DS. They typically do not incorporate steps and guidance for applying a created DS. So, none of the studies proposes an approach that prescribes how to use a DS that is created with that approach. In the literature there are publications about the usage of a specific DS, *e.g.*, [58] and the many examples from [43]; these, however,

Chapter 2. State-of-the-Art Specification Techniques

are specific for the DS at hand, and as such are excluded as primary studies, because they do not contribute a specification technique.

We think that an approach for creating a DS should also take into account that the DS is applied properly. We looked separately for literature on the application of a created DS and found some studies on evaluating the usability of a created DSL from Barivšić *et al.* [13, 14, 12]. They state that evaluation of DSLs does not happen often. Gabriel *et al.* [65] state that the community in software language engineering does not systematically report on the experimental validation of the languages it builds. Barišić *et al.* also state that DSL usability testing can be costly, but that poor usability is more costly in the long run. Rodrigues *et al.* [136] conclude in their SLR that there is a lack of systematic approaches to evaluate DSLs, and notice that most authors do not report problems with the language they created. Gray *et al.* [74] write that “poor documentation and training [of a DSL]” is one of the 10 reasons why DSLs are not used more in industry. Völter [165] states that it is important to communicate to users how the DSL works, and that documentation should be example-driven or task-based. He even observes that in nonscientific domains, domain experts are not expected to be the ones who specify systems with the DSL. Instead, the domain experts pairs up with the DSL developer to apply the DSL, or the DSL developer does all the specification based on discussions with the domain expert.

The lack of methodical guidance for applying a created DS is risky. The creators of a DS might know its semantics precisely and also how to use it. But targeted users of the DS probably do not, and should be instructed. Although not specifically mentioned in the found literature, we have seen this many times in industry projects. A DSL is defined, but only its creators understand how to make models with it or understand its exact semantics. The effect is that the DSL is not easy to learn by the targeted users, and that they have to learn it via examples or personal guidance from the creators. Even if such support is present, then the available examples might not exploit all the features of the DSL and the creators might not be available for guidance, or assign different semantics to the DS. The effect is that the DS is not used optimally, and often leads to the perception that investments in domain-specific modeling do not pay off. We think this lack of methodological guidance is one of the most important shortcomings in the literature on domain-oriented methods, which is subscribed by Gray *et al.* [74] and Barivšić *et al.* [12].

The potential advantage of having DS creation and DS usage in the same method is that the method part about usage can use the method engineering properties of a created DS. For example, knowing that a DSL has a context-free grammar makes the

parsing of models in terms of that DSL predictable. Or, if the behavior in a domain is specified via atomic actions that form the only way to change objects, then those actions may form building blocks for specifying the steps in a process model. In the design of MuDFoM, we could benefit from the studies about specifying requirements in terms of a given DS [7, 96, 164, 107].

2.5.3 RQ1.3: No Integral Support for Multiple Domains

In our view, a (software) system specification can be seen as an integration of multiple domain specifications and domain-based specifications. A method to realize this view should guide the creation of those specifications, as well as their integration. We did not find such a method. We found studies that offer a single technique for integrating two specifications, *e.g.*, [62, 24], but there is no study that offers techniques that aim at integrating the specifications of more than two domains. Dynamic mechanisms, such as transformation and runtime weaving, can only work in such a context, if they are based on specifications of consistency between domain specifications and domain-based specifications. Several methods mention aspect weaving as a mechanism for integrating specifications *e.g.*, [146, 41]. Weaving could be used to integrate functionality with other quality attributes, and thus can be seen as one of the options to integrate domains. However, none of the studies explains how to do it. The studies that offer techniques for integrating models in general, and not DMs specifically, *e.g.*, [52, 109] do not benefit from the properties that a well-defined language and method can offer, because they cannot assume anything about the method engineering properties of the models they integrate. For example, if two DMs are normalized in the 3rd normal form, then their integration is easier, and the resulting model is also easier to normalize.

2.5.4 Behavior is Mostly Ignored and Poorly Integrated with Structure

Although all studies are domain independent, they do not offer the modeling concepts for the same aspects of a domain, and vary in how well and how clear those concepts are integrated with each other. Most studies offer concepts for structural properties (classes, attributes, associations, specializations); some are centered around behavior via concepts like processes, functions, or features. This means that the suitability of such methods highly depends on the aspects that are relevant in the modeled domain. If a domain is mainly data-centric, then a method that has mainly concepts

Chapter 2. State-of-the-Art Specification Techniques

for structural properties will probably suffice. Similarly, if a domain is mainly characterized by behavioral aspects, then processes or functions could suffice as the central modeling concept. But if a domain has both structural and behavioral properties, then most methods will not suffice. To our knowledge, only the KISS method [96] offers a language that enables treating both structural and behavioral properties, and offers modeling concepts, steps, and guidelines to relate these properties coherently.

2.5.5 Minimal Interface with Natural Language

Some methods provide guidelines for the transformation of natural language text into a model in a specification language. Because natural language has evolved over thousands of years, it contains concepts and grammar that suit the way humans want to communicate about the world (domain) around them. According to Chomsky [23] and elaborated by Pinker [123], humans have an innate capability to process natural language.

In general, a modeling language and natural language are not isomorphic, leading to differences in their expressiveness. The effect is that the verbalization of model parts does not resemble the input text that was used to make the model. Therefore, the translation of a specification into natural language text suffers from loss of semantics with respect to the original input text. The most noticeable symptom of this shortcoming is the use of entities or classes in the model to represent verbs from the text, like in [138, 47]. In such a model, it is not clear which classes correspond with a noun and which classes correspond with a verb. So, you cannot generate a sentence that expresses the original meaning, in the case that a class was derived from a verb.

We think that a specification language for creating DO specifications should have concepts that are close to human reasoning, observation, and communication. This might be offered partially by natural language. But, more research is needed to decide which natural language concepts can be used and which additional concepts are needed to define the specification language for a method like MuDForM.

2.5.6 The Terminology around Domain Models is not Unified

During our work for this SLR, we discovered the use of different terminologies related to domain modeling. In the context of DSLs, a DM is comparable to the abstract syntax of a DSL. Both can be seen as a graph with domain concepts. In the studies that discuss the design of a DSL, a DM is often mentioned as the starting point

for defining an abstract syntax. However, we did not find any study that offers a completely defined method for the transformation of domain knowledge into an abstract syntax.

We learned that the term *domain analysis* refers to the activity that leads to a DM. As such, publications about domain analysis might contribute more in-depth insights related to RQ1.1.

As already discussed in Section 2.3.1, where we defined RQ1.1, a DM and a domain ontology are comparable. If one creates both for a specific domain, then one ends up with two structures that have a high overlap in the used terms.

To be complete, also the terms *underlying model* (as we use it in our classification framework in Section 2.3.5) and *metamodel* are sometimes used when a DS is used to create another specification, *e.g.*, in code generation. We think it is a good idea to create an overview of all these related terms and their meaning, possibly via a domain model, to be able to understand the existing literature better and to decrease ambiguity. To this aim, we find it valuable to carry out a follow-up study specifically on the literature in domain analysis and ontology engineering.

2.6 Related Work

One of the most cited papers on DSLs is “When and How to Develop Domain-specific Languages” of Mernik *et al.* [112]. This chapter defines 5 phases of DSL development: decision, analysis, design, implementation, and deployment. In this SLR, we specifically focus on analysis and design. The implementation phase is addressed in this SLR via the integration of multiple domains. Namely, we consider the implementation of a DSL or DM if it is realized as a DO integration specification between a source domain and a target domain. We also consider the phase of usage, *i.e.*, to apply a DS in the specification of a system (or parts or aspects of it). We did not find DSL-related literature about this phase.

The study by Nascimento *et al.* [44] classifies DSLs into domains. Their domain classification contains mainly different types of software systems, *e.g.*, control systems, web applications, embedded systems, and parallel computing. This is logical if a DSL is seen as a system specification language, but not if a DSL is seen as a language for making statements about a domain, like we do. Kosar *et al.* [95] also present a systematic mapping study on DSLs. They observe that 59% of their included primary studies

Chapter 2. State-of-the-Art Specification Techniques

mention domain analysis as a step in the creation of a DSL. Of those studies, only 6% used a formal analysis approach. They conclude that domain analysis is mostly informal and incomplete. They mention that one of the reasons for this weakness is that domain analysis is too complex and outside software engineers' capabilities. This might be because that study seems to see a DSL as a software creation language, instead of a domain knowledge capturing technique, or as a specification language in general. We think that domain analysis is not a software engineer's task. With MuDForM, we explicitly aim at making the domain analysis phase a systematic activity, with explicit steps and guidance, and a formal underlying model which makes it more predictable, less complex, and easier to learn.

Czech *et al.* [28] collected best practices for domain-specific modeling. They explicitly state that domain-specific modeling aims at generating code and raising the level of abstraction beyond programming. They distilled 130 best practices from 19 studies. They grouped the best practices into different classes that each indicate an aspect of a domain-specific modeling solution: domain model, language design and concepts, generators, DSL-tooling, metamodel tooling, and practices that concern an entire domain-specific modeling solution. Only 3 best practices are about the domain model. These practices are not about modeling itself, but about the context of a DM. We went through all best practices that are about the language concepts. We observe that they did not find and distill best practices for applying a DSL, nor for dealing with multiple domains, which corresponds with our discussion in Sections 2.5.4 and 2.5.5. Also, this study has a programming-oriented perspective on modeling and DSLs, whereas our perspective is more focused on expressing, formalizing, and applying domain knowledge and inter-domain knowledge, in several types of DO specifications.

Iung *et al.* [87] published a systematic mapping study (SMS) on DSL development tools. We think that SMS is complementary to this SLR, because both investigate literature on DO specifications, but each from a different perspective. There is only one overlap in the classification frameworks, namely with respect to "Notation". Iung *et al.* considers whether there is support for graphical or textual specification, while we consider the actual notation of the specification language itself.

Torres *et al.* [159] published an SLR on cross-domain model consistency checking by model management tools. They use a totally different notion of domain as the one discussed in Section 2.2.1. Their notion of domain relates to the engineering discipline of the modeler, such as electrical, mechanical, or software engineering. Their notion of consistency is not about different related aspects in one or more

models, but about how models from different engineering backgrounds actually fit together in the structure and behavior that they describe. This notion of consistency between models resembles interface matching between components. As such, this type of consistency addresses questions like: Are the parameters the same? Do they have the same types? Does the behavior match? We consider consistency between models as a property that is specified by consistency rules that can address any specification aspect.

2.7 Threats to Validity

This SLR has similar threats to validity as other SLRs that followed the guidelines of Kitchenham [94] and Wohlin [169, 88]. We took a number of actions to make sure we selected a proper set of primary studies and achieved valid results.

To assure that we defined the right search strings, we did an initial scan of the literature that we knew, we made concepts models of the found terminology in that literature, as presented in Sections 2.3.1 and 2.3.2. For feasibility reasons, initially we only searched in the titles of the studies; therefore, we cannot rule out the possibility that we missed studies that report contributions to specification techniques. To mitigate this threat, we applied snowballing, in which we did not pose the limit to publications with the terms domain model or domain-specific language (and derivatives) in the title. Moreover, through the inclusion of relevant books, and the inclusion of studies from the initial informal search, we are confident that we have covered a significant knowledge base.

The exclusion of relevant studies during the selection process is another potential threat to validity. We mitigated this threat by defining clear exclusion criteria and a multistage screening process. When we doubted the inclusion of a study after reading the abstract, introduction, and conclusion, we first included it and then read it in full. We excluded it only when it became clear, during the data extraction, that such a study did not provide relevant information. If not, both authors read the paper and decided together. Though, for classification aspects that are mostly not addressed explicitly in the included studies, like *assuring consistency*, *providing traceability*, or *detecting incompleteness*, we analyzed if a study contributed indirectly, and reported that analysis. Finally, the objectives defined in Section 2.2.3 were used as a yardstick in the discussion in Section 2.5 to come to an analysis that is relevant for the domain-oriented development method that we envision, *i.e.*, MuDForM.

2.8 Conclusions and Future Work

This chapter describes a systematic review of the literature on domain-oriented specification techniques, through the perspective of our vision on system specification. It provides an overview of the state-of-the-art in specification techniques to create domain models and domain-specific languages, and their use in the creation of other specifications. We were interested in studying research questions on specification techniques to: *create specifications of domains (RQ1.1)*, *create specifications in terms of domain specifications (RQ1.2)*, and *create specifications of integrations between domain specifications, and between domain-based specifications (RQ1.3)*. This SLR also identifies shortcomings in the existing literature on domain-oriented specification techniques. We found that no method covers all the method engineering aspects framed in our classification framework. In addition, most studies focus on creating DSs, and none provide guidance on how to actually use the created DS. Similarly, no method for creating DSs has a method part that addresses how to deal with multiple domains. This implies that dealing with the application of the DS, and the integration of multiple DSs, is left to be addressed in a specific development context.

These results provide an important foundation for our future work on MuDForM. As described in the beginning of this chapter and framed in the classification framework used to review the existing literature, we aim to design a method that is well engineered, covers the creation of DMs, the use of DMs to create other specifications, the integration of multiple DMs and domain-based specifications, and that is closely connected to natural language.

Following the results of this SLR, we envisage the following research topics and activities:

- The development of an underlying model, method steps, and guidelines:
 - We think that the starting point can be the KISS method [96], because it is the only method that serves objectives O3, O4, O5, and O6.
 - The method part for DO specification integration has to be designed from scratch, because there is not one candidate study that offers a good foundation. But we think that the UML package concept, combined with integration specification options to serve objectives O1 and O2, is a good starting point. The studies mentioned under “Suitability for working with multiple domains” in Section 2.4.4 offer several options for DO integration specifications.

2.8 Conclusions and Future Work

- We can also use the existing guidelines for processing natural language, mentioned under “Natural language as input” in Section 2.4.4. This serves objective O6.
- We will validate MuDForM in a number of industry cases.
- We think a DO specification method should be aligned with human perception, thinking, and reasoning, which means it should be based on primitives that are close to human cognition. Natural language fulfills this vision to a certain extent, but not completely. We plan to investigate the literature from cognitive sciences to analyze what those cognitive concepts could be, identify which ones do not have a clear natural language counterpart, and extend MuDForM accordingly. For example, Hofstadter and Sander [79] explain and reason that analogies lie at the core of human thinking. We will investigate how this concept can be supported in a specification method.
- As mentioned in Section 2.5.6, we will further investigate the literature on domain analysis to understand if we can use method ingredients from existing methods. If needed, we will start an SLR on this topic, to benefit from existing methods and to justify MuDForM design decisions. We already looked at the approaches from Shlaer and Mellor [143], FODA [90], and ODM [148], which are mentioned as the most common domain engineering methods by van Deursen *et al.* [43]. DSSA [158] is also often mentioned in literature, but it does not offer a specific solution for making specifications, and as such did not qualify as a primary study for this SLR.
- We also mentioned in Section 2.5.6 that it would help researchers and domain engineers if a clear overview of domain-related terminology would be created. This involves at least the terms domain model, ontology, conceptual model, feature model, DSL, abstract syntax, metamodel, and underlying model.

Finally, we have defined a reusable approach for comparing methods in an SLR. First, we clarified our overall (MuDForM) research objectives, and we made models of the method domain and the compared methods’ application domain. After that, we used the objectives and models to define the research questions, the search queries, and the classification framework. Our approach is an extension to the well-established standard for literature reviews described by Kitchenham [94], and can be used by others who want to perform an SLR or method comparison.

3 Engineering MuDForM: a Cognition-based Method Definition

“What is the architect doing?
He is by the riverside
What is he thinking out there?
He is committing egocide
Now isn’t that a strange thing?
Well, to him, it feels just
Oh, we guess a person’s gotta do
What a person feels he must”

The Architect, dEUS, 2008

This chapter is based on sections of the publication  R. Deckers and P. Lago, “Engineering MuDForM: a Cognition-based Method Definition”, Software and Systems Modeling, Springer, 2024, Under submission [39].

Chapter 3. Cognition-based Method Engineering of MuDForM

This chapter addresses RQ2.

Context. Our vision is that people involved in a software development process should be able to capture knowledge and decisions in a way that is close to how they communicate and reason. Therefore, we are investigating a method, called MuDForM, to formalize and integrate knowledge of multiple domains into domain models and into specifications in terms of those domain models. We have previously defined a vision and a list of method objectives (see Section 1.1).

Goal. Establish a consistent method definition that is explicitly based on concepts from cognitive science.

Method. Based on conclusions from our systematic literature review, we made a metamodel of the KISS method for Object Orientation, extended its viewpoints and method steps, and defined modeling guidelines. We evaluated our method in several case studies, identified improvements, and adapted the method definition accordingly. To ensure that MuDForM is based on concepts close to human communication and reasoning, we made a model of cognitive aspects found in literature, which serves as the foundation of the metamodel.

Result. We have identified relevant cognitive aspects and defined method construction patterns, which led to the definition of modeling concepts, viewpoints, method steps, and guidelines. This chapter explains how these elements form the MuDForM definition, shows its application in a case study, and identifies open issues for further method development.

Contents

3.1	Introduction	58
3.1.1	Problem Statement	59
3.1.2	Contribution and Target Audience	60
3.2	Research Methodology	60
3.3	Method Definition Concepts	62
3.4	Cognitive Aspects	63
3.5	Method Definition	68
3.5.1	Construction Patterns	68
3.5.2	Main Model Structure: Domain, Feature, Context	69
3.5.3	Metamodel	71
3.5.4	Method Flow	74
3.5.5	Guidance	75
3.5.6	Viewpoints	77
3.6	Examples from a Case Study	78
3.6.1	Specification Spaces and Dependencies of Printing Jobs	79
3.6.2	The Interaction View of the Domain Job printing	80
3.6.3	Object Lifecycle of Domain Class Job	81
3.7	Related Work	83
3.7.1	Domain-oriented Specification Methods	83
3.7.2	Generic Method Engineering Techniques	84
3.7.3	Method Engineering from a Cognition-based Perspective	85
3.8	Discussion	86
3.8.1	Method Ingredients and their Application	86
3.8.2	Relation between the Cognitive aspects and Method Definition	87
3.9	Conclusion and Future Work	88
3.A	Detailed Explanation of Construction Patterns	89
3.A.1	Named Elements	89
3.A.2	Embedding Formalisms	90
3.A.3	Structures	90
3.B	All Major Modeling Concepts	93
3.C	All Method Steps	98
3.D	All Viewpoints	105

3.1 Introduction

Many specification methods and languages exist for system development, and for software development in particular, *e.g.*, [143, 148, 96, 90, 53], each with their own background and objectives. We envision software development as a process in which the involved people make decisions in their own area of knowledge, *i.e.*, *domain*, and in which those decisions are explicitly integrated, such that it finally results in a machine-readable specification (see also Section 1.1.1). That is why our research focuses on an integral method for creating domain models, for using domain models as a language to create other (domain-based) specifications, and for integrating multiple domain models and domain-based specifications. The method should support both analysis and design activities. To direct the development of a method, called MuDForM (Multi-Domain Formalization Method), that realizes the vision, we have defined a set of method objectives (see Section 1.1.3). Objective O6 states that the specification method should be based on concepts from human cognition. We made a model of so-called cognitive aspects, which is used as the foundation of the method's metamodel.

Our systematic literature review (SLR) from Chapter 2 concluded that the KISS method for object-orientation [96] is a good starting point for the definition of MuDForM. We defined a method engineering approach that started with making a model of the modeling concepts and steps of the KISS method, and have gradually enriched it. Over the years, we applied and validated the method in several case studies [114, 93, 36, 38]. This chapter explains how MuDForM is engineered and presents the resulting method definition, along with fragments from a case study. The full method definition, including metamodel, method steps, guidelines, and viewpoints, is published online [33].

Section 3.1.1 describes the problem statement, and Section 3.1.2 the contribution and target audience. Section 3.2 explains the followed research methodology. Section 3.3 gives an overview of the concepts involved in the method definition. Section 3.4 explains the cognitive aspects on which the modeling concepts are based. Section 3.5 explains the complete method definition. It explains the main patterns in the metamodel, and presents examples of modeling concepts, method steps, guidelines, and viewpoints. Section 3.6 shows the application of the method in a case study. Section 3.7 discusses the results and related work. Section 3.8 discusses the method definition. Section 3.9 presents the conclusions and the topics for future work.

3.1.1 Problem Statement

We have conducted an SLR (see Chapter 2) based on explicit method objectives (see Section 1.1.3). The SLR concluded that most of the specification methods and languages used in software development focus on describing system design or functionality [143, 90, 25, 21]. Even specification languages that can be used for any aspect of the system or its environment, *e.g.*, domain specific languages (DSLs), are sometimes positioned as programming languages [61].

To capture knowledge and decisions that are not tied to software functionality or software design, we investigate the use of concepts from human cognition and human communication as the foundation of a modeling language and method, starting from, but not limited to, natural language. We do not know of any existing specification method or language that is explicitly based on concepts from human cognition. However, we found two publications that suggest it, and that provide a partial illustration.

Siau [144] states that despite the pivotal role of modeling methods, most method designs are just based on common sense and intuition of the method designers. He proposes to use cognitive psychology as a reference discipline for information modeling and method engineering. He discusses the different types of memory and knowledge from the Adaptive Control of Thought theory from Anderson [6, 5], and gives an example of how this can impact and restrict a method design. He does not concretely state how it can be used to define method ingredients. Wand [166] distinguishes ontological elements and metamodel elements, and demonstrates and concludes that they can be related, but should not necessarily be the same. Wand does not state what the proper ontological elements are, nor does he define an approach for finding them. This chapter shows how a cognition-based, domain-oriented method can be defined.

The SLR also concluded that the definition of most domain-oriented specification techniques is incomplete. They just focus on defining a specification language. They lack in the definition of other method ingredients, *i.e.*, method steps, guidelines, and viewpoints. Chapter 4 (based on [36]) and Chapter 5 (based on [38]) present those ingredients for a part of MuDForM. This chapter explains how the full method is defined.

3.1.2 Contribution and Target Audience

This chapter offers four major contributions. First, it provides a framework for defining a method, driven by method objectives and based on cognitive concepts, which can be used by other method developers.

Second, the method distinguishes and combines analysis support via descriptive domain specifications, design support via prescriptive feature specifications, and explicit use of external (context) specifications. This structure can be used by method developers targeting self-contained analysis and design specifications.

Third, this chapter presents an example of a coherent and detailed method definition. It comprises a metamodel of about 65 modeling concepts, 15 viewpoints, 33 modeling steps, and 125 guidelines, which exceeds the detail level of the definitions of other domain-oriented methods. It is an example of what a mature method definition looks like.

Fourth, this chapter, together with the full method definition [33], supports the people who want to apply MuDForM, build tools for it, or develop training and support. Furthermore, this chapter provides the starting point for understanding, and for developing MuDForM further.

3.2 Research Methodology

This section describes the research methodology that led to the method engineering approach and resulting method definition. It focuses on RQ2 of this thesis, introduced in Section 1.2:

How to define a specification method that is explicitly cognition-based?

The MuDForM development project started with a vision (see Section 1.1.1), which is detailed in a set of method objectives (see Section 1.1.3). They formed the starting point of an SLR on domain-oriented specification techniques (see Chapter 2). Then, we defined a first version of the method, which was applied to several cases and adjusted based on the findings in those cases. In parallel, we studied literature to identify relevant cognitive concepts, and connected them to the method definition. The approach can be categorized as action research according to the description by Petersen *et al.* [121], resulting in the phases of Diagnosis, Action Planning, Action Taking, Evaluation, and Specifying Learning, which are explained below.

3.2 Research Methodology

Diagnosis. The SLR from Chapter 2 revealed that processing natural language to create models, dealing with multiple domain models, and using domain models and DSLs to make other specifications, are hardly addressed in existing literature. The SLR also concluded that most methods are not well-engineered, *e.g.*, they do not provide a coherent specification of modeling concepts, method steps and guidelines, and notation.

To identify cognitive concepts that could be relevant for the method definition, we should study literature in the fields of philosophy, linguistics, and cognitive science.

Action planning. The mentioned SLR concluded that the KISS method [96] comes closest to realizing the method objectives. Therefore, it is used it as the starting point for the MuDForM definition. MuDForM intends to use the following main characteristics of the KISS method: 1) the separation between domain and feature, 2) having activities as first class citizens, including the relation between activities and classes, 3) the use of natural language processing in model creation and validation, and 4) the modeling of lifecycles of domain classes and functions in a process algebra style. These characteristics respectively serve method objectives O4, O5, O6, and O7 (see Section 1.1.3).

The first step was to consolidate our experience in an initial method definition, consisting of an explicit metamodel, a detailed method flow, and guidelines, which are all lacking in the existing literature about the KISS method. After that, we foresaw two tracks: 1) incorporating method ingredients from other methods, and 2) applying the method in practice via case studies to validate and improve the method definition.

To find relevant literature on philosophy, linguistics, and cognitive science, we started reading literature on language philosophy and linguistics from Noam Chomsky [23], Steven Pinker [123], John McWhorter [111], and John Searle [139]. Chapter 6 describes the study on this literature in detail.

Action taking. We defined initial versions of the metamodel, method flow, and guidelines. The metamodel and method flow are foremost the result of our 25 years of experience in creating, using, and maintaining models of various domains and systems, in the context of business analysis, requirements engineering, functional design, software architecture, process modeling, and test specification. We have started to record and consolidate our experiences, since we started the MuDForM research in 2015. We actively manage the metamodel, steps, guidelines, in a UML model [120] with the tool Enterprise Architect [152]. A document version and a web version of that model are available online [33].

Meanwhile, we contacted industry partners for doing case studies. Together with them, we agreed on the case's objectives, the timeline, and availability of people and documentation.

The study of literature on philosophy and cognitive sciences led to an intermediate document titled Cognition Literature Analysis [34] and Appendix A, in which we discuss how it impacts MuDForM. We made a model of the identified concepts, called cognitive aspects, that we consider to be relevant for the foundation of the MuDForM metamodel.

Evaluation. Together with experts from industry, we applied MuDForM in several case studies [114, 93, 36, 38]. We recorded examples of major analysis and modeling decisions, which led to new and revised guidelines. During the case studies, we also refined the method steps and their descriptions and came up with new viewpoints. After the modeling process, the recorded model was presented and explained to the employees of the industrial partner, often by verbalizing the model in natural language.

Specifying learning. After completing the case studies, we first consolidated the method changes, *i.e.*, made adjustments to the metamodel, method steps, and guidelines. We sometimes also received suggestions for guidelines from peers and, after acknowledging their validity and usefulness, fit them into the method definition.

To be clear, this chapter does not report on the details of the activities mentioned above. It reports the method definition that was used and adapted during the case studies.

3.3 Method Definition Concepts

This section explains the concepts involved in the method definition, as presented in Figure 3.1. According to Kronlöf [97] a method definition should contain the following ingredients:

- An underlying model, *e.g.*, metamodel, core model, or abstract syntax, which defines the modeling concepts, *i.e.*, the elements of the modeling language, and the semantics of a specification.
- Explicit method steps that guide the modeler, from eliciting knowledge from documents and experts, to making a consistent and complete model. Each step involves several modeling concepts and one or more viewpoints.

3.4 Cognitive Aspects

- Guidance for taking steps and making specification decisions during the steps.
- An explicit notation, possibly used in different viewpoints. All the viewpoints of the method should be defined in terms of the concepts of the underlying model.

We extended these ingredients to have a framework that serves the development of MuDForM. First, each ingredient supports a number of the method objectives (see Section 1.1.3) that the method definition aims to realize. Second, to make the modeling concept based on concepts from human cognition, we introduce the notion of cognitive aspect, which will be explained in Section 3.4. Third, we define several patterns that are used to ensure that MuDForM models are uniformly structured. Furthermore, we use the notion of specialization from UML in the definition of the modeling concepts. Consequently, modeling concepts can be based on cognitive aspects, are formed by applying a construction pattern, and can have parent classes.

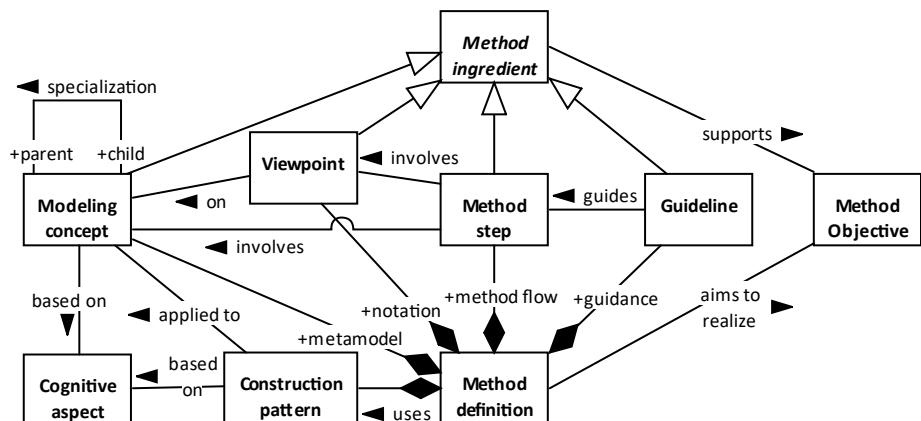


Figure 3.1: Overview of method definition concepts (UML class diagram)

3.4 Cognitive Aspects

This section explains the cognitive aspects that form the foundation for the MuDForM modeling concepts. They represent concepts that the human mind uses to think and communicate, and that we think should be supported by modeling concepts.

Chapter 3. Cognition-based Method Engineering of MuDForM

The diagram in Figure 3.2 presents cognitive aspects that we identified during the literature study described in Chapter 6. The diagram presents an initial set. We do not claim it is complete, but it underpins the scope of the current metamodel.

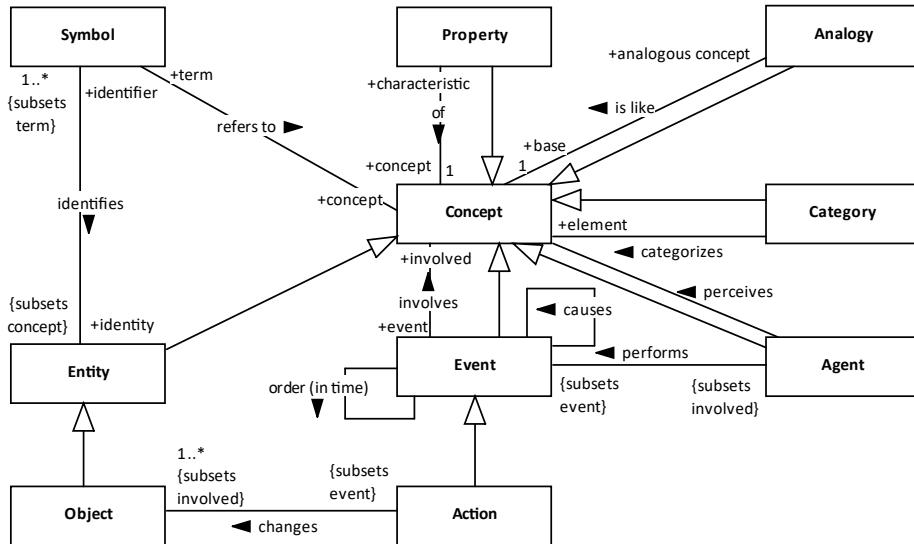


Figure 3.2: Cognitive aspects (UML class diagram)

We have used a UML class diagram to show the coherence between the cognitive aspects. As it is not a very familiar UML notation, we mention that the role names on the association ends are used in the subsets qualifications. For example, the association `identifies`¹ can be seen as a specialization of `refers to`, and the association `changes` can be seen as a specialization of `involves`. The following explains all the cognitive aspects in the diagram.

The root of the diagram is the class `Concept`, which stands for something that the human mind can give meaning to. Examples: “my car”, “the Eiffel Tower”, and “the fear of spiders”. It is the root of most other cognitive aspects; the most fundamental class in the metamodel. `Concept` does not stand in isolation, because a specification requires at least a `Symbol` to refer to a `Concept`, similar to the language triangle of Chomsky [23] and the notion of Mentalese mentioned by Pinker [123, 125]. The

¹From here on, the typescript font is used to indicate cognitive aspects.

3.4 Cognitive Aspects

subclasses of Concept are non-disjoint, *e.g.*, something that is a Property can also be an Entity.

We look for cognitive aspects in natural language, because it has evolved over a long period, and is used for many things that the human mind wants to communicate. We are not limited to natural language, because people can think and reason without it. Some other approaches speak of “modeling reality”. MuDFoM does not follow this perspective, because it requires some philosophical assumption about what reality is. MuDFoM just focuses on what the human mind perceives and utters.

A Category is a Concept that categorizes other Concepts as elements of the Category. Examples: “cars”, “things that fit in my car”, “animals that I fear”. Categories are ubiquitous in all specification languages. Operations, functions, attributes, classes, and data types are all categories. Typically, nouns and verbs are Symbols in natural language that refer to Categories. For example, “chair” refers to a set of things that we consider a chair. Categories are not limited to things that can be referred to by a single word. For example, the category “things in my refrigerator” does not have a designated English word for it.

An Entity is a Concept identified by at least one Symbol. Entities are not dependent on any Category. So, Entities exist without being explicitly categorized by a Category. The aspect Entity brings the possibility to distinguish things in our perception as separate Concepts and that we can refer to those things via their identifying Symbol (name, number, picture of it, pointing to it, ...). Example: “my car” is identified by “its chassis number”. Or, in the context of my possessions and the knowledge that I possess only one car, I can use the Symbol “my car” to identify my car.

An Event is something that happens (in time). Events can be indivisible in time, *i.e.*, they are atomic, or they can have a distinct start and stop moment. An Event may involve zero or more Concepts. Examples: “I am driving my car”, “I see my car on the driveway”, and “It is raining”. An Event may have some order (in time) in relation to other Events, like sequentially, or simultaneously, or one Event happens during another Event. Examples: “I see my neighbor, while I am driving”, and, “I put my seatbelt on, before I start my car.”

An Event may cause other Events. Perceiving causality is very common for people (and other animals). An Event can be performed by an Agent, but that is not always the case. For example, when the Event “I knock the glass off the table” causes the Event “the glass falls on the floor”, who would be the Agent of the second Event? It

Chapter 3. Cognition-based Method Engineering of MuDFoRM

is weird to state that the first Event is the Agent of the second Event. However, there is clearly a relation between causes and performs. There is also a relation between causes and order, because an Event can not be caused by an Event that happens later.

An Object is an Entity that is changed by Actions. An Object exists over a period of time; therefore we can speak of its state. Examples: “I start my car, and now the engine is running.” Or, “I step into my car, which decreases the number of free seats by one.” One can say that an Object is alive when it can still be changed by Actions, and dead when it cannot be changed anymore. A dead Object can still be involved in Events, but such an Event is not changing the state of the Object.

An Action is an Event that changes one or more Objects. Changes is a specialization of involves. An Action may also involve Concepts that are not an Object. Examples: “I start my car with the key”, in which the car changes state, but the key itself does not. Or, “my car accelerates to 100 km/hour”, in which my car changes state, but “100 km/hour” does not.

A Property is a Concept that is a characteristic of another Concept. In other words, a Property is a Concept that belongs to, is part of, or is used to define another Concept. Examples: “cars have wheels”, “my car is blue”, and “cars are vehicles”. Typically, properties are used to express composition or hierarchy, or to determine time, location, or possession.

Property can also be seen as the aspect that reflects coherency of matter, *i.e.*, state. For example, if you step into your car, and you get out after a while, then you expect that the wheels are still on the same location and that the color has not changed. The car wheels and the car color are both Properties of the car Concept.

An Agent is a Concept that performs Events and perceives Concepts. The associations performs and perceives are both a specialization of involves, which is expressed in the diagram via the subsets annotations. Examples: “I start my car”, “I see (perceive) that my car is blue”.

An Analogy expresses that a Concept is like another Concept, *i.e.*, the base Concept. The analogous Concept has Properties that are like the Properties of the base Concept. Example: “a tree with branches is like an animal with limbs”. “Tree” is the analogous Concept, and “animal” is the base Concept. “Branch” is the analogous Property and “limb” is the base Property.

3.4 Cognitive Aspects

A **Symbol** is something that can be perceived and uttered by the human mind, and refers to a **Concept**. Symbols can have different forms, such as a word, an icon, a sign, a photo, a sound, etc. In modeling, a word or phrase is the most common Symbol form, combined with a Symbol that refers to a modeling concept like a function, class, or activity. Notice that a Symbol is not a subclass of Concept, because it does not have meaning by itself. It only refers to a Concept. When a Symbol identifies an Entity, it is called an **identifier**.

Symbols are used to make **Statements** in a **Statement context**, which is depicted in Figure 3.3. A Statement is a Concept that contains Symbols that refer to the Concepts that the Statement is about. The Statements are written in a language, which is part of the Statement's context. The grammar of the language, in combination with the Symbols, makes a Statement interpretable.

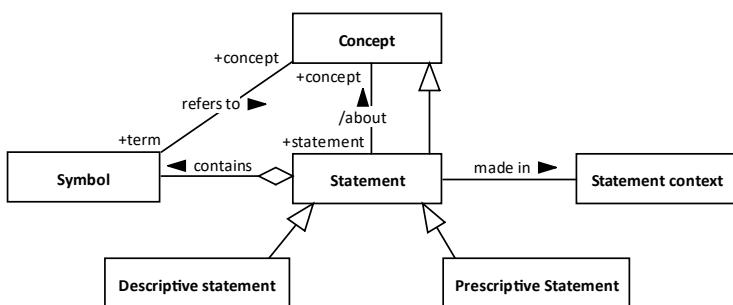


Figure 3.3: Statements (UML class diagram)

To support analysis and design, MuDForM distinguishes **Descriptive statements** from **Prescriptive statements**. A **Descriptive statement** is about Concepts as they are (perceived). They are used for analysis, *i.e.*, understanding something. A **Prescriptive statement** states what should be true for the Concepts it is about. They are used for design, *i.e.*, they state something that should be realized. The distinction is not driven by a cognitive notion, but by the need to have a specification language for developing systems, which involves understanding a system's context through an analysis, and describing the system through a design. For example, "*a car can be blue or red*" is a **Descriptive statement**, and "*my car must be blue*" is a **Prescriptive statement**.

This section listed the cognitive aspects, on which the current version of MuDForM is based. The next section explains the method definition.

3.5 Method Definition

This section explains the MuDForM method definition via the concepts in Figure 3.1. Section 3.5.1 explains the patterns used in the metamodel, and what their relation is with the cognitive aspects. Section 3.5.2 explains the main structure of a MuDForM model, and how it relates to the cognitive aspects around Statement in Figure 3.3. Sections 3.5.3-3.5.6 explain the definition of the modeling concepts, the method steps, the guidelines, and the viewpoints. For the sake of space, we will only present a subset of the method ingredients to clarify how the ingredients' definitions look like, and how they are related. The description of all method ingredients can be found in the complete method definition [33].

As the method definition is sometimes quite abstract and maybe difficult to understand, we invite the reader to look already at the examples in Section 3.6, in which the explained ingredients are applied.

3.5.1 Construction Patterns

This section introduces three patterns that are used throughout the definition of the MuDForM modeling concepts. The purpose of the patterns is to create a uniformly structured metamodel.

The first pattern is for naming model elements. To make models usable for people, model elements must have a humanly understandable symbol, *e.g.*, a name, to identify them. This pattern is not new. We mention it, because we use it to cover the cognitive aspect **Symbol1**. It is explained in Appendix 3.A.1.

The second pattern is for applying other formalisms or theories in models without making an explicit metamodel of them. This can be general formalisms like predicate logic and calculus, as well as specific formalisms like the Object Constraint Language [118]. This pattern allows choosing appropriate formalisms for a specific modeling context. The pattern itself is very straightforward. We present it here, because it is used to cover the cognitive aspect **Statement context**. Namely, an included formalism forms a **Statement context** for **Statements** that use **Concepts**, *i.e.*, operations, and value types, from the formalisms. The pattern is explained in more detail in Appendix 3.A.2.

The third pattern, or rather, set of patterns, is used for defining nested and coordinated relations between modeling concepts. The **Structure** pattern is derived from

the so-called coordinators in the KISS method language. But the KISS documentation [96, 80, 161] does not clarify how the coordinators are related, nor does it provide a metamodel. We use the structure patterns to cover hierarchy between the cognitive aspects. Hierarchy is present when the meaning of one aspect is based on the meaning of another aspect, as captured by the cognitive aspect *Property*. The relation *involves* can also be seen as a form of hierarchy, *i.e.*, an *Event* is defined via the *involved Concepts*. The derived patterns *Coordinated structure*, *Static structure*, and *Flow structure* are defined as template classes, which have parameters that are bound when they are applied in the definition of a modeling concept. Appendix 3.A.3 explains the structure patterns in detail. Their application in the metamodel is demonstrated in Section 3.5.3, and Section 3.6 shows their use in a model.

3.5.2 Main Model Structure: Domain, Feature, Context

MuDFoM uses the concept of specification spaces to organize and manage models, similar to packages in UML. MuDFoM distinguishes three types of spaces: domains, features, and contexts. They are explained below.

We found two different notions of **domain** in literature. The notion that we use is that of a statement space, analogous to a domain in the mathematical sense. The term **domain** refers to an area of knowledge or activity, and a **domain model** describes what can happen (behavior) and what can exist (state) in a domain; in other words, what can be controlled and managed in a domain. A **domain model** is not intended to express what should happen, does happen, is likely to happen, or has always happened in the domain. In general, approaches for DSLs, *e.g.*, [3, 61, 92], comply with this notion of domain. A **domain model** is the foundation for a shared lexicon between stakeholders, and can serve directly as a structured vocabulary for making other specifications, or form the underlying model of a DSL. For example, classes in a banking domain model, such as *Customer* and *Bank account*, can be used in other specifications to restrict or control their instances, or can serve as the foundation of a DSL's abstract syntax. Instances of the modeling concept *Domain*² form a **Statement context** for **Descriptive Statements**, which clarifies how *Domain* is related to the cognitive aspects.

²From here on, method ingredients, *i.e.*, modeling concepts, method steps, guidelines, and viewpoints, are written in *italics sans serif font*.

Chapter 3. Cognition-based Method Engineering of MuDForM

The other notion of domain in literature is that it is a collection of related systems, often forming a product line or family. According to this notion, a domain model defines a set of (system) features that are common in the domain of interest. This notion is used for example by FODA [90], ODM [148, 147] follows this notion too, but speaks of descriptive and prescriptive aspects of a domain. MuDForM puts these aspects in a different specification space, because they typically have a different evolution. Domains (the descriptive aspect) are more stable than features (the prescriptive aspect).

Lee *et al.* [103] identified different meanings in literature of the term **feature**. In the ODM approach [147], feature is defined as a distinguishable characteristic of a “concept”, *e.g.*, artifact, area of knowledge, system, that is relevant to some stakeholders. MuDForM follows this definition and additionally covers the internal specification of features. Consequently, a MuDForM feature model specifies desired characteristics, *i.e.*, describes what must happen (behavior) and what must exist (state). Instances of the modeling concept *Feature* form a **Statement context** for **Prescriptive statements**.

Domain model elements can be used as the types of elements in a feature model. For example, a feature model prescribes how the traffic lights at a crossing must be positioned above driving lanes, and how they must behave, which could be used as a requirements specification of a traffic light control system. In this case, the domain model describes that you can turn the lamps in a traffic light on and off, that you can hang lights above driving lanes, and that a crossing can have several driving lanes. As such, the domain model serves as a specification space for the feature model.

Our notion of feature model differs from some other definitions in literature, where a feature model shows the relation between several features of a product line [90]. Features often correspond with high-level functions of a system, and as such a feature model can be seen as a functional decomposition, as explained in Chapter 5.

A model mostly requires elements that get their meaning outside the domain or feature of interest, and hence, should not be defined in the model of such domain or feature. MuDForM offers the notion of **context**. A context model captures assumptions and knowledge about elements that are needed to specify domains and features, but that exist outside those domains and features. By declaring the contexts on which a feature or domain depends, models have no implicit semantics. Context models can be seen as descriptive, because they define which concepts can be used from outside a domain or feature. They can also be seen as prescriptive, because they define what is assumed to exist outside a domain or feature.

Domain models, feature models, context models, and the relationships between them, make up the main structure of a MuDForM model. For example, for a toll-road payment system, the **domain model describes** that vehicles can arrive and leave at a toll booth, someone can open and close the gate, and someone can pay a fee for a passing vehicle. The **feature model prescribes** that when a vehicle arrives at a toll booth, someone pays for that vehicle, someone opens the gate, and the vehicle leaves the toll booth. This assures that there will be no open payments, at the expense of blocking gates when someone cannot pay. The **context model defines** external concepts, like payment service, fee, and license plate, to the extent needed to specify the elements in the feature and domain model.

The domain model in the example is relatively stable because it describes properties that always hold. Feature models change more often, because desired behavior changes over time. For example, a change could be that someone must pay after the vehicle leaves the toll booth, in favor of traffic flow, but with the risk of unpaid fees. Consequently, the feature model is adapted, but the domain model remains the same.

3.5.3 Metamodel

This section explains the MuDForM metamodel. For the sake of space, we present here two metamodel fragments: the modeling concepts *Specification element* and *Specification space*, depicted in Figure 3.4, and the modeling concepts *Domain activity* and *Domain class*, depicted in Figure 3.5. Table 3.1 explains the definition of some modeling concepts from the two diagrams. For each metamodel class, it shows its parent classes, the applied construction patterns, and the cognitive aspects on which it is based. Appendix 3.B presents a similar table with the definition of all major modeling concepts. Chapter 6 explains in detail how the metamodel elements are based on the identified cognitive aspects.

The diagram in Figure 3.4 is an example of how the complete metamodel is defined. First, there are three types of *Specification spaces*: *Domain*, *Feature*, and *Context*. Second, a *MuDForM model* has a *Specification declarations* structure, which only contains *Specification spaces*. It is not a subclass of *Specification space*, because it is the root of a complete model and does not have *Specification space dependencies*. The complete metamodel is available online [33].

Chapter 3. Cognition-based Method Engineering of MuDForM

Table 3.1: Example of modeling concepts construction based on cognitive aspects

Meta-model class	Parent classes	Applied construction patterns (Pattern (elements) in this font)	Based on cognitive aspects (Aspect in this font)
<i>Specification space</i>	<i>Specification element</i>	1. has <i>Specifications declarations</i> , which is a <i>Coordinated structure</i> where: <i>element</i> → <i>Specification element</i> . 2. has <i>Specification space dependencies</i> , which is a <i>Coordinated structure</i> where: <i>reference</i> →' <i>Specification space dependency</i> , <i>referred item</i> → <i>Specification space</i> .	The <i>Specifications declarations</i> structure realizes the name space aspect of <i>Statement context</i> . The <i>Specification space dependencies</i> express which other <i>Specification spaces</i> form the <i>Statement context</i> of <i>Specification elements</i> in the <i>Specification space</i> .
<i>Domain class</i>	<i>Domain element, Class</i>	1. has an <i>Object lifecycle</i> , which is a <i>Flow structure</i> where: <i>referred item</i> → <i>Involvement</i> 2. has <i>Object life dependencies</i> , which is a <i>Static structure</i> where: <i>reference</i> → <i>Object life dependency</i> , <i>referred item</i> → <i>Class</i>	Is used to model <i>Objects</i> . The <i>Object lifecycle</i> expresses the <i>order relation</i> . The <i>Object life dependencies</i> express <i>hierarchy Properties</i> between <i>Domain classes</i> .
<i>Domain activity</i>	<i>Domain element, Activity</i>	1. has a <i>Role structure</i> , which is a <i>Static structure</i> where: <i>element</i> → <i>Activity role</i> , <i>reference</i> → <i>Involvement</i> , <i>referred item</i> → <i>Domain class</i> . <i>Activity role</i> is a <i>Named relation</i> . 2. has an <i>Activity flow</i> , which is a <i>Flow structure</i> , where: <i>flow step</i> → <i>Activity operation</i> , <i>reference</i> → <i>Operation invocation</i> , <i>referred item</i> → <i>Operation</i>	Is used to model <i>Actions</i> . The <i>Role structure</i> expresses the <i>involves relation</i> . The <i>Activity flow</i> expresses the <i>order of the Operations</i> that are a <i>Property</i> of the <i>Domain activity</i> .

3.5 Method Definition

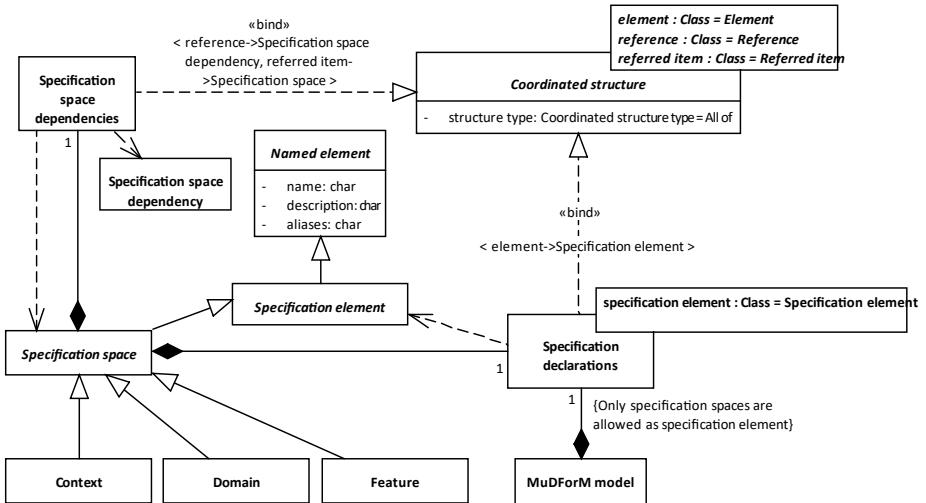


Figure 3.4: The metamodel regarding specification spaces (UML class diagram)

Figure 3.5 presents the modeling concepts for the relation between domain activities and domain classes.

Each *Domain activity* has a *Role structure*, which is a *Static structure*, where the *static elements* are *Activity roles*, the *references* are *Involvements*, and the *referred items* are *Domain classes*. An *Involvement* is the connection of a *Domain class* to an *Activity role*, which means that an instance of the *Domain class* may participate in an instance of the *Activity role* of an instance of its *Domain activity*.

A *Domain class* has an *Object lifecycle*. An *Object lifecycle* is a *Flow structure*, in which the *referred Items* are *Involvements*. It specifies the possible life of instances of a *Domain class*. A *Flow step* in the *Object lifecycle* specifies when the *Domain class* instance may participate in an instance of the *Activity role* of the corresponding *Involvement*.

Chapter 3. Cognition-based Method Engineering of MuDForM

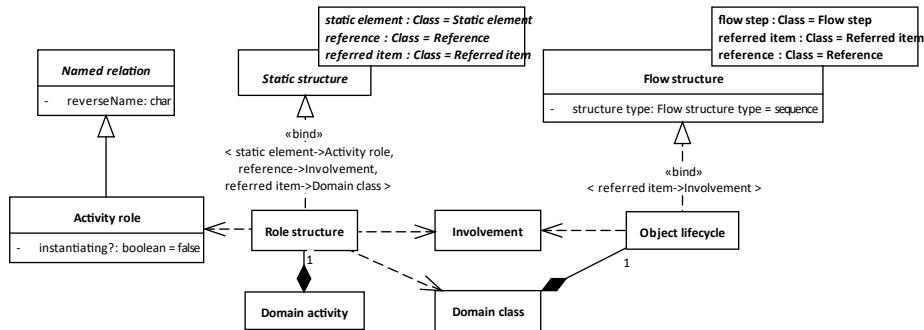


Figure 3.5: Metamodel fragment for model fragment of the case study (UML class diagram)

3.5.4 Method Flow

The MuDForM method flow is, just like the metamodel, derived from the KISS method. Figure 3.6 shows the main steps:

1. **Scoping** defines the purpose, boundaries, and input text. This step does not explicitly exist in the KISS method. The goal of scoping is to have relevant input for the modeling process, in order to (i) prevent unnecessary modeling work, (ii) detect other relevant input, and (iii) keep the model and the modeling process manageable.
2. **Grammatical analysis** elicits and structures knowledge from the input text. We have extended the corresponding KISS method step with support for more grammatical constructs, and with more guidelines. The goal of this step is to maximize the knowledge elicitation from the source, and to make the resulting model traceable to the input. Chapter 4 explains this step in more detail.
3. **Text-to-model transformation** creates an initial model from the structured text. The goal is to move from natural language analysis to modeling. This step is also detailed in Chapter 4.
4. **Model engineering** ensures that all model aspects are covered and that inconsistencies are solved. We have extended the KISS method with steps related to *Function signatures*, *Specification space dependencies*, *Domain invariants*,

Class relations, Function steps, and Behavior decomposition. The steps pertaining to feature modeling are explained in detail in Chapter 5. Furthermore, we have defined about 125 guidelines, which are described in the full method definition [33].

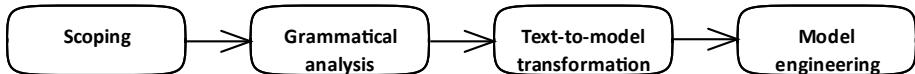


Figure 3.6: Main steps in the method flow (UML activity diagram)

Appendix 3.C describes all the steps in the MuDForM method flow. Each step has a textual description, which is written in terms of the modeling concepts as much as possible. Each step has guidelines, and possibly preconditions and postconditions. Guidelines help to achieve the postconditions, or help the modeler to perform the step in practice, *i.e.*, in a work process with involved (domain) experts. The next section presents examples of steps and guidelines.

3.5.5 Guidance

For each method step, guidelines are defined, which can have different origins:

- The KISS method already offers some guidelines, which were described, mostly implicitly, in several publications [96, 161, 80].
- Some of the authors' knowledge, gathered during 25 years of modeling, is captured in guidelines.
- Our SLR (see Chapter 2) identified studies with potential guidelines. We have added those when applicable. To do so, we sometimes had to adjust the formulation of such guideline to match the MuDForM terminology.
- During the case studies, we have formulated analysis and modeling decisions, of which some have been rewritten as guidelines, because we think they are generically applicable.

Here, we present three method steps, which are used in the examples of Section 3.6. For each step, we present some of its postconditions. Previous MuDForM publications present several of the (currently) 125 guidelines [36, 38]. The full method definition [33] describes all guidelines.

Chapter 3. Cognition-based Method Engineering of MuDForM

For the sake of practical usability, we have chosen to use natural language for the specification of the guidelines, which are expressed in terms of the metamodel as much as possible. The descriptions below require knowledge of the metamodel, simply because steps and guidelines pertain to modeling concepts. The required knowledge is provided by Sections 3.5.1, 3.5.3, and Appendix 3.B.

Step **Manage specification space dependencies:** Define a *Specification space dependency* from *Specification space P* to *Specification space Q*, if P contains *Specification elements* that refer to *Specification elements* of Q. The purpose of this step is to guard the use of *Specification elements* across *Specification spaces* (see Figure 3.4).

- **Postcondition:** *Allowed dependencies*

A *Specification space* may depend on other *Specification spaces* with the following constraints: 1) *Features* may depend on *Features*, *Domains*, and *Contexts*. 2) *Domains* may depend on *Domains*, and *Contexts*. 3) *Contexts* have no dependencies; they form the bridge between a MuDForM model and external specifications.

- **Postcondition:** *No cyclic dependencies*

The graph of all *Specification space dependencies* in a *MuDForM model* may not contain cycles. If a cycle appears, it is most likely that a *Specification space* must be split, because it contains parts that differ in abstraction level or aggregation level.

Step **Specify interaction view:** Define the *Domain activities* and their *Activity roles*, and the *Involvements* of *Domain classes* in each *Activity role*. Indicate if an *Activity role* is instantiating a domain object, i.e., instance of a *Domain class*. Define the *Generalizations* between *Domain activities* (see Figure 3.5).

- **Postcondition:** *Domain classes must have Domain activities*

Each *Domain class* must have an *Involvement* in at least one *Activity role*. Otherwise, it is not a *Domain class*.

- **Postcondition:** *Objects from a domain must be instantiated in that domain*

Each *Domain class* must have at least one *Involvement* in an *instantiating Activity role* of a *Domain activity* that is declared in the same *Domain* as the *Domain class*.

Step **Specify object lifecycle (OLC):** For each *Domain class*, define the order in which an instance of the *Domain class* can participate in instances of the related Do-

main activities, i.e., the *Involvements* of the *Domain class* in *Activity roles* are placed in the *OLC* as a *Flow step*, and connected to the other *Flow steps* (see Figure 3.5). As the *OLC* is a *Flow structure*, it is possible to use sub-*Flows* for the coordination of *Flow steps*, i.e., a *Selection* between multiple *Flow steps*, or to state that multiple *Flow steps* are executed in *Parallel*. It is possible to specify *how many times* each *Flow step* can be executed. (The default value of *how many times* equals “One”.)

While making an *OLC*, it is possible that new *Domain activities* are discovered, which impacts other method steps in the main the step *Engineer domain*.

- **Postcondition:** *An OLC has steps for each involvement*
If the *OLC* of a *Domain class* is made, then each of its *Involvements* is referred to at least once by a *Flow step* in the *OLC*.
- **Postcondition:** *An OLC describes a life*
Each *OLC* must have at least two *sequential Flow steps*. Otherwise, the objects of that *Domain class* do not have a life, i.e., do not have a changeable state.
- **Postcondition:** *The first step is instantiating*
A *Domain class* must have at least one *Involvement* in an *instantiating Activity role*. That *Involvement* must be the parent of the first *Flow step* in the *OLC*. In the case there are more *instantiating Involvements*, the *OLC* starts with a *Selection* between those *Involvements*.

3.5.6 Viewpoints

The definition of the viewpoints goes hand in hand with the definition of the method steps. Namely, a step is typically centered around one viewpoint, or is about the transition from one viewpoint to another viewpoint.

Figure 3.7 presents the pattern for creating Views that correspond to a Viewpoint. The pattern can be seen as an extension of the view and viewpoint concept in the ISO/IEEE 42010 standard [85]. A MuDForM Viewpoint uses one or more Modeling concepts as elements of the Viewpoint. Each Viewpoint has one Modeling concept as its root. Accordingly, a View on a root Model element presents view elements. The derivation attribute of the Viewpoint, explains which Model elements are visible in the View.

Table 3.2 contains the viewpoint definitions that are used in the examples of Sections 3.6.1-3.6.3. Section 3.D contains the definitions of all the MuDForM viewpoints.

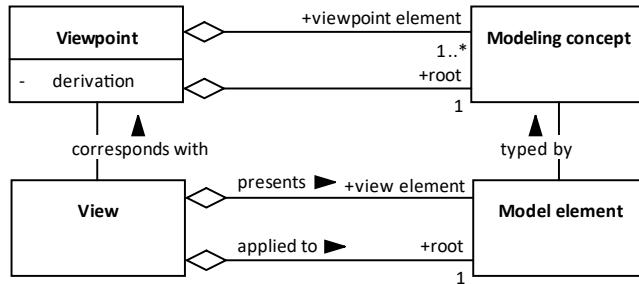


Figure 3.7: The pattern for defining viewpoints (UML class diagram)

We have explicitly chosen not to prescribe a concrete syntax for MuDForM, because we want to have the flexibility to use notations that are familiar to the industry partners with which we did the case studies. As a result, we have experience with notations based on the KISS language [96], UML [120], and MPS [89].

Table 3.2: Example viewpoints

Viewpoint	Root	Derivation (<i>Viewpoint elements in this font</i>)
<i>Dependencies view</i>	<i>MuDForM model</i>	The <i>Specification space dependencies</i> of all the <i>Specification spaces</i> in the <i>MuDForM model</i> .
<i>Interaction view</i>	<i>Domain</i>	The <i>Domain activities</i> in the <i>Domain declarations</i> of the <i>Domain</i> . Per <i>Domain activity</i> , the <i>Activity roles</i> and the <i>Involvements of Domain classes</i> . The <i>Generalizations</i> in the <i>Generalization structure</i> of each <i>Domain activity</i> .
<i>Object lifecycle view</i>	<i>Domain class</i>	The <i>Flow steps</i> of the <i>Object lifecycle</i> of the <i>Domain class</i> , and their order according to the <i>Object lifecycle</i> .

3.6 Examples from a Case Study

This section presents example views from a case study that we performed together with domain experts from a high-tech company. The goal was to obtain a so-called system behavior description (SBD), and to validate the MuDForM definition. The high-tech company develops and produces products and services for printing and workflow management. Currently, SBDs contain mostly natural language text; the company wants to transform them into model-based specifications. The case in this

section is about one of in total 90 SBDs. It addresses the management of print jobs. Another part of the case study results, which demonstrates the MuDForM support for transforming textual input into an initial model, is presented in Chapter 4. The complete model is not publicly available due to intellectual property rights.

Sections 3.6.1, 3.6.2, and 3.6.3 respectively present a *Dependencies view*, an *Interaction view*, and an *Object lifecycle view*. Their descriptions clarify how the method ingredients from Section 3.5 are used. The diagrams are based on the UML syntax, and made with the modeling tool Enterprise Architect [152].

3.6.1 Specification Spaces and Dependencies of Printing Jobs

Figure 3.8 shows the *Specification spaces* and *Specification space dependencies* of the case study model. It complies with the metamodel fragment of Figure 3.4, and is an example of a *Dependencies view* as defined in Table 3.2. It is involved in the method step *Manage specification space dependencies*, described in Section 3.5.5. We have used UML packages and stereotypes to depict the different types of *Specification spaces*. A UML dependency arrow represents a *Specification space dependency*.

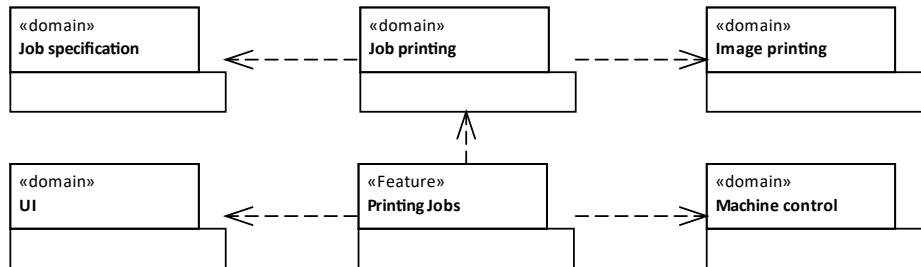


Figure 3.8: The *dependencies view* of the *MuDForM model* of the case study (UML package diagram)

There are five *domains* and one *feature* in the diagram. Each is a *specification element* in the *specification declarations* of the *MuDForM model*. The *feature* Printing jobs³ has a *specification space dependencies* structure, which contains *specification space dependencies* on the *domains* UI, Job printing, and Machine control. The *domain* Job Printing has a *specification space dependencies* structure that contains *dependencies* on the *domains* Job specification, and Image printing.

³Terms from the case study model are in sans serif font.

The diagram complies with the postconditions for the method step *Manage specification space dependencies*. The *domain Job* printing refers to two other *domains*, and the *feature Printing jobs* refers to three *domains*, which complies with the postcondition *Allowed dependencies*. It is clearly visible that the diagram complies with the postcondition *No cyclic dependencies*.

3.6.2 The Interaction View of the Domain Job printing

Figure 3.9 presents the *interaction view* (defined in Table 3.2) of the *domain Job* printing in UML notation. It complies with the metamodel fragment from Figure 3.5 and is the result of the method step *Specify interaction view*, described in Section 3.5.5. It contains 10 *domain activities*, depicted with a UML activity symbol, 3 *domain classes*, depicted with a UML class symbol, and 14 *activity roles*, each with one *involvement*, depicted with a line between an activity symbol and a class symbol.

When an *activity role* is *instantiating*, its *involvements* are depicted with a UML dependency relation with stereotype «*instantiate*». When an *activity role* is *not instantiating*, its *involvements* are depicted with a UML association relation. In both cases (dependency and association), the name of the relation is the name of the *activity role*. For example, the *role structure* of *domain activity* to *Copy* has three *activity roles*, with each one *involvement*: an nameless *activity role* with an *involvement* to *domain class* *Job*, an *activity role* to with an *involvement* to *domain class* *JobStore*, and an *instantiating activity role* giving with an *involvement* to *domain class* *Job*.

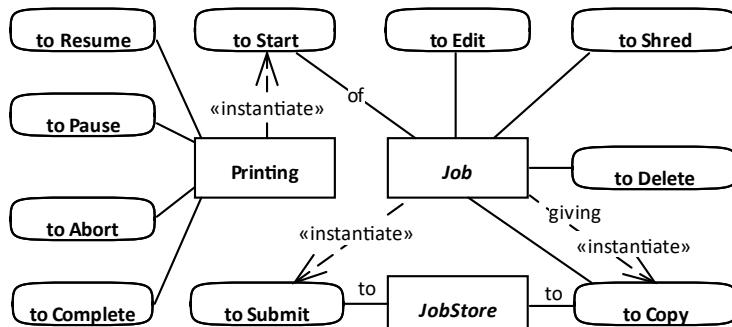


Figure 3.9: The *interaction view* of the *domain Job* printing (UML diagram)

3.6 Examples from a Case Study

An *Interaction view* can be verbalized in natural language; one phrase for each *Domain activity*. For example, to Edit a Job, to Start a Printing of a Job, and to Copy a Job to a JobStore, giving a Job. The *Domain activity name* is the verb, a nameless *Activity role* indicates the direct object, and named *Activity roles* indicate a preposition object.

To be clear, the possibilities of the Structure template class, which is the base of *Role structure*, are not fully demonstrated in this example view. The view in the next section demonstrates the capabilities of the structure pattern better.

It is easy to check if the given postconditions are met. The postcondition *Domain classes must have Domain activities* is true, because each *domain class* is related to several *domain activities*. The postcondition *Objects from a domain must be instantiated in that domain* is true for the *domain classes* Job and Printing, because they are both connected to *instantiating activity roles*. It is not true for *domain class* JobStore, because it does not have an *instantiating activity role* connected to it. (We assume that the diagram presents the complete *interaction view* of the *domain*.) The postcondition can become valid by adding a *domain activity* that *instantiates* a JobStore, e.g., to Open a Jobstore.

3.6.3 Object Lifecycle of Domain Class Job

Figure 3.10 depicts the *object lifecycle view*, defined in Table 3.2, of the *domain class* Job, and complies to the metamodel fragment of Figure 3.5. It is the result of the method step *Specify object lifecycle* defined in Section 3.5.5.

The diagram uses the symbols of UML activity diagrams and resembles the KISS notation [96]. The actions (activity symbols with a fork) in the diagram represent the *flow steps* that refer to the *involvements* of the *domain class* Job. The names of actions are a concatenation of the *activity role name* and the *domain activity name*, which makes it easy to check that if *involvements* of Job are specified in the interaction view of the *domain* Job printing (see Figure 3.6.2).

The activity symbols with the XOR-icon represent sub-*flow structures* within the *object lifecycle*. The XOR-icon indicates that the *structure type* is “*Selection*”, i.e., one of the contained *flow steps*, indicated with a UML composition relation, must be chosen. The REP-icon represents that the *flow attribute how many times?* has the value “*Zero or more*”, which means that the contained XOR-activity can take place zero or more times. Although the REP-activity and the second XOR-activity represent

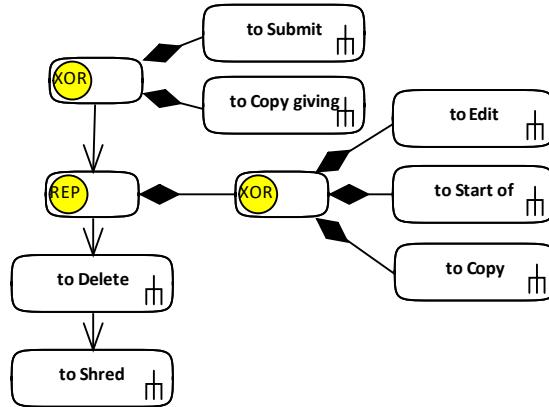


Figure 3.10: The *object lifecycle view* of the *domain class Job* (UML diagram)

one *flow structure*, we had to depict it with two activity symbols, because our UML tool (Enterprise Architect [152]) does not allow two icons within one activity symbol.

The diagram reads as follows. The *OLC* is a *flow structure* with the *structure type* “*Sequence*”. The life of a Job starts with a *flow structure* with *structure type* “*Selection*” (XOR) containing two *flow steps*: an *involvement* in the nameless *activity role* of the *domain activity* to Submit, and an *involvement* in the giving *activity role* of *domain activity* to Copy. Then follows a *flow structure* with *how many times?* equal to “Zero or more” (REP), *structure type* “*Selection*” (XOR), and three *flow steps*: an *involvement* in the nameless *activity role* of the *domain activity* to Edit, an *involvement* in the of *activity role* of *domain activity* to Start, and an *involvement* in the nameless *activity role* of *domain activity* to Copy. After that, follows an *flow step* of an *involvement* in the nameless *activity role* of *domain activity* to Delete, followed by *flow step* of an *involvement* in the nameless *activity role* of *domain activity* to Shred.

The diagram complies with the postconditions of the method step *Specify object lifecycle*. All the *involvements* of Job, visible in the *interaction view* (Figure 3.6.2), occur in the diagram. So, the postcondition *An OLC has steps for each involvement* is true. The *OLC* is a “*Sequence*” of several *flow steps*. So, the postcondition *An OLC describes a life* is true. The first *step* is a “*Selection*” between to Submit and to Copy giving, which both refer to an *instantiating activity role* (see Figure 3.6.2). So, the postcondition *The first step is instantiating* is also true.

The chosen UML notation enables the modeling of MuDForM *Flow structures*. One could also represent the *object lifecycle* of Job with a regular expression, resulting in the following string⁴:

(to Submit|to Copy giving)–>(to Edit|to Start of|to Copy)*–>to Delete–>to Shred

This section demonstrated how the method definition is used, and how some viewpoints can be represented in UML notation. We have experimented with other notations, *e.g.*, regular expressions for structures. We also have, together with an industrial partner, partially implemented the metamodel in MPS [89], which used a fully textual notation. The next section discusses literature about other domain-oriented methods that also provide a metamodel or method flow.

3.7 Related Work

The work related to ours can be found in three areas: literature on domain-oriented specification methods that also offer an explicit metamodel and method flow, literature on generic method engineering techniques, and literature on method engineering from a cognition-based perspective.

3.7.1 Domain-oriented Specification Methods

The SLR of Chapter 2 reports about domain-oriented specification techniques from the perspective of the MuDForM method objectives (see Section 1.1.3). They observed that none of the found methods has a complete method definition, *i.e.*, covering underlying model, notation, steps, and guidance. Some approaches provide a metamodel and a notation, and others provide high level steps, and sometimes guidance.

Most of the studies included in the SLR use UML [120] or MOF [119] as their underlying model. Some have their own metamodel, like KerMeta [62] or MEMO ML [63, 64]. Most metamodels limit themselves to structural concepts like classes, attributes, and relations between classes. Only ORM [129], Normalized Systems [107], and the KISS method [96, 80, 161] offer behavioral concepts as well.

Many studies included in the SLR provide a step for making a domain specification. Those steps are mostly coarse-grained, and reflect phases or stages in the specifi-

⁴A “–>” represents a *sequence*.

cation process. The only study that offers fine-grained steps for making a model is from Ibrahim and Ahmad [82]. The other studies do not offer detailed steps for transforming text into a model, nor do they provide steps and guidelines for engineering a model through multiple viewpoints.

The KISS method, on which MuDForM is based, does not provide a metamodel, fine-grained method steps, and detailed modeling guidelines. We have defined those for MuDForM. We have extended the KISS method with the concept of specification spaces, which offer support for handling multiple domains, features, contexts, and their relations. MuDForM offers modeling concepts for N-ary class relations, which are not supported by the KISS method. MuDForM has extended the KISS viewpoints with the Function signature, the Context structure view, the Feature structure view, the Actors view, and the Dependencies view.

3.7.2 Generic Method Engineering Techniques

In a paper from 1996, Brinkkemper introduces method engineering as a research framework for information systems development methods and tools [18]. That paper provides definitions of the terms method and technique, to which this chapter adheres. Brinkkemper also introduces situational method engineering, *i.e.*, when a method is created from fragments of other methods, mostly for a specific project. Although MuDForM is not created for a specific situation or project, it is clearly based on existing method fragments.

In another paper, Brinkkemper *et al.* [19] distinguish five granularity levels for method fragments: *method*, *stage*, *model*, *diagram*, and *concept*. The MuDForM definition can be seen at all these levels. Namely, MuDForM is an extension of the KISS *method*. The KISS method's *stages* Grammatical analysis, Transformation of text to model, and Model engineering, form, together with Scoping, the backbone of the MuDForM method flow. Although literature about the KISS method literature did not provide a metamodel, it is clear that its *models*, *diagrams*, and *concepts*, are present in the MuDForM definition. Consequently, the MuDForM definition is also usable at all five granularity levels.

The MuDForM definition offers several techniques for situational method engineering. Firstly, it is possible to replace the Grammatical analysis step (see Section 3.5.4) with another way of gathering usable input from documents and involved experts, *e.g.*, by analyzing Gherkin scenarios [150], textual requirements, or user interfaces, of existing systems. The prerequisite is that such a stage produces elements that can be

classified as a MuDForM modeling concept, such as *Domain class*, *Domain activity*, or *Function*. Secondly, via the Formalism pattern (see Section 3.A.2), MuDForM offers a technique for using concepts from other methods and languages. Thirdly, it is possible to use model structures and structural concepts of other methods by (partially) replacing the structure patterns defined in Section 3.A.3, *e.g.*, replace the structure of the object lifecycle view with a state diagram structure, or replace the generalization structure of classes with the corresponding part in the UML definition [120].

Engels and Sauer describe a meta-method for defining software engineering methods [49], which contains the specification of a method lifecycle that corresponds to the approach described in Section 3.2. The MuDForM definition does not cover all the core method engineering concepts that they distinguish. Engels and Sauer consider concepts for the organization and work process of software engineering, *e.g.*, milestone, task, artifact, discipline, and tool. Although these concepts are important when a method is implemented in an organization and applied in projects, they do not distinguish MuDForM from other methods.

3.7.3 Method Engineering from a Cognition-based Perspective

The problem statement in Section 3.1.1 refers to the work of Sia [144]. He proposes to use cognitive psychology as a reference discipline for method engineering. But, he does not concretely state how it can be used to define method ingredients. The MuDForM definition realizes explicit method objectives, and its modeling concepts are based on cognitive aspects, which demonstrate the use of knowledge from cognitive psychology as Sia suggests.

The problem statement also refers to the work of Wand [166] on ontology-based metamodeling. He distinguishes ontological elements and metamodel elements, but does not state what the proper ontological elements are. We think that the cognitive aspects of Section 3.4 can be seen as elements of an ontology of human cognition, which matches the idea of Wand.

Chapter 6 discusses the relation between MuDForM and cognitive science, linguistics, and philosophy in more detail.

3.8 Discussion

This section discusses the method definition from this chapter. Firstly, we reflect on the defined method ingredients and their application. Secondly, we reflect on the relation to the cognitive aspects.

3.8.1 Method Ingredients and their Application

The often-cited paper “When and how to develop domain-specific languages” from Mernik *et al.* [112] states that making a DSL is hard because it requires domain knowledge and language development knowledge, and that few people have both. We do not fully agree with this statement. We think that those two expertises do not have to present in one person. MuDForM provides guidance for developing a domain model, which can be used as the basis of a DSL. The method offers detailed steps and guidelines to let a modeler exploit domain knowledge of other people, via analyzing texts and interaction. Of course, this requires the availability of those people. We think that developing a domain model is not an art. It is deductive, and can be seen as an engineering activity. A positive side effect that we as modelers noticed many times, is that applying MuDForM causes one to learn the essential concepts in a domain very quickly.

It is an option not to make all the prescribed views. For example, one can decide to make only the *interaction view* and the *attribute views* of a domain model, which provides the ‘signature’ of a domain. Of course, this means that the model is not as well engineered as it could be. Namely, making the *object lifecycle views* and the *activity views* validates the other views, often leading to a better model. The point is that incomplete models are possible to save on modeling effort, but they might lead to less accuracy. By making more views, you not only capture more knowledge, you will also improve the model correctness.

The **structure patterns** (see Section 3.5.1) are used throughout the whole metamodel, which brings several benefits. They provide expressive relations between metaclasses, dealing with aspects like multiplicities, nesting, ordering, and (conditional) selection between the elements in an N-ary relation. They provide uniformity in relations, which simplifies their representation and implementation. The disadvantage is that sometimes not all possible instances of a used pattern are allowed, and that an extra constraint is required to express the restriction. See for example the constraint *Only specification spaces are allowed as specification element* in the diagram of Figure 3.4.

We could have modeled the declarations of a MuDForM model in another structure that only contains specification spaces, and thus does not require the extra constraint. But then it would not be clear that the essence is the same as the declarations in specification spaces, *i.e.*, being a container of specification elements.

3.8.2 Relation between the Cognitive aspects and Method Definition

Section 3.5.3 explained how modeling concepts are based on construction patterns and cognitive aspects, and presented a few examples. Appendix 3.B presented showed that almost all cognitive aspects from Figure 3.2 in Section 3.4 are used in the definition of at least one metaclass, and thus in a modeling concept.

The cognitive aspect **Analogy** is currently not supported via a dedicated modeling concept. An analogy can be achieved by identifying a model fragment that is the base of the analogy, and then replacing its model elements with the analogous elements. We think that having specific modeling concepts and method steps for making analogies will be an improvement to MuDForM, and as such, is a topic for further research.

Besides the relation between cognitive aspects and modeling concepts, we also **related some guidelines to a cognitive concept**, which are explained in detail in Chapter 6. We give two examples.

The guideline ‘Ensure consistency between classifiers and known instances’ corresponds with the observation of Steven Pinker [125] that people have to get something out of using categories, which is: inference. If you know a category of a thing, then you should be able to infer that that thing has the properties of the category.

Corballis [26] writes about the notions of episodic memory and semantic memory. Episodic memory is memory for events or episodes in our lives. Semantic memory is memory for statements about the world. Involved experts can be asked to share general knowledge about the target domain or system, which can be used directly as model input. But, this is not always easy. The guideline ‘Ask for examples when generic knowledge is hard to phrase’ appeals to the episodic memory of the involved expert.

3.9 Conclusion and Future Work

This chapter describes the definition of a specification method called MuDForM. We were interested in the following research question (RQ2): *How to define a specification method that is explicitly cognition-based?*

We conclude that the results from our study fill an important gap in the state of the art, which to the best of our knowledge, offers far less detailed method steps, guidelines, viewpoints, and is not explicitly cognition-based. We have defined detailed modeling concepts, method steps, guidelines, and viewpoints, and explained how the modeling concepts are cognition-based. We explained how these ingredients together form the MuDForM definition.

The current method definition lays the foundation for future work on the following topics:

- Extending the metamodel with modeling activities such that it is a MuDForM-compliant domain model itself, and that the method steps can be fully expressed in that domain model.
- Application of MuDForM in case studies that require the integration of multiple domains, and quality domains in particular. We think it is possible to define modeling concepts specifically for the specification of domain and feature integrations.
- Investigating what modeling concepts are useful for defining analogies, as suggested in Section 3.8.2, and validate them in case studies.
- Improvement and validation of the set of guidelines, which would comprise the following:
 - Search for guidelines in existing literature, and extending the list of useful publications [96, 138, 2, 79, 82, 47], which were found by our literature review from Chapter 2.
 - Conduct a literature review, to find and analyze guidelines from process modeling approaches.
- Consider the formal specification of the postconditions of the steps, and maybe the guidelines too, in OCL or another language.

3.A Detailed Explanation of Construction Patterns

- A community that actively validates, identifies, and manages guidelines. Building such community is not a research subject. It requires that MuDForM is actually used in practice, which is not easy to realize. We think it starts with providing training and tooling.

The current set of cognitive aspects are derived from a large range of literature, which was not found via a systematic process. Using explicit criteria for finding cognitive aspects and deciding if they should be used in a method definition are topics for future research. The starting point for this research is Chapter 6.

Finally, to facilitate industrial adoption, we have started to work on practical support for practitioners, such as instruction videos on modeling topics, and a modeling course. The method definition from this chapter is the starting point.

3.A Detailed Explanation of Construction Patterns

This section explains the construction patterns that are used in the metamodel in more detail. It is an extension of Section 3.A.3.

3.A.1 Named Elements

Figure 3.11 shows the two base classes to give model elements a name. The *Named element* class has attributes for the *name*, a *description*, and a list of possible *aliases*. The *Named relation* class also has an attribute *reverseName* to define how a directed relation is called in the reverse direction.

Of course, giving model elements meaningful names is not new. We just made the pattern explicit, and state that things with a name can also have a description, and a set of aliases.

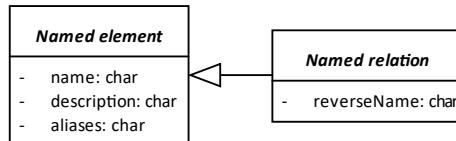


Figure 3.11: Named elements (UML class diagram)

3.A.2 Embedding Formalisms

Figure 3.12 shows the classes that make up the pattern, including examples of well-known formalisms.

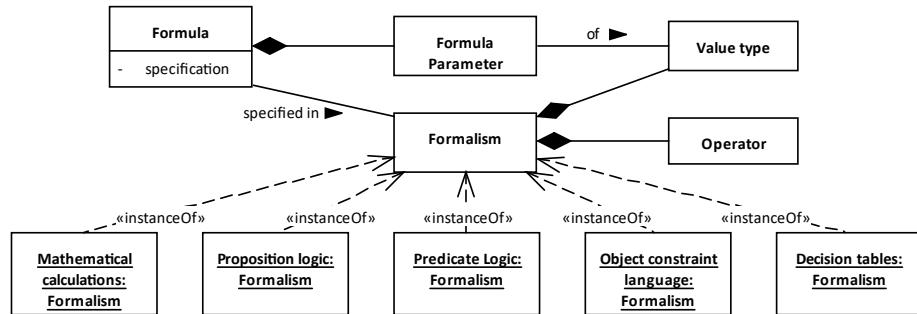


Figure 3.12: The pattern for including other formalisms (UML class diagram)

A *Formalism* consists of *Operators* and *Value types*. For example, the *Formalism* Proposition logic has Boolean *Operators*, such as AND and OR, and Boolean (true or false) as *Value type*. A *Formula* has *Formula parameters*, which get a value when the *Formula* is applied. A *Formula* is specified in a *Formalism*, in which the *Formula* uses the *Operators* and *Value types* of that *Formalism*. The given instances are examples of typical *Formalisms* that might be used in a MuDForM model.

Each *Formalism* must have an unambiguous syntax, varying from a string of characters to a decision table. For example, a *Formula* CalculateAverageSpeed with *Formula parameters* distance, duration, and averageSpeed, *specified in* the *Formalism* Mathematical calculations has a *specification* “averageSpeed:=distance/duration”.

3.A.3 Structures

This section describes the set of patterns that are used to create relations between model elements. The root pattern is **Structure** (see Figure 3.13). A *Structure* consists of *Elements*. An *Element* can be, but not necessarily, a *Substructure* that consists of other *Elements*, or a *Reference* to a *Referred item*. A *Substructure* is itself a *Structure*. In this way, a tree-structure can be created, where the nodes of the tree are the *Elements*, and the leaves, *i.e.*, *References*, can refer to other model elements, *i.e.*, the *Referred items*.

3.A Detailed Explanation of Construction Patterns

The structure pattern is used as the foundation for more specific structures, like the *Roles* of a *Domain activity*, and the *Object lifecycle* of a *Domain class*. To achieve this, the class *Structure* is declared as a template class. The parameters of the class correspond with the variable part of the pattern. The parameters *element*, *reference*, and *referred item* can be bound to specific classes when applying the *Structure* class.

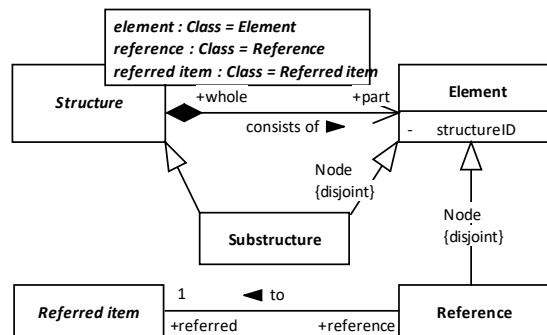


Figure 3.13: The structure pattern (UML class diagram)

A **Coordinated structure** (see Figure 3.14) is meant to constraint the instances of *Elements* of instances of *Structures*. The *structure type* defines which instances of *Elements* an instance of the *Coordinated structure* must or can contain. For example, the modeling concept *Domain class* has a *Coordinated structure* for the *Attributes*. If an *Domain class* Person has attributes name and registration code, then several cases can be expressed via the *structure type* attribute of *Coordinated structure*. If a Person must have a value for:

- name **and** code, then the *structure type* equals “*All of*” (AND).
- name **and/or** code, then the *structure type* equals “*Some of*” (OR).
- *either* a name **or** a code, then the *structure type* equals “*One of*” (XOR).

A **Static structure** is a *Coordinated structure*, where the parameter *element* is bound to the class *Static element* (see Figure 3.14). The attribute *how many?* specifies the possible number of instances of a *Static element*. For example, the modeling concept *Class* has a *Static structure* for the attributes of a *Class*. Then, if you define a *Class* *CarType* that has:

Chapter 3. Cognition-based Method Engineering of MuDForM

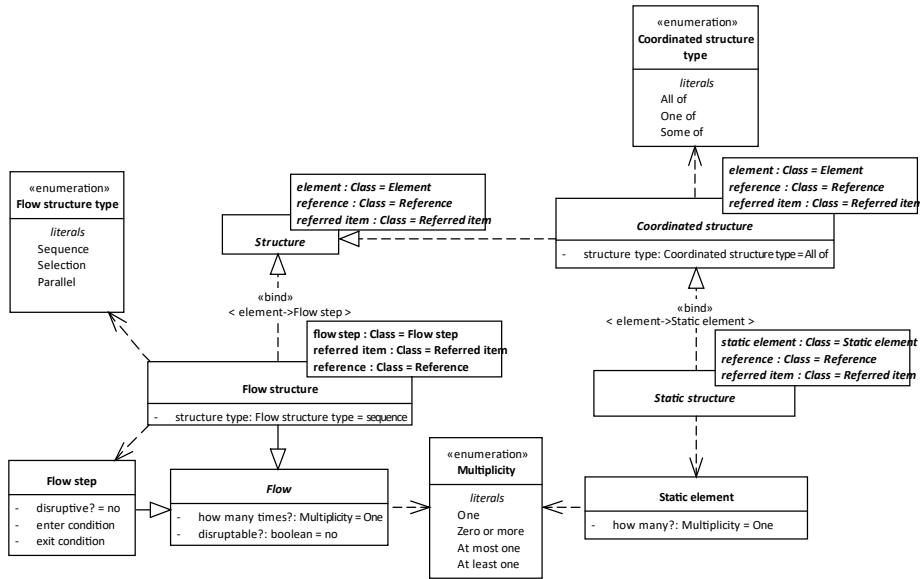


Figure 3.14: The coordinated, static, and flow structure patterns (UML class diagram)

- exactly one value for an *Attribute* `driverSeatType`, then *how many?* equals “*One*”. (This is the default value.)
- possibly several values for an *Attribute* `doorType`, then *how many?* equals “*Zero or more*”.
- possibly a value for an *Attribute* `roofType`, then *how many?* equals “*At most one*”.
- an *Attribute* for specifying one or more colors, *how many?* equals “*At least one*”.

A **Flow structure** is used to describe a temporal order of instances of *Referred items* (see Figure 3.14). A **Flow structure** is a **Structure** where the *element* parameter of the template class **Structure** is bound to the class **Flow step**. The order between the **Flow steps** is specified via the value of the attribute *structure type*, i.e., *Sequence*, *Selection*, or *Parallel*. The superclass **Flow** is used to indicate *how many times?* an instance of a **Flow structure** or **Flow step** can be executed, similar to the *how many?* attribute of a

3.B All Major Modeling Concepts

Static element. The attribute *disruptable?* indicates if a *Flow* can be disrupted by a succeeding *Flow step*. The attribute *disruptive* of the class *Flow step* indicates if that *Flow step* can preempt a preceding *Flow step*, which is only meaningful when the *structure type* is *Sequence*. The *enter condition* specifies what must be true to start the execution of a *Flow step*, and the *exit condition* states what must be true when the execution of a *Flow step* stops. The flow structure pattern enables the specification of process structures similar to process algebra [60].

3.B All Major Modeling Concepts

The MuDForM metamodel is defined in four packages (see Figure 3.15). The package *Generic concepts* defines modeling concepts that are usable in all specification spaces, or that are used in the definition of modeling concepts in the other packages. The package *Context concepts* defines modeling concepts that are used in context models, which may be based on generic modeling concepts. The package *Domain concepts* defines modeling concepts that are used in domain models, which may be based on generic modeling concepts, and on context modeling concepts. The package *Feature concepts* defines modeling concepts that are used in feature models, which may be based on generic modeling concepts, on context modeling concepts, and on domain modeling concepts.

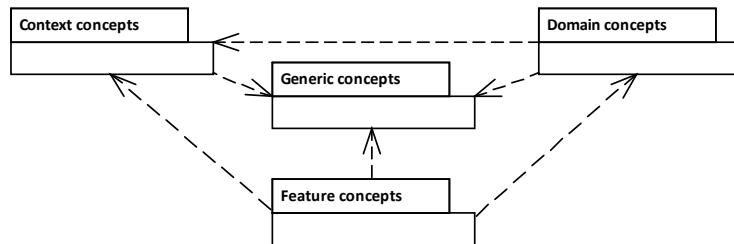


Figure 3.15: Structure of the metamodel definition (UML package diagram)

Table 3.3 defines the classes in the metamodel. For each class, the table lists the parent classes, the applied construction patterns from Section 3.A.3, and the cognitive aspects from Section 3.4 on which it is based. For the sake of space, the minor modeling concepts and the attributes of the modeling concepts are omitted. Of course, the complete metamodel [33] contains all the metaclasses, including a description of each metaclass.

Chapter 3. Cognition-based Method Engineering of MuDForM

Table 3.3: Modeling concepts construction based on cognitive aspects

Meta-model class	Parent classes	Applied construction patterns (Pattern (elements) terms in this font)	Based on cognitive aspects (Aspect in this font)
Generic concepts			
<i>Specification element</i>		<i>Specification element</i> is a <i>Named element</i>	The name of a model element, together with its type, <i>i.e.</i> , the modeling concept, realize the aspect <i>Symbol</i> . The relation <i>identifies</i> is realized by ensuring that the combination of model element name and type is unique in its scope, <i>i.e.</i> , the <i>Statement context</i> .
<i>Specification space</i>	<i>Specification element</i>	1. has <i>Specifications declarations</i> , which is a <i>Coordinated structure</i> where: <i>element</i> → <i>Specification element</i> . 2. has <i>Specification space dependencies</i> , which is a <i>Coordinated structure</i> where: <i>reference</i> →‘ <i>Specification space dependency, referred item</i> → <i>Specification space</i> . 3. has <i>Specification space declarations</i> , which is a <i>Coordinated structure</i> where: <i>element</i> → <i>Specification element</i> . Only <i>Specification spaces</i> are allowed as <i>Specification element</i> of a <i>MuDForM model</i>	The <i>Specifications declarations</i> structure realizes the name space aspect of <i>Statement context</i> . The <i>Specification space dependencies</i> express which other <i>Specification spaces</i> form the <i>Statement context</i> of <i>Specification elements</i> in the <i>Specification space</i> . Via the <i>Specification declarations</i> structure, a <i>MuDForM model</i> forms a <i>Statement context</i> for the contained <i>Specification spaces</i> .
<i>MuDForM model</i>			

3.B All Major Modeling Concepts

Meta-model class	Parent classes	Applied construction patterns <i>(Pattern (element) terms in this font)</i>	Based on cognitive aspects <i>(Aspect in this font)</i>
<i>Classifier</i>		has a <i>Generalization structure</i> , which is a <i>Coordinated structure</i> where: <i>reference</i> → <i>Generalization</i> , <i>referred item</i> → <i>Classifier</i> .	All modeling concepts are based on the aspect <i>Category</i> , because the only practical application of a model is that it can be instantiated multiple times. The model elements that are a <i>Specification element</i> in a <i>Declarations</i> structure, all have the abstract class <i>Classifier</i> as a parent class. The <i>Generalization structure</i> is based on the aspect <i>Property</i> , as it expresses that one <i>Category</i> , i.e., the <i>Generalization</i> , is a characteristic of another <i>Category</i> .
<i>Attribute container</i>		has an <i>Attribute structure</i> , which is a <i>Static structure</i> where: <i>reference</i> → <i>Attribute</i> , <i>referred item</i> → <i>Class</i> . <i>Attribute</i> is a <i>Named element</i>	Expresses the aspect <i>Property</i> in case the characteristic is a <i>Class</i> .
<i>Class</i>	<i>Classifier</i>		Based on <i>Entity</i> , i.e., it represents a <i>Category</i> of identified Concepts.
<i>Activity</i>	<i>Classifier</i>	has a <i>Behavior composition</i> , which is a <i>Static structure</i> where: <i>reference</i> → <i>Activity reference</i> , <i>referred item</i> → <i>Activity</i> .	Based on <i>Event</i> . The <i>Attribute structure</i> expresses the involves relation in case the involved Concept is a <i>Class</i> . The <i>Behavior composition</i> expresses a <i>Property</i> in case the characteristic is an <i>Activity</i> .
Context concepts			

Chapter 3. Cognition-based Method Engineering of MuDForM

Meta-model class	Parent classes	Applied construction patterns (Pattern (element) terms in this font)	Based on cognitive aspects (Aspect in this font)
Context	Specification space	has <i>Context declarations</i> , which is a <i>Specification declarations</i> , where <i>specification element</i> → <i>Context element</i>	Represents a Statement context for Statements needed to make Statements in other Statements contexts.
Class	Context element, Attribute container		To represent an Entity from a Context.
Value type	Class	has attributes <i>type</i> and <i>possibleValueSpec</i> of type <i>Value type</i> , which can be defined via a used <i>Formalism</i> .	Is used to express a single value Entity that exists outside the <i>Domain</i> or <i>Feature</i> of interest. The <i>possibleValueSpec</i> specifies the Entities that are identified by such values.
Operation	Context element, Activity, Attribute container	has an attribute <i>formula</i> of type <i>Formula</i> defined via a used <i>Formalism</i>	Represents an Event from a Context.
Class relation	Context element, Classifier	<i>Class relation structure</i> , which is a <i>Static structure</i> , where <i>element</i> → <i>Class relation role</i> , <i>reference</i> → <i>Role connection</i> , <i>referred item</i> → <i>Class</i> . <i>Class relation role</i> is a <i>Named relation</i> .	

3.B All Major Modeling Concepts

Meta-model class	Parent classes	Applied construction patterns <i>(Pattern (element) terms in this font)</i>	Based on cognitive aspects <i>(Aspect in this font)</i>
<i>Actor</i>	<i>Class, Activity</i>		<i>Actor</i> is based on <i>Agent</i> . The <i>Behavior composition</i> express the capabilities of an <i>Actor</i> , and the <i>modality</i> attribute of an <i>Activity reference</i> expresses the <i>performs</i> and <i>perceives</i> relation.
Domain concepts			
<i>Domain</i>	<i>Specification space, Attribute container</i>	has <i>Domain declarations</i> , which is a <i>Specification declarations</i> where: <i>specification element</i> → <i>Domain element</i>	Represents a <i>Statement context</i> for <i>Descriptive statements</i> .
<i>Domain class</i>	<i>Domain element, Class</i>	1. has an <i>Object lifecycle</i> , which is a <i>Flow structure</i> where: <i>referred item</i> → <i>Involvement</i> 2. has <i>Object life dependencies</i> , which is a <i>Static structure</i> where: <i>reference</i> → <i>Object life dependency</i> , <i>referred item</i> → <i>Class</i>	Is used to model <i>Objects</i> . The <i>Object lifecycle</i> expresses the <i>order</i> relation. The <i>Object life dependencies</i> express hierarchy <i>Properties</i> between <i>Domain classes</i> .
<i>Domain activity</i>	<i>Domain element, Activity</i>	1. has a <i>Role structure</i> , which is a <i>Static structure</i> where: <i>element</i> → <i>Activity role</i> , <i>reference</i> → <i>Involvement</i> , <i>referred item</i> → <i>Domain class</i> . <i>Activity role</i> is a <i>Named relation</i> . 2. has an <i>Activity flow</i> , which is a <i>Flow structure</i> , where: <i>flow step</i> → <i>Activity operation</i> , <i>reference</i> → <i>Operation invocation</i> , <i>referred item</i> → <i>Operation</i>	Is used to model <i>Actions</i> . The <i>Role structure</i> expresses the <i>involves</i> relation. The <i>Activity flow</i> expresses the <i>order</i> of the <i>Operations</i> that are a <i>Property</i> of the <i>Domain activity</i> .

Chapter 3. Cognition-based Method Engineering of MuDForM

Meta-model class	Parent classes	Applied construction patterns (Pattern (element) terms in this font)	Based on cognitive aspects (Aspect in this font)
Feature concepts			
<i>Feature</i>	<i>Specification space, Function</i>	has <i>Feature declarations</i> , which is a <i>Specification declarations</i> where: <i>specification element</i> → <i>Feature element</i>	Represents a Statement context for Prescriptive statements.
<i>Function</i>	<i>Feature element, Activity</i>	1. has a <i>Function lifecycle</i> , which is a <i>Flow structure</i> where: <i>flow step</i> → <i>Function step, referred item</i> → <i>Function event</i> . (<i>Function event</i> is an <i>Activity reference</i>). 2. has a <i>Behavior life dependencies</i> , which is a <i>Static structure</i> where: <i>reference</i> → <i>Behavior life dependency, referred item</i> → <i>Function</i>	<i>Function</i> is a combination of Action and Object. The <i>Function lifecycle</i> expresses the order and causes relations. The <i>Behavior life dependencies</i> expresses which <i>Activities</i> are a Property of the <i>Function</i> .

3.C All Method Steps

This section describes the steps of the complete MuDForM method flow, depicted in Figure 3.16. We only included the step description, and left out the guidelines, preconditions, and postconditions. Those can be found in the complete method definition [33].

Although the method flow is presented as a sequence of steps⁵, they are often performed iteratively. For instance, the *Grammatical analysis* starts with a subset of the targeted input text, which is transformed into a model, and then incrementally more text is analyzed. Moreover, when a modeling decision requires more information, it is possible to go back to a previous step to check if the required information was already present, yet overlooked.

⁵For readability, the start- and end-nodes are omitted. The first step is the step without incoming control flow. The end step has no outgoing control flow.

3.C All Method Steps

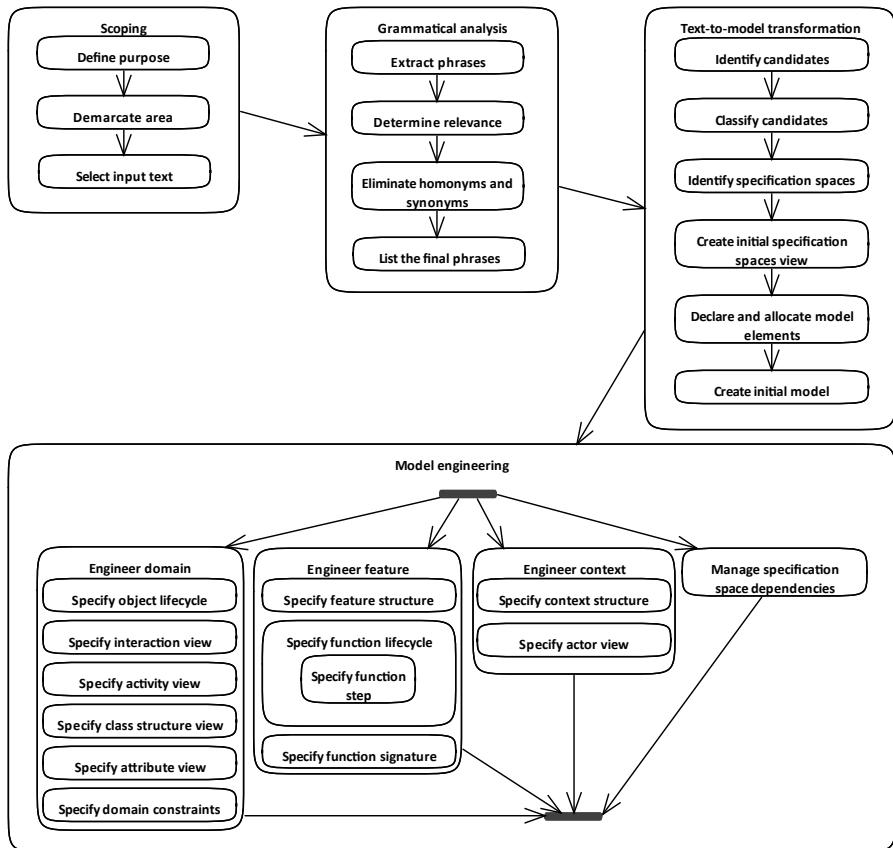


Figure 3.16: The complete method flow (UML activity diagram)

1. **Scoping:** The scope of the targeted model is specified. The goal of scoping is 1) to prevent unnecessary modeling work, and too little modeling work 2) to have relevant input, and 3) to keep the model and the modeling process manageable. The steps of Scoping are:
 - (a) **Define purpose:** Specify who the users/customers of the targeted model (or parts thereof) are, and what they want to do with it.
 - (b) **Demarcate area:** Name concepts that are expected to be in scope, and concepts that are out of scope.

- (c) **Select input text:** Explicitly state which pieces of text from a knowledge source are the starting point. The knowledge source is often an existing document, or a document that is created from interviews with (domain) experts. For each piece of input text, a domain expert is appointed to provide missing information and assist with inconsistencies during analysis and modeling. When there is no existing input text, the identification of possible model elements can be covered by letting (domain) experts coming up with sentences about the targeted scope.
2. **Grammatical analysis:** the input text is analyzed and transformed into a set of phrases with terms that are candidate elements for the model. The goal of this step is to maximize the knowledge elicitation from the source, and to make the resulting model traceable to the input. The steps of grammatical analysis are:
- (a) **Extract phrases** from the selected input text and format them according to one of the phrase types: interaction structure phrase, static structure phrase, state structure phrase, and conditional phrase. (The explanation of these terms is given in the complete method definition [36]). Typically, each sentence from the input text leads to one or more extracted phrases. As such, the extracted phrases form a decomposition of the original sentence. The next method steps might lead to changes in terminology or structure of the phrases. Based on those changes, it is always possible to rewrite an original sentence in the new terminology or structure, to check if the model still expresses the initially intended meaning. When there is no input text selected, sentences can be elicited directly from domain experts.
 - (b) **Determine the relevance** of each extracted phrase from the perspective of the defined scope. Discard phrases that do not fit the scope definition, or that are duplicates.
 - (c) Check all phrases for **homonyms and synonyms**, and **eliminate** them in consultation with the involved (domain) experts to assure that all terms (words) have exactly one meaning, and that all relevant meanings are covered by exactly one term.
 - (d) **List the final phrases** via these criteria: all extracted phrases that are marked as relevant and that are not discarded, all newly added and rewritten phrases, and replace of the homonyms and synonyms with the chosen term.

The KISS method [96], which is the starting point for MuDForM definition, provides a more detailed description of the Grammatical analysis phase.

3. **Text-to-model transformation:** This is the transition from working with text to working with models. The specification spaces, which form the top-level structure of a model (see Section refME:specification spaces and dependencies), are identified, and the phrases are transformed into pieces of model. The transformation consists of the following steps:
 - (a) **Identify candidates:** Determine the terms in the phrases, *i.e.*, nouns, verbs, adjectives, and adverbs, that are a potential element of the model.
 - (b) **Classify candidates:** Select which type of element each identified term is. The possible types are: domain class, domain activity, context class, function, attribute, domain, feature, context, operation, condition, function event, function step, activity operation, class relation, or specialization.
 - (c) **Identify specification spaces:** Identify contexts, domains, and features. Choose spaces for specification elements that are coherent. Each specification space should have an owner who is responsible for its content.
 - (d) **Create initial specification spaces view:** Create a view (*e.g.*, a diagram) with all the specification spaces. Create relations (dependencies or sub-declarations) between spaces if they are expected or already known, in compliance with the rules specified in Section refME:specification spaces and dependencies.
 - (e) **Declare and allocate elements:** Place each specification element in the most logical specification space. An element can be reallocated during model engineering.
 - (f) **Create initial models:** Create a first version of the models in the specification spaces from the list of final phrases. In the case that there is no input text, and hence no grammatical analysis, this is the starting point of the modeling process.
4. **Model engineering:** The model is completed and inconsistencies are solved by following the method steps and guidelines, and iterating over the different views. Domain experts are involved to answer questions and decide about missing concepts and model conflicts. Model engineering consists of a step to **manage the dependencies between the specification spaces**, and three steps for **engineering** the different types of specification spaces, *i.e.*, **contexts, domains, and features**. There is no restriction to the order of the main steps. But typically, the focus is on engineering a domain or feature, and capture in a context the needed concepts that are not in the domain or feature of interest.

When the focus is on engineering a feature, then existing domain models and feature models might be used to specify the behavior of the functions in the feature, which also validates the used models. Often during feature engineering, new domain model elements and context model elements are identified. They have to be added to the domain or context models, to assure that all the concepts used in the feature model are properly defined. The four main steps are composed of sub-steps in the following way:

- (a) **Engineer domain** consists of the following steps:
 - i. **Specify object lifecycle:** For each domain class, define the order in which an instance of the domain class can participate in instances of the related domain activities, *i.e.*, the involvements of the domain class in activity role are placed in the object lifecycle as a step, and connected to the other steps. As the object lifecycle is a flow structure, it is possible to use sub-flows for the coordination of steps, *i.e.*, to select between sequences of steps or to state that several sequences of steps are executed in parallel.
While making an object lifecycle, it is possible that new domain activities are discovered, which impacts the next steps.
 - ii. **Specify interaction view:** Define the domain activities and their roles, and which domain classes are involved in each role. Define the possible generalizations between domain activities.
 - iii. **Specify activity view:** Create a view for the activity flow of each domain activity, which expresses the order of the operations that make up the activity. Operations can be a local operation defined in a formula (following the pattern described in Section refME:formalisms), invocations of operations defined in a context, operations that set object attributes, or operations that instantiate, link or unlink objects. Possibly specify preconditions for the domain activity. Operations and preconditions may use the activity's attributes and the attributes of involved domain classes. Instead of making an activity flow, it is possible to only specify the postcondition of the domain activity.
 - iv. **Specify class structure view:** Define the domain class relations, the generalizations for each domain class, and the object life dependencies between domain classes.
 - v. **Specify attribute view:** Define the attributes of domain activities and domain classes.

- vi. **Specify domain constraints:** Add invariants to the domain, domain classes, domain class relations, and attributes. This step is typically done at the end of domain engineering and can be needed when there are domain properties that cannot be expressed via the other modeling concepts.
- (b) **Engineer feature:** A feature is engineered by working in parallel on the feature structure, function lifecycles, and function signatures. More details about feature engineering can be found in Chapter 5. Engineer feature consists of the following steps:
- i. **Specify feature structure:** Manage the behavioral composition of the feature and its functions. This means specifying the decomposition of the feature into functions, and possibly of each function into sub-functions. Additionally, the use of behavioral elements (operations, activities, and functions) from outside the feature is specified. The feature is the root of the resulting tree structure.
 - ii. **Specify function lifecycle:** The control flow of each function is specified, *i.e.*, describing the order in which the function steps must be executed. The function steps refer to sub-behaviors of the function, which are defined in the feature structure. All the sub-behaviors of the functions must occur at least once as a function step. The sub-behaviors are references to domain activities, operations, or functions, which must be declared in one of the specification spaces that the containing feature depends on. MuDForM distinguishes different types of ordering: sequences, selections, concurrency, or iterations. There are typically two ways to reason about the lifecycle. The first way is to start with the main input attributes of the function and decide which activities should be performed on them going forwards in time. The second way is to start with the end of the function in mind, which can be a postcondition or the execution of an essential domain activity, and then reason backwards about the order of the steps.
 - iii. **Specify function step:** Each function step is related to the context with respect to the objects (parameters) that play a role in the step. The following aspects must be specified:
 - Function attributes are allocated to the (actual parameters of) the step. The function attributes must belong to the same function as the step, or they belong to a function that contains the function. Typically, a feature, which itself is also a function, has attributes

that will be used in many steps of the functions of the feature.

- The preconditions for this step are defined, *i.e.*, the constraints on the step participants. A condition is typically expressed in a logical language and may only use terms that are elements within the scope of the function. (Step preconditions are also called enter conditions.)
- The postconditions for this step, that is, the conditions that have to be true for this step to end. (Step postconditions are also called exit conditions.)

A specified function lifecycle must be consistent with the domains that the function uses, which means that for function steps that are an instance of a domain activity holds that: 1) The objects allocated to the action have a type that corresponds to a domain class that is involved in the domain activity. 2) Function attributes are allocated to each input attribute of the action and their types match. 3) The function lifecycle does not violate the object lifecycle of an involved object (which can only be fully controlled at execution time and not during modeling).

- iv. **Specify function signatures:** A function signature describes the interface of each function in terms of attributes and events, and frames the behavior of the function. Attributes are typed by a (domain) class, and events are typed by an Activity. All function signatures can be put in a single view, or a separate view is created for important and complex functions. Each attribute can be an input, output, or local attribute. Local attributes, which are used to pass on data between function steps, could be omitted from the signatures, because they are not visible outside the function. But then, a different view must be created for the declaration of the local attributes. Thus, normally we put them in the same view. The function signatures also specify the preconditions and invariants that hold for the function attributes, that is, things that must be true in order to guarantee the proper outcome of the function. It is possible to specify postconditions, but this is not necessary, because MuDForM follows a whitebox perspective on function specifications. Although, a postcondition could help to guide the design of the function lifecycle.

- (c) **Engineer context:** Specify the operations, classes, and actors, and cross-check their use in domains and features. Engineer context consists of the following steps:
 - i. **Specify the context structure:** Identify the operations and classes, and their attributes, and the class relations, the class relation roles, and the role connections. Specify the generalizations between classes, between class relations, and between operations. Express the value types and formulas of operations in terms of the used formalisms.
 - ii. **Specify actor view:** Identify the actor, its attributes, and its generalizations. Specify in the behavior composition structure which activities the actor can execute, call, and react to.
- (d) **Manage specification space dependencies:** Define the dependencies between the different specification spaces. Define a dependency from space P to space Q (the dependency structure of P has a reference to Q) if P has elements that refer to elements of Q. This step exists to guard the correct use of model elements across specification spaces.

3.D All Viewpoints

Table 3.4 defines all MuDForM viewpoints via the pattern explained in Section 3.5.6.

A MuDForM model may contain a subset of the views, depending on the purpose of the model. Furthermore, views can be combined when a model is small. For example, the Attribute view and the Class structure view can be combined in one diagram that shows all the static relations of the *Domain classes*. On the other hand, when a view becomes too big, it might be split into sub-views. For example, the Attribute view of a *Domain* can be split into multiple attribute views, *e.g.*, one view per *Domain element*. Deviating from the standard viewpoints does not change the semantics of the model, because the meaning of the model is fully captured by the model elements and their relations.

Chapter 3. Cognition-based Method Engineering of MuDForM

Table 3.4: All viewpoints

Viewpoint	Root	Derivation (<i>Viewpoint elements in italics</i>)
MuDForM model		
Declarations view	<i>MuDForM model</i>	The <i>Specification elements</i> , which are all <i>Specification spaces</i> , in the <i>Specification declarations</i> of the <i>MuDForM model</i> .
Dependencies view	<i>MuDForM model</i>	The <i>Specification space dependencies</i> of all the <i>Specification spaces</i> in the <i>MuDForM model</i> .
Context		
Declarations view	<i>Context</i>	The <i>Context elements</i> in the <i>Context declarations</i> of the <i>Context</i> .
Context structure view	<i>Context</i>	<p>The <i>Operations</i> and <i>Classes</i> in the <i>Context declarations</i> of the <i>Context</i>.</p> <p>The <i>Attributes</i> in the <i>Attribute structures</i> of those <i>Operations</i> and <i>Classes</i>.</p> <p>The <i>Class relations</i>, and the <i>Class relation roles</i> and <i>Role connections</i> in the <i>Role structure</i> of each <i>Class relation</i>.</p> <p>The <i>Generalizations</i> in the <i>Generalization structure</i> of each <i>Operation</i>, <i>Class</i>, and <i>Class relation</i>.</p>
Actors view	<i>Context</i>	<p>The <i>Actors</i> in the <i>Context declarations</i> of the <i>Context</i>.</p> <p>For each <i>Actor</i>, the <i>Activities</i> it can <i>react to</i>, <i>generate</i>, and <i>perform</i> as specified via the <i>Activity references</i> in the <i>Behavior composition</i> of the <i>Actor</i>.</p>
Domain		
Declarations view	<i>Domain</i>	The <i>Domain elements</i> in the <i>Domain declarations</i> of the <i>Domain</i> .
Interaction view	<i>Domain</i>	<p>The <i>Domain activities</i> in the <i>Domain declarations</i> of the <i>Domain</i>.</p> <p>Per <i>Domain activity</i>, the <i>Activity roles</i> and the <i>Involvements</i> of <i>Domain classes</i>.</p> <p>The <i>Generalizations</i> in the <i>Generalization structure</i> of each <i>Domain activity</i>.</p>

3.D All Viewpoints

Viewpoint	Root	Derivation (<i>Viewpoint elements in italics</i>)
Class structure view	<i>Domain</i>	<p><i>Domain classes</i> and <i>Domain class relations</i> in the <i>Domain declarations</i> of the <i>Domain</i>.</p> <p>For each <i>Domain class</i>, the <i>Object life dependencies</i> in the <i>Object life dependencies</i> structure of the <i>Domain class</i>.</p> <p>For each <i>Domain class</i>, the <i>Attributes</i> in the <i>Attribute structure</i> of the <i>Domain class</i> if the <i>Attribute refers to a Domain Class</i> in the same <i>Domain</i>. For each <i>Domain class relation</i>, the <i>Class relation roles</i> and <i>Role connections</i> in the <i>Role structure</i> of the <i>Domain class relation</i>.</p> <p>For each <i>Domain class</i> and <i>Domain class relation</i>, the <i>Generalizations</i> in their <i>Generalization structure</i>.</p>
Attribute view	<i>Domain</i>	<p><i>Attributes</i> and their types from the <i>Attribute structure</i> of each <i>Domain class</i>, <i>Domain activity</i>, and <i>Domain class relation</i> from the <i>Domain declarations</i>, including the <i>Attributes</i> of the <i>Activity roles</i> and <i>Involvements</i>.</p>
Object lifecycle view	<i>Domain class</i>	The <i>Flow steps</i> of the <i>Object lifecycle</i> of the <i>Domain class</i> , and their order according to the <i>Object lifecycle</i> .
Activity view	<i>Domain activity</i>	<p>The <i>Activity operations</i> and <i>Operation invocations</i> in the <i>Activity flow</i> of the <i>Domain activity</i>, and their order.</p> <p>The <i>bindings</i> of the <i>Domain activity's Attributes</i> and <i>Attributes</i> of involved <i>Domain classes</i>, to the <i>Actual parameters</i> in the <i>Actual parameter structure</i> of the <i>Operation invocations</i>.</p>
Feature		
Declarations view	<i>Feature</i>	The <i>Feature elements</i> in the <i>Feature declarations</i> of the <i>Feature</i> .
Functional composition	<i>Feature</i>	All the <i>Function events</i> in the <i>Behavior composition</i> of all the <i>Functions</i> of the <i>Feature</i> (including the <i>Feature</i> itself).
Function lifecycle	<i>Function</i>	<p>The <i>Function steps</i> in the <i>Function lifecycle</i> of the <i>Function</i>, and their order.</p> <p>The <i>Step participants</i> in the <i>Function step participants</i> structure, and the <i>bindings</i> of <i>Function Attributes</i> of to the <i>Step participants</i>.</p>
Function signature	<i>Function</i>	<p>All the <i>Attributes</i> in the <i>Attribute structure</i> of the <i>Function</i>.</p> <p>All the <i>Function events</i> in the <i>Behavior composition</i> of the <i>Function</i> that have a <i>behavior modality</i> equal to “<i>Observe</i>” or “<i>Call</i>”.</p>

4 Methodical Conversion of Text to Models: MuDForM Definition and Case Study

“Words in papers, words in books
Words on TV, words for crooks
Words of comfort, words of peace
Words to make the fighting cease
Words to tell you what to do
Words are working hard for you
Eat your words but don’t go hungry
Words have always nearly hung me
What are words worth?
What are words worth? Words”

Wordy Rappinghood,
Tom Tom Club, 1981

This chapter is based on  R. Deckers and P. Lago “*Methodical Conversion of Text to Models*”, CEUR Workshop Proceedings of PoEM 2022 Forum, 15th IFIP Working Conference on the Practice of Enterprise Modeling 2022, November 23-25, 2022, London UK [36].

Chapter 4. From Text to Model with MuDForM

This chapter addresses RQ3.

Context. To enable the people involved in a software development process to communicate and reason close to their area of knowledge, we are investigating a method to formalize and integrate knowledge into domain models and into specifications in terms of those domain models. For this purpose, we have defined a vision and a set of method objectives (see Section 1.1). We performed an SLR, which concluded that there is not much methodical support for the using natural language in making specifications (see Chapter 2).

Goal. Provide methodical support for using text in the creation of model-based specifications, enabling a systematic and traceable process for converting knowledge in texts from (domain) experts into models.

Method. We made a metamodel of the grammatical analysis phase of the KISS method, and captured guidelines and extra concepts through an action research approach (as described in Chapter 3).

Result. A definition of the method part that covers the creation of an initial model from textual documents via systematic grammatical analysis, which is especially helpful in the transition from a text-based to a model-driven development process. We performed a case study in the printing domain to validate the method. We found that the presented analysis concepts, method steps, and guidelines help to systematically convert a textual specification into an unambiguous model.

Contents

4.1	Introduction	111
4.2	Background: MuDForM Development	113
4.3	Research Methodology	114
4.4	MuDForM Overview	116
4.4.1	MuDForM Modeling Process	116
4.4.2	MuDForM Model Structure	118
4.4.3	MuDForM Model Elements	118
4.5	Grammatical Analysis	119
4.6	Text-to-Model Transformation	120
4.7	A Case Study: System Behavior Description of the History Feature	122
4.7.1	Case Study Overview and Execution	122
4.7.2	Grammatical Analysis of the History SBD	123
4.7.3	Text-to-Model Transformation for the History Domain Model	125
4.8	Discussion	125
4.9	Related Work	128
4.10	Conclusion and Future Work	130

4.1 Introduction

This chapter explains the text-to-model phase of MuDForM, which provides support for the creation of *domain models* (DMs), and for the creation of models that are defined in terms of a domain model, called *domain-based models* (see Fig. 4.1). Together, these are called *domain-oriented models*. MuDForM provides analysis and modeling concepts, steps, and guidelines to conduct a modeling process, which starts with a knowledge source, like a *(domain) text* or *(domain) expert*. The SLR of Chapter 2 observed that the existing approaches for domain modeling approaches do not offer much methodical support for the extraction of model elements and model fragments from *(domain)* texts. Some offer a metamodel, some offer process steps, and some offer guidelines. But none of them integrates them all. This chapter explains the MuDForM part for creating an initial *MuDForM model* from a text, and demonstrates it in a case study.

The rest of this section describes the problem we aim to address, our contribution, and target audience. Section 4.2 explains in more detail what we aim to achieve

Chapter 4. From Text to Model with MuDForM

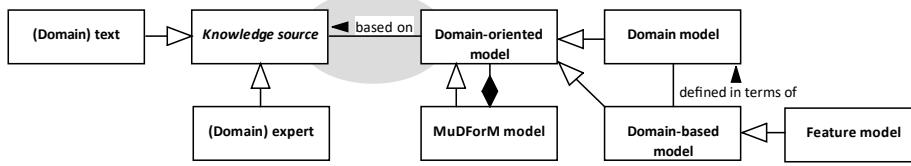


Figure 4.1: Context of MuDForM models (UML class diagram)

with MuDForM. Section 4.3 explains the research methodology. Section 4.4 gives an overview of MuDForM, which explains how the support for grammatical analysis and the text-to-model transformation, respectively defined in Sections 4.5 and 4.6, are integrated in MuDForM. Section 4.7 reports on a case study in which we applied MuDForM to formalize system behavior descriptions. Section 4.8 reflects on MuDForM’s support for grammatical analysis. Section 4.9 discusses related work, and Section 4.10 concludes the paper and presents suggestions for future work.

Problem Statement. When organizations are transitioning from a development process based on specifications in natural language to a model-based development process, they face the challenge of creating correct models from not only the input of (domain) experts, but also from existing system specification documents. A process that utilizes such documents, which often have cost significant effort, such that it minimizes the need for involvement of often busy domain experts, would be a great advantage.

Kosar *et al.* [95] present a systematic mapping study on domain specific languages (DSLs). They conclude that (domain) analysis is mostly done in an informal and incomplete way. Among the reasons for this weakness, they mention that domain analysis is too complex and outside software engineers’ competencies. Czech *et al.* [28] gathered 130 best practices from 19 studies on domain-specific modeling. They group the best practices in different classes: domain model, language design and concepts, generators, DSL-tooling, metamodel tooling, and practices that concern an entire domain-specific modeling solution. Only 3 best practices are about the domain model, and those are actually not about modeling itself, but about the context of a domain model. We observe that they did not find and distill any best practices for extracting domain models from text.

The systematic literature review (SLR) from Chapter 2 observes that most approaches for domain-oriented specifications do not offer full methodical support, *i.e.*, a meta-

4.2 Background: MuDForM Development

model, notation, fine-grained method steps, and guidelines, for extracting models from natural language texts. Some offer parts of those, but none integrates them all.

MuDForM explicitly aims at making the (domain) analysis phase a systematic activity, with integrated metamodel, steps, and guidelines, starting from a natural language text, in order to make the creation of models more predictable and easier to learn.

Contribution and Audience. This chapter has two main contributions. First, it presents integrated methodical support for the analysis of domain texts to extract model elements and model fragments. The support goes further than other comparable methods, because, next to extracting domain models, MuDForM supports extraction of model elements for feature and context models (clarified in Sections 4.8 and 4.9). The metamodel covers the grammatical analysis concepts and their integration into the modeling concepts. The method steps enable the planning and organization of analysis and modeling activities. The explicit guidelines help to capture and dissipate analysis and modeling knowledge. Moreover, the method steps and guidelines are defined in terms of the metamodel. Practitioners may use the methodical support to bootstrap their modeling activity. Method developers may use the description of the support as an example of how to extend a modeling method with a part for bootstrapping a model from an input text.

As a second contribution, the paper presents the validation of the method in an industrial case study. This chapter reports on the phase from text to initial model. Researchers may use the case study to understand the methodical support. Practitioners may use it as an example of how to systematically analyze a text in order to create domain models.

4.2 Background: MuDForM Development

To understand the work that is reported in this chapter, we explain what we aim to achieve with MuDForM.

We envision software development as a process in which the involved people make decisions in their own area of knowledge, *i.e.*, *domain*, and in which those decisions are integrated, and finally result in a machine-readable specification.

We have presented the objectives for MuDForM in Section 1.1.3. Objective O7 comprises that a method should have a complete definition, which means it has a clear underlying model, *i.e.*, metamodel with clear semantics, a defined notation (view-

points and syntax), defined method steps, and guidance for the steps and viewpoints. We have explicitly defined a new metamodel (see Section 3.5.3), because no existing metamodel fulfilled all the objectives. We have based the method steps on the KISS method for object-orientation [96], which was already offering grammatical analysis, integrated with domain modeling and feature modeling, and extended it with explicit guidelines.

Objective O6 is the focus of this chapter, in particular the part that is about natural language. It goes as follows: Almost all people, including domain experts, use natural language to convey their knowledge and decisions. It is used in many documents that are relevant in a system development process. A specification method should **support the transformation of knowledge described in natural language into unambiguous models**. The purpose of this support is to minimize loss of semantics and increase mutual understanding in the communication between modelers and domain experts. The MuDForM vision (see Section 1.1.1) is to have method concepts that are close to human cognition. Natural language is a starting point for the method concepts, because it has evolved over thousands of years to support communication between people.

4.3 Research Methodology

This chapter addresses RQ3 of this thesis:

What methodical support can be given for the conversion of text into a domain-oriented model?

This section describes the research methodology we have applied to gather the results presented in this chapter. Based on the problem statement, the MuDForM vision, and the fact that we are working on an integral method, we define the following sub-questions:

(RQ3.1) What methodical support can be given for the conversion of text into ingredients of a domain-oriented model? The answer is given in Sections 4.5 and 4.6, in terms of grammatical analysis concepts, method steps, and guidelines, and validated through the case study from Section 4.7.

(RQ3.2) How should methodical support for extracting knowledge from text be integrated in a method that aims to produce domain-oriented models? The answer is given in Section 4.4, in terms of how modeling concepts and method steps fit with MuDForM's other modeling concepts and method steps.

4.3 Research Methodology

The development of MuDForM started with capturing experience from industry in a method vision and definition, followed by a phase in which the method is applied to cases, and improved through case findings. The approach can be seen as a combination of design science and action research in the way described by Iivari and Venable [83]. This chapter focuses on the action research aspects according to the description by Petersen *et al.* [121, 122], which inspired us to organize our study along the phases described below. The research method described below is based on the description in Section 3.2, but is limited to the part that is relevant to this chapter.

Diagnosis. Based on our experience with modeling, architecture, and model driven development in the past 25 years, we have defined a vision on software development and related method objectives (see Section 4.2), and defined an initial version of the method.

We have started to record and generalize our experiences, and work them out in detail since the start of the MuDForM research program in 2015. The method definition is available online [33].

Action planning. We have performed an SLR (see Chapter 2), which was derived from the same method objectives. From the SLR and the initial method definition, we identified topics needing further research, and the parts of MuDForM that needed further development. One of them is the methodical support for extracting models from natural language texts, *i.e.*, the topic of this chapter. Meanwhile, we contacted industry partners and explained the MuDForM vision, the MuDForM modeling process, and what a case study could do for them.

Action taking. We have defined the metamodel and steps for the identified gaps and added applicable guidelines from other approaches.

Case study. We defined the case-specific objectives together with the industry partner, and agreed on the timeline, and availability of people and documentation. We performed the case study and presented and explained the recorded model to the industry partner. They used the final model as the terminology in Gherkin test scenarios (*e.g.*, [150]), in order to make those scenarios unambiguous, and provided feedback. In Sections 4.8 and 4.9, we reflect on the case study from the perspective of the research questions and related work.

Reflection and action re-design. After completing the case study, we identified method gaps and flaws, and defined the required method changes, *i.e.*, revised the metamodel, method steps, and guidelines.

4.4 MuDForM Overview

This section presents an overview of MuDForM, which forms the framework for the method parts described in Sections 4.5 and 4.6.

MuDForM is defined according to the guidelines of Kronlöf [97], which has resulted in a method definition with the following ingredients: (i) a metamodel containing classes, activities, attributes, associations, specializations, and constraints, which define the modeling concepts and their relations, and (ii) a method flow containing steps, guidelines, and viewpoints, which guide the modeling process.

Section 4.4.1 explains the overall MuDForM modeling process; Section 4.4.2 the high level structure of a MuDForM model; and Section 4.4.3 the modeling concepts that form the link between the grammatical analysis and the model engineering phase.

4.4.1 MuDForM Modeling Process

Section 3.5.4 described the main steps of the MuDForM modeling process, as depicted in Figure 4.2. We elaborate here on the aspects specific to this chapter: The main steps are:

1. **Scoping:** the scope of the targeted model is specified by defining its purpose, its boundaries, and the input text that is selected from the knowledge source. The knowledge source is often an existing document, or a document that is created from interviews with (domain) experts.
2. **Grammatical analysis:** the input text is analyzed and transformed into a set of phrases with terms that are candidate elements for the model. The goal of this step is to maximize the knowledge elicitation from the source, and to make the resulting model traceable back to the input. This step is explained in detail in Section 4.5.
3. **Text-to-model transformation:** the specification spaces, which form the top-level structure of a model (see Section 4.4.2), are identified, and the phrases are transformed into model fragments, which are allocated to one of the specification spaces. This transformation is the transition from working with text to working with models, and is explained in Section 4.6.
4. **Model engineering:** the initial model is completed and inconsistencies are solved. Model engineering consists of a step to manage the dependencies

4.4 MuDForM Overview

between the specification spaces, and three steps for engineering the different types of specification spaces, *i.e.*, contexts, domains, and features. Appendix 3.C describes all the method step, and the complete MuDForM definition [33] contains more details.

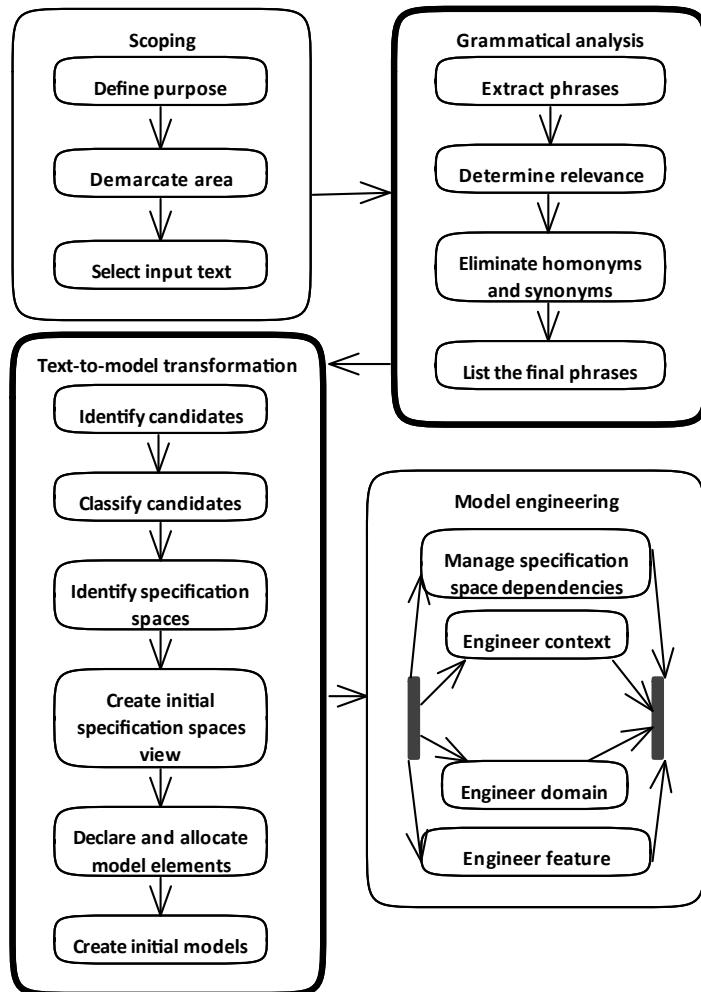


Figure 4.2: MuDForM method steps (UML activity diagram)

4.4.2 MuDForM Model Structure

The top-level structure of a MuDForM model consists of related *Specification spaces*, as depicted by the MuDForM metamodel fragment in Fig. 4.3. MuDForM uses *Specification spaces* (similar to UML packages) as containers for *Model elements*.

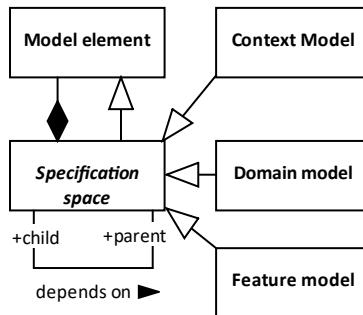


Figure 4.3: Model structure (UML class diagram)

In our notion of domain, a *Domain model* describes what can happen and what can exist in a domain. A *Feature model* prescribes what shall happen and what shall exist, and is expressed in terms of domain model elements (see Figure 4.1). *Context models* capture assumptions and knowledge about elements that are needed to specify domains and features, but that exist outside those domains and features. By defining the *dependencies* between the different specification spaces, models have no implicit semantics. Section 3.5.2 explains the notions of domain, feature, and context in more detail.

4.4.3 MuDForM Model Elements

MuDForM offers different types of *Model elements*. The type of *Specification space*, i.e., *Domain*, *Feature*, or *Context*, determines which types of *Model elements* are allowed, and what is their semantics. The three different *Specification spaces* all have concepts to specify state, behavior, and concepts to specify the relation between state and behavior. Moreover, almost all *Model elements* can have *attributes* and *generalization*, and can have *constraints* attached to them. The following elements are specific for engineering the domain model, and are thus possible output of the steps *Grammatical analysis* and *Text-to-model transformation*:

- **Domain activities** define what can happen in a domain. Instances of *domain activities* are actions, which represent atomic (state) changes in the *domain*.
- **Domain classes** define what objects can exist in a *domain*. Instances of *domain classes* are objects, which have a state that can be changed via actions.
- **Interactions** define which objects can participate in which actions. Objects change state when participating in an action.

We limit the explanation above to *domain models*, because *feature models*, and *context models*, are absent in the description of the case study in Section 4.7. All modeling concepts are explained in the complete MuDForM metamodel [33]. The overview of MuDForM from this section forms the context for the definition of the steps *Grammatical analysis* and *Text-to-model transformation*, which are explained in the next two sections.

4.5 Grammatical Analysis

This section describes the *Grammatical analysis* step as introduced in Section 4.4.1. This chapter only describes the method steps and concepts for *Grammatical analysis*. The sub-steps of *Grammatical analysis* are:

1. **Extract phrases** from the selected *Input sentences* and format them according to one of the following *Phrase types*:
 - An **interaction structure** expresses a change to one or more objects. The format is: *(subject) TO verb object (preposition object)**.
 - A **static structure** expresses a static relation between two terms. The format is: *noun HAS noun*, or *verb HAS noun*, or *verb HAS verb*.
 - A **state structure** expresses a property or type of a term. The format is: *noun IS adjective* or *verb IS adverb* or *noun ISA noun* or *verb ISA verb*
 - a **constraint** that expresses some condition to a term, typically formatted with propositional or predicate logic, like “if A then B”, or “for all A: B”. Temporal constraints are also possible, e.g., “after A then B”, or “within X seconds after A”.

Table 4.1 in Section 4.7.2 shows examples of these *Phrase types*. Typically, each *Input sentence* leads to one or more *extracted Phrases*, each containing two or more *Terms* (nouns, verbs, adverbs, adjectives). The *extracted Phrases* form

a decomposition of the original *Input sentence*, and are processed in the next method steps, in which they can change in terminology or structure due to *analysis decisions*.

2. **Determine the relevance** of each *extracted Phrase* from the perspective of the defined scope. Discard *Phrases* that do not fit the scope definition. Also check if *Phrases* are still valid in case legacy text is analyzed.
3. All *Phrases* are checked for **homonyms and synonyms**. These are then **eliminated** in consultation with the domain experts to assure that all *Terms* have exactly one meaning, and that all relevant meanings are covered by exactly one *Term*.
4. This results in a **list of final phrases** which is used as input for the model. The list contains all *extracted Phrases* that are marked as relevant and not discarded, and newly added *Phrases*, in which the identified homonyms and synonyms are replaced with the chosen *Term*.

During the analysis, *Analysis issues* can be raised for an *Analysis item*, i.e., a *Phrase*, or a *Term*. *Guidelines* can be used in the *decisions* made to solve an *Analysis issue*. Fig. 4.4 presents the metamodel for the steps *Grammatical analysis* and *Text-to model transformation*.

The complete MuDForM definition [33] presents more elaborate descriptions of these steps. The current version of the definition offers 28 guidelines for Grammatical analysis, organized per sub-step.

4.6 Text-to-Model Transformation

The transformation from text to an initial model consists of the following steps:

1. **Identify candidates**: Determine which *Terms*, i.e., nouns, verbs, adjectives, and adverbs, are a potential *Model element*.
2. **Classify candidates**: Select the *Term type* of each identified *Term*. The meta-class *Term type* in Fig. 4.4 shows the possible types, which are partially explained in Section 4.4.3.
3. **Identify specification spaces**: Identify *Contexts*, *Domains*, and *Features*. Each specification space should have an owner who is responsible for its content.

4.6 Text-to-Model Transformation

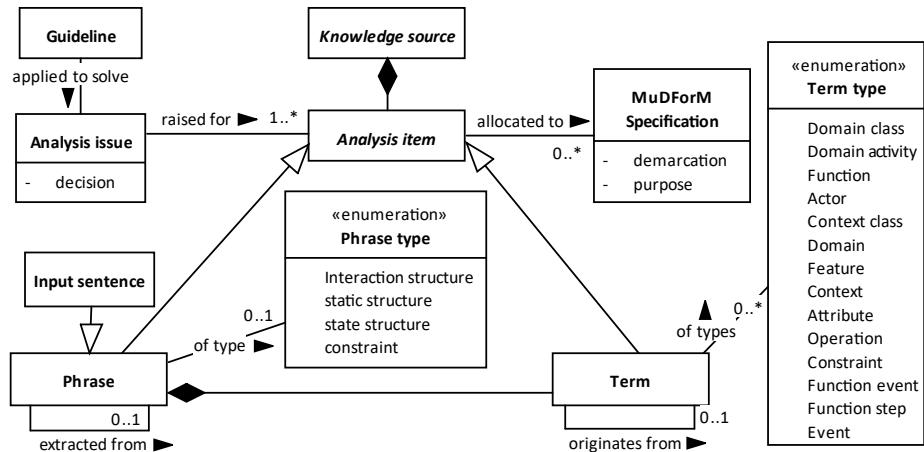


Figure 4.4: MuDForM concepts for grammatical analysis (UML class diagram)

4. **Create initial specification spaces view:** create a view with all the *Specification spaces*. Create *dependencies* and *compositions* between *spaces* if they are expected or already known.
5. **Declare and allocate elements:** create a *Model element* for each candidate *Term*, and put it in a *Specification space*. The *Model engineering* phase will reallocate a *Model element* if it was initially *allocated* incorrectly.
6. **Create initial models:** create a first version of the models from the list of final *Phrases*. All *interaction structure Phrases* become a relation between a behavioral element (*activity*, *operation*, *function*) and a *class*. All *static structure Phrases* become an *attribute* of the subject, and the attribute type corresponds with the object of the *Phrase*. All *state structure Phrases* become a *generalization* relation between the subject and the nominal part of the *Phrase*. For the *constraint Phrases* it varies; they can become *invariants*, *preconditions*, *postconditions*, or a temporal ordering in the *Object lifecycle* of a *Domain class* or *Function*. The initial model is the input for the *Model engineering* step, which offers support for making the model complete and consistent.

The complete MuDForM definition [33] presents more elaborate descriptions of these steps. The current version of the definition also offers 19 guidelines for Text-to-model transformation, organized per sub-step.

4.7 A Case Study: System Behavior Description of the History Feature

This section presents the results from a case study in which we, together with domain experts from a high-tech company, applied MuDForM to a system feature, described in a so-called system behavior description (SBD).

The high-tech company develops and produces products and services for printing and workflow management. The development process for one of their product lines uses SBDs to specify the behavior of product features. SBDs are the result of discussions and negotiations between product managers, developers, and testers, and are used throughout the development and test process. Currently, SBDs contain mostly natural language text. The case in this section is about one of in total 90 SBDs, namely the SBD of the History feature, which describes the system behavior for the management of completed print jobs.

The goal of the case study is to evaluate MuDForM support for transforming textual specifications into initial models. The rest of this section focuses on the phase from text to initial domain model. Chapter 5 presents more examples of MuDForM's *Grammatical analysis* step.

4.7.1 Case Study Overview and Execution

The case study was executed as a collaboration between a MuDForM researcher, a modeling expert from the customer that guards the usability of the model, and several domain experts that help to solve unclarities in the SBD text.

During the modeling process, the most important decisions were recorded and some of them are used to explain the results. We show examples of the resulting model to illustrate how the method is applied. The complete model is not publicly available due to intellectual property rights. But a more elaborate excerpt of the case is available online [32]. The next two sections discuss the execution of the steps *Grammatical analysis* and *Text-to-model transformation*, as explained in Sections 4.5 and 4.6.

4.7 A Case Study: System Behavior Description of the History Feature

4.7.2 Grammatical Analysis of the History SBD

The step *Grammatical analysis* starts with *Extracting phrases* from the input text. We follow the guideline *Use a structure to separate input sentences*, as described in [33], and created Table 4.1, which shows the *Sentences* that are *selected* from the History SBD, and the *Phrases* that are extracted from them. In each row, the first column contains the *Input sentence*, and the second column has one or more *extracted Phrases*. After the extraction, we *Determined the relevance* of each *Phrase* together with the domain expert, and *Eliminated homonyms and synonyms* from the *Phrases*. The last column explains the *decisions* made for the raised *Analysis issues*, possibly with a reference to the guideline on which the decision is based. After that, we *List the final phrases*, which are the *emphasized Phrases* in the table. For clarification, we explain one row (highlighted in gray) of the table: “Therefore, jobs that are too old will automatically be removed from the History”. First, we *extracted* two *Phrases*: “TO remove job from History” and “Job IS too old”. We already had “to Delete”. So we asked what the difference is with “to Remove”. The domain experts said they are synonyms, and chose the *Term* “to Delete”. Following the guideline *Detect type of adjectives and adverbs*, we asked what kind of thing “too old” is. The involved domain experts could not immediately provide clarity, and kept discussing it. So, we applied the guideline *Postpone too long analysis discussions*, and kept the information as is. We postponed the discussion to the *model engineering* phase, which will solve the issue, because then the discussions are more directed due to the use of specific viewpoints like the object lifecycle, and model engineering criteria like (data) normalization.

Table 4.1: Selection of the grammatical analysis

<i>Input sentence</i>	<i>Extracted (including final) Phrases</i>	<i>Decisions (including new final Phrases)</i>
When a print job is completed, it will be archived in the so-called “History”.	TO complete job. TO archive job in History.	To archive and to move are synonyms. Chosen: to move.
The History is a job store that will be used as a local temporary job store and is not intended for long term archiving purposes.	History ISA job store. TO use History as local temporary job store. TO intend History for purpose.	To intend and to use are ignored because of guideline <i>Ignore intention phrases</i> .

Chapter 4. From Text to Model with MuDForM

<i>Input sentence</i>	<i>Extracted (including final) Phrases</i>	<i>Decisions (including new final Phrases)</i>
Only jobs that have been completed will end up in the History.	TO complete job. Job TO end up in History.	To end up is not a domain activity. “job is in History” is a state after “to archive”. Chosen: <i>to move job from job store to job store</i> .
Proof prints initiated from the waiting room and system jobs will not end up in the History when completed.	TO initiate proof print from waiting room. <i>System job ISA job.</i>	To initiate is considered out of scope. (It is in the scope of Job scheduling.), but <i>Proof print ISA job</i> .
Also jobs that have been aborted or deleted will not end up in the History.	<i>To abort job.</i> <i>To delete job.</i>	
The Settings editor provides functionality to clean up the History at specified time periods. The following time periods can be specified: One day, One week, One month, Forever.	TO clean up History at time period. TO specify time period. One day, one week, one month, forever ISA time period.	Use retain period instead of time period. Furthermore, it is the retain period of the History which is specified, giving <i>TO specify retain period of History</i> , and <i>TO clean up History at retain Period</i> . One day, one week, one month, forever are possible values of retain period.
Jobs that have been longer in the History than the specified time period for the automatic cleanup are removed from the History.	<i>History HAS jobs.</i> To specify time period.	
Therefore, jobs that are too old will automatically be removed from the History.	TO remove job from History. <i>Job IS too old.</i>	To remove and to delete are synonyms. Chosen: to delete. Following the guideline <i>Detect type of adjectives and adverbs</i> , we asked what kind of thing “too old” is. We did not get a clear answer. So, we kept it.
If the History is disabled, new completed jobs will be removed from the system, so they will not end-up in the History.	<i>TO disable History.</i> TO complete job. TO remove job from system.	System and controller are synonyms. Chosen: controller. Giving: <i>Controller HAS History.</i>
A job can be reprinted from the History by copying them from History to waiting room.	TO reprint job from History. <i>TO copy job from History to waiting room.</i>	Is “reprint” the activity or “copy”? Answer: To copy. Reprint is the intention. And, what is a waiting room? Answer: <i>Waiting room ISA job store</i> . Giving: <i>TO copy job from job store to job store</i> .

4.7.3 Text-to-Model Transformation for the History Domain Model

This section discusses the creation of the initial models from the results of the *Grammatical analysis* step.

The first steps are *Identify candidates* and *Classify candidates* as described in Section 4.6. In this case, every *Term* becomes a *domain class* if it is a noun, or a *domain activity* if it is a verb. “Too old” is an adjective, which probably indicates a possible value of a context *class*. But we classified it as a *class*, for reasons explained in the previous section.

The next step is to *Identify the specification spaces*. We used the guideline *Begin with one context, one domain, and one feature*, because the case was relatively small, and there were no existing *Specification spaces*. This led to the specification spaces History domain, History feature, and Context.

The next step is to *Declare and allocate the model elements* to the *Specification spaces*. We have *allocated* all *Terms* to the History domain by following the guideline *In case of doubt, put a candidate term in the domain*, except for Retain period and its possible values, which are *allocated* to the Context.

The last step is to *Create the initial models* from the *Phrases* and *Terms*, which resulted in Figures 4.5 and 4.6. (The depicted *interaction view* is compliant with the UML metamodel [120], because classes and activities are both classifiers and, as such, can have association relations.) All the emphasized *Phrases* are present in the diagrams. The more elaborate report of the case study contains more *Phrases* (see [32]).

4.8 Discussion

In the following, we reflect on the research questions and discuss the findings from the case study. We first discuss the support for grammatical analysis (RQ3.1 from Section 4.3) and how it fits in the rest of MuDForM (RQ3.2).

Figure 4.2 presents the steps for the conversion of a text into an initial model. The steps form a clear structure on how to organize this process. Each step can be planned and executed accordingly. However, during the case study we found out that in practice it is easier to first focus on the basic elements of the domain model, and not on the constraints and other aspects of the feature model. This means that there are at

Chapter 4. From Text to Model with MuDForM

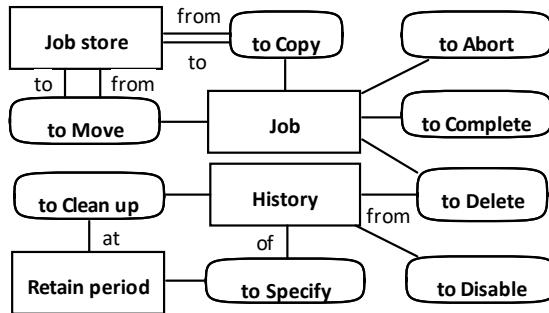


Figure 4.5: Initial model of the History domain: Interaction view (UML notation)

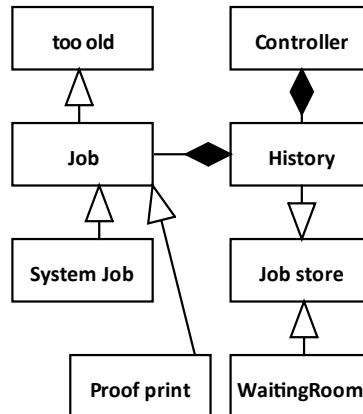


Figure 4.6: Initial model of the History domain: Static view (UML notation)

least two iterations. The first iteration focuses on the extraction of static phrases and state phrases, and interaction phrases that have no actor, *i.e.*, most phrases starting with “TO”. After that, conduct model engineering until the domain model is stable. The second iteration is about the extraction of interaction phrases with actors and about constraint phrases, which can immediately be rewritten to match the created domain model. This second iteration is also a validation of the created domain model. Namely, all the constraints should be expressed in terms of the domain model and possibly context model. If not, then either the constraint phrase is unclear or incorrect, or the domain model must be adapted. Thanks to this insight, we have added

the guideline *First do the domain model, then the feature model* to the MuDForM method flow [33]. We also found that the usefulness of this and other guidelines depends on the purpose of the used input text, *i.e.*, describing system behavior. If the text's purpose is different, *e.g.*, a set of requirements, a process description, or a pure explanatory statement, then the usefulness of the guidelines might shift. This is even more apparent when other than natural-language specifications are analyzed. Further research is needed to take this aspect into account.

Section 4.4 presents how the sub-steps of the *Text-to-model transformation* fit into MuDForM. The partial metamodel in Figure 4.4 addresses how the concepts fit. All the possible values for *Term type* and *Phrase type* correspond to classes and relations from the rest of the MuDForM metamodel [33]. The fact that we do not have all the classes from the MuDForM metamodel as a possible *Term type* is due to two pragmatic reasons. First, we have only put classes in the metamodel that we have actually used in one of our past modeling projects. Second, the main purpose of the model engineering phase, which comes after the phase described in this chapter, is to bring preciseness, consistency, and completeness to the model. The modeling environment is more suitable to do that than the natural language environment. However, it might be possible that we change the possible *Phrase types* and *Term types* due to new insights in later projects.

The above discussion only pertains to the integration of grammatical analysis in MuDForM. We think that similar constructs should be applied when the support for grammatical analysis is integrated in other modeling methods. The following describes the general aspects of such an integration.

On the metamodel. The presented metamodel (Fig. 4.4) has concepts that are specific for grammatical analysis, which are related to the MuDForM modeling concepts via the metaclasses *Phrase type* and *Term type*. For another method, other *Phrase types* and *Term types* may be used. For example, most domain modeling methods do not have a primary modeling concept for specifying behavior, like the *Domain activity* concept in MuDForM. They just model classes, attributes, and relations between classes, and often capture behavior in class operations or in generic data-oriented operations like create, update, and delete.

On the notation. The case study uses tables and plain text for the notation. MuDForM itself does not prescribe a specific notation. When grammatical analysis is integrated with another modeling method, it is possible to choose a notation that is close to the existing notation of that modeling method.

On the method steps. The four main steps of the MuDForM method flow (Fig. 4.2 cf. page 117) can be generalized into: Scoping, Discovery and Elicitation (for capturing specific knowledge from a knowledge source), Switch to modeling, and Model engineering. In general, the *Grammatical analysis* step can (partially) replace the Discovery and Elicitation step from another method. Such other may have different modeling concepts, which implies that the step *Text-to-model transformation* may also differ.

On the guidelines. Guidelines can be reused as is. But if other *phrase types* and *Term types* are identified, which is very likely, the guidelines might must be adjusted too.

4.9 Related Work

The SLR on domain-oriented specification techniques of Chapter 2 identified several approaches that extract models from text [129, 7, 96, 47, 138, 82, 161, 2]. However, none of them provides a metamodel for grammatical analysis.

MuDForM is based on the KISS method [96], which is the only approach from the mentioned SLR with an explicit phase and concepts for grammatical analysis, and a distinction between domain and feature. It however does not provide a metamodel, fine-grained method steps, or guidelines.

Abirami *et al.* [2] give guidelines for conceptual modeling of nonfunctional requirements. They overlap with the MuDForM guidelines for extracting phrases, but do not distinguish an explicit intermediate step for grammatical analysis.

Arora *et al.* [7] present an approach for extracting domain models from natural-language requirements. They give guidelines for creating classes, associations, and attributes from sentences. Some of those guidelines are also present in MuDForM. The main difference is that they do not distinguish behavioral concepts, such as the domain activity concept in MuDForM, and do not distinguish between domain, feature, and context.

Elbendak *et al.* [47] describe an approach for automatic generation of class diagrams from use case descriptions. They solved the issue of multiple binary associations representing one action by using n-ary associations. However, they too do not distinguish between domain, feature, and context, and let the creation of a class in the target model depend on the number of occurrences that its corresponding noun has in the text. The same holds for the paper from Sagar and Abirami [138], which reuses

and improves many of the rules given by Elbendak, and introduces a clear distinction between a strict text-to-model transformation and suggesting model candidates. However, it is limited to models that can be fully captured in standard UML class diagrams. Ibrahim and Ahmad [82] introduce a tool for the automatic extraction of class diagrams from textual requirements, which follows many of the rules from the other papers. Compared to MuDForM, these approaches loose semantics in the transition from text to model, regarding the different specifications spaces (domain, feature, context) and the way behavior is captured. Repairing this semantic loss in the model would require to go back to input text to perform the grammatical analysis anyway.

Although we are open to automating part of the text-to-model process, we think that the participation of domain experts in the grammatical analysis process is essential. They do not only provide missing information and help to eliminate homonyms and synonyms, but often feel more comfortable discussing natural language sentences than discussing graphical models. The paper from Hoppenbrouwers *et al.* [161], which is based on the KISS method [96], makes a claim for partially automating the text-to-model phase, such that domain experts are still actively involved via natural language. MuDForM also supports the involvement of domain experts via the verbalization of models in natural language, which is also addressed by Proper *et al.* [129], Kristen, [96], and Hoppenbrouwers *et al.* [161]. The method steps List the final phrases, Identify candidates, and Create initial models can easily be automated. In an experiment, we have tried to automate the step Extract phrases. However, we observed that this leads to an abundance of irrelevant phrases, which cost more effort to discard than the time it saves compared to doing the extraction manually. The use of machine learning techniques, combined with large language models, could improve the automation of Grammatical analysis. We did not study this possibility yet.

There are more papers about the transformation of text into models, *e.g.*, the 20 primary studies in the SLR of Yue *et al.* [172]. They all have in common that they focus on the transformation from text to model, but do not consider an explicit model engineering phase with similar main principles as MuDForM. For example, they do not separate domain, feature, and context, and they do not have modeling concepts for integrating static and behavioral properties in a model. However, some studies might contain useful guidelines for the text-to-model phase of MuDForM, which we will investigate.

4.10 Conclusion and Future Work

This chapter describes the MuDForM methodical support for converting a text into an initial model and reports on an industrial case study.

In doing so, we observe that the defined metamodel and method steps are quite mature, as we did not detect relevant knowledge from the case text that we could not capture. But the guidelines are far from complete, because we easily found new ones during the relatively small case study.

The results from our study fill an important gap in the state of the art, which to the best of our knowledge lacks in providing methodical support in the first place. It lays the foundation for our future work on building a validated and reusable set of guidelines, for which we foresee the following: (i) A community that actively validates, identifies, and manages guidelines. (ii) A literature review to find, and analyze guidelines from natural language processing approaches, *e.g.*, the primary studies from [172] *et al.*, to possibly integrate in the *Grammatical analysis* step of MuDForM.

To facilitate industrial adoption, we plan to create a MuDForM handbook for practitioners. It is a replacement of the document that currently contains the method definition [33]. We are investigating the requirements and possibilities for a modeling tool that supports MuDForM, in order to replace MS Word and Enterprise Architect [152]. Such a tool could benefit from machine learning techniques to (partially) automate the Grammatical analysis and Text-to-model transformation.

5 Specifying Features in Terms of Domain Models: MuDForM Definition and Case Study

“De-do-do-do, de-da-da-da
Is all I want to say to you
De-do-do-do, de-da-da-da
They’re meaningless and all that’s
true”

De Do Do Do, De Da Da Da,
The Police, 1980

This chapter is based on the publication  R. Deckers and P. Lago “*Specifying Features in Terms of Domain Models: MuDForM Definition and Case Study*”, Journal of Software: Evolution and Process, Wiley, 2023 [38].

This chapter addresses RQ4 of this thesis.

Context. To enable the people involved in a software development process to communicate and reason close to their area of knowledge, we are investigating and engineering a method that formalizes and integrates knowledge of multiple domains into domain models and into specifications in terms of those domain models. For this purpose, we have previously defined a vision and a list of method objectives (see Section 1.1), and an initial version of the method. We performed an SLR, which concluded that there is no methodical support for using domain models as the terminology in other specifications (see Chapter 2).

Goal. Provide methodical support for creating feature specifications that are expressed in terms of domain models and context models, such that those feature specifications are self-contained and unambiguous.

Method. We follow an action research approach, with phases for diagnosis, action planning, and action taking. During the evaluation phase, we performed a case study to validate how well the method helps in the specification of processes. The case study pertains to the formalization of the ISO26262 standard for functional safety in the automotive domain. During the specifying learning phase, we have extended our method with concepts, steps, and guidelines for grammatical analysis, for formalization of constraints, and for the specification of processes.

Result. We found that MuDForM is suitable to systematically formalize processes described in natural language, such that the resulting process models are fully expressed in terms of domain concepts and concepts from outside the domains and processes of interest. We discuss how the method part of this chapter can be used to bridge the gap between domain models and the feature models of some other approaches, *e.g.*, FODA. The case-specific results are the unambiguous specification of a part of the ISO26262 processes.

Contents

5.1	Introduction	134
5.1.1	Problem Statement	135
5.1.2	Contribution and Audience	136
5.2	MuDForM Vision and Terminology	137
5.2.1	MuDForM Objectives	138
5.2.2	Domain and Feature	138
5.3	Research Methodology	139
5.4	MuDForM Foundation	142
5.4.1	MuDForM Modeling Process	143
5.4.2	MuDForM Model Structure	146
5.4.3	MuDForM Specification Elements	147
5.5	Feature Modeling	151
5.5.1	From Modeling Initiation to Initial Model	151
5.5.2	Engineer Feature	154
5.6	A Case Study: Modeling the Processes of ISO26262	157
5.6.1	Introduction to the Case	158
5.6.2	Case Study Overview and Execution	159
5.6.3	From Modeling Initiation to Initial Model	160
5.6.4	Specify Function Lifecycles	167
5.6.5	Specify Function Signatures	173
5.6.6	Specify Feature Structure	173
5.7	Discussion	174
5.7.1	Support for Domain-based Specifications	175
5.7.2	Bridging the Gap between Feature Trees and Domain Models	178
5.7.3	Feature Modeling is Part of MuDForM	180
5.7.4	Reflection on Using UML for MuDForM	182
5.7.5	Reflection on Case-specific Objectives	183
5.8	Threats to Validity	184
5.9	Related Work	186
5.10	Conclusion and Future Work	187
5.A	Guidelines Pertaining to Feature Modeling	189

5.1 Introduction

Since the 1980s, several methods for domain modeling have been proposed [143, 148, 96]. Domain models can be used in the development of a system in various ways, *e.g.*, as a basis to derive a software design [53], or as the terminology for other specifications, like a requirements specification [58] or a functional specification [96].

MuDForM is an integral domain-oriented modeling method that not only provides support for the creation of domain models, but also for the creation of models that are defined in terms of a domain model, called *domain-based models* (see Figure 5.1). MuDForM provides steps, concepts, and guidelines for an analysis and modeling process, which starts with a knowledge source, like a (domain) text or (domain) expert. This chapter explains the part of MuDForM for domain-based feature modeling, and its application in a case study.

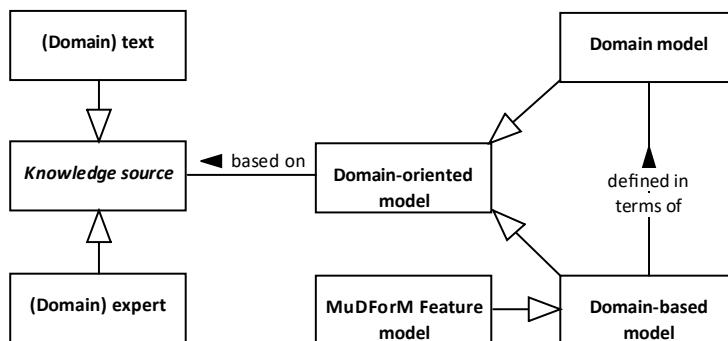


Figure 5.1: Context of a domain-based model (UML class diagram)

The rest of this section describes the problem, our contribution, and target audience. Section 5.2 explains in more detail what we try to achieve with our research, and how this chapter fits. Section 5.2 also clarifies what we mean by the terms domain and feature. Section 5.3 explains the research methodology. Section 5.4 describes the foundation of MuDForM, which is required to understand the support for specifying features in terms of a domain model, defined in Section 5.5. These two sections are based on the complete MuDForM definition (metamodel, method steps, viewpoints, and guidelines) [33], of which the phase from input text to initial model is explained in Chapter 4. Section 5.6 reports on a case study in which MuDForM is applied to formalize process specifications of the ISO26262 standard [84]. Section 5.7 reflects on MuDForM's feature modeling support via the results from the case study. Section 5.8

discusses the threats to validity of the work reported in this chapter. Section 5.9 discusses related work, and Section 5.10 concludes the paper and presents suggestions for future work.

5.1.1 Problem Statement

The systematic literature review (SLR) in Chapter 2 reports that methods for domain-oriented specifications are mostly limited to creating a domain specification (DS), like a domain-specific language (DSL) or domain model (DM), and do not incorporate steps and guidance for applying a created DS. There are many publications about the usage of a specific DS, *e.g.*, the security domain model of Firemsmith [58], and the many examples from van Deursen *et al.* [43]. However, they only provide support for the DS at hand in the form of an underlying model and a notation, and mostly do not provide detailed steps and guidelines. The KISS method for object-orientation [96] appears to be the only approach that has an explicit modeling phase and concepts for applying a domain model in functional specifications. But the KISS method does not provide a metamodel, fine-grained method steps, and detailed guidelines. **With MuDForM, we aim to offer methodical support, including steps and guidelines, for the creation of DMs, as well as for the creation of models in terms of DMs, called domain-based models.** Section 5.2.1 explains in more details our motivation for the work presented in this chapter.

Although we do not know of literature that explicitly states the need for methodical support for applying a DS in making other specifications, we found several papers related to this topic. There is literature about evaluating the usability of created DSLs, *e.g.*, Barivšić *et al.* [13, 14, 12]. They state that the evaluation of DSLs does not happen often. Furthermore, Gabriel *et al.* [65] state that the community in software language engineering does not systematically report on the experimental validation of the languages it builds. So, it is not recorded if a created DSL is usable in practice. Gray *et al.* [74] write that “poor documentation and training [of a DSL]” is one of the 10 reasons why DSLs are not used more frequently in industry. Völter [165] states that it is important to communicate to users how a specific DSL works, and observes that in nonscientific domains, domain experts, although they are the targeted modelers, are often not the ones that make models with the DSL. Instead, the domain expert pairs up with the DSL developer to apply the DSL, or the DSL developer does all the specification based on discussions with the domain expert. We see this as a symptom of that **it is not straightforward for a domain expert to make models with a DSL that is created by someone else.**

We, the authors, have observed the same phenomenon regarding DS developers and domain experts in many industry projects. A DS is defined, but only the DS developers know its exact semantics and know how to use it as a language for other specifications. The targeted users lack this knowledge, and hence, might not apply the DS correctly in making specifications with them. Without methodical knowledge about the use of a DS, they also do not know what steps to take, how to make good modeling decisions, what information to retrieve from domain experts or from the input text, or how to organize the specification process. The lack of methodical support, in particular clear steps and guidelines, leads to an unpredictable and difficult to organize specification process, which results in specifications of which the quality is only controllable by validation, and not by design. We think that this **lack of methodical guidance is one of the most important shortcomings in the literature on domain-oriented methods**, which is subscribed by Gray *et al.* [74] and Barivšić *et al.* [12].

The SLR (see Chapter 2) also concluded that none of the found methods on domain-oriented modeling has a complete method definition (*i.e.*, covering underlying model, notation, steps, and guidance). Some approaches provide a metamodel and a notation, and others provide high level steps, sometimes guidance, and sometimes the use of an existing language like UML. The lack of an underlying model makes it hard to have an unambiguous well-defined interpretation of models, and difficult to separate the semantics from the syntax. If there is a language defined, but no steps and guidance, then modelers must be experienced in the language. Otherwise, the specification process becomes unpredictable.

MuDForM is a domain-oriented method that supports the formalization of a piece of (prescriptive) text into unambiguous models. This chapter focuses on the support for making specifications in terms of domain models, and how it fits the rest of MuDForM, because support for specifying domain-based models is hardly covered in the existing literature.

5.1.2 Contribution and Audience

This chapter has two main contributions. First, it presents methodical support for using domain models as a coherent vocabulary to make feature specifications, such that that support is an intrinsic part of our method, which also covers the creation of domain models. Practitioners may use the support as guidance for applying a domain model as a structured vocabulary for other specifications. The benefits of the support are the predictability and the manageability of the specification process,

5.2 MuDForM Vision and Terminology

because the method steps (see Figure 5.6 and 5.7) form the basis for planning the specification activities. Furthermore, the provided guidelines (see Appendix 5.A) help to make decisions throughout the steps of the analysis and modeling process. The latest version of the complete MuDForM definition is available online [33].

Method developers may use the description of the support as an example of how to extend a domain modeling method with a part for using a domain model to create other specifications. This facilitates bridging the gap between the literature that considers a domain model as a decomposition of a product line, as in FODA [90], and the literature that considers a domain model as a structured vocabulary to define other specifications, like the security domain model of Firemsmith [58] and the many examples from van Deursen *et al.* [43]. As such, this contribution might also help researchers and method developers to understand the mentioned literature gap. It must be clear that the methodical support in this chapter is an integral part of MuDForM, and hence, is most suited when the domain specification that is used to create the domain-based specification, is also created with MuDForM.

As a second contribution, the paper presents the validation of the methodical support in an industrial case study, which covers both the creation of a domain model (published by Khabbaz Saberi [93]), and the creation of a domain-based feature model (the latter being the focus of this chapter). Researchers may use the case study to understand the methodical support. Practitioners may use it as an example of how to create feature specifications in terms of a domain model, or, as in the case study in Section 5.6, process models.

Overall, this chapter gives guidance to people developing (domain-oriented) specification and modeling methods. It shows how to define and present a method, in particular the modeling process and the guidance for modeling decisions. It supports practitioners in applying domain models and DSLs in order to formalize prescriptive behavior specifications, like process specifications, and scenarios of use cases or user stories.

5.2 MuDForM Vision and Terminology

To understand the reason behind the work in this chapter, we explain what we aim to achieve with MuDForM (Section 5.2.1), and what we mean by the concepts *domain* and *feature* (Section 5.2.2).

We envision software development as a process in which the involved people make decisions in their own area of knowledge, *i.e.*, *domain*, and in which those decisions are explicitly integrated, so that they finally result in a machine-readable specification (see also Section 1.1.1). That is why our research focuses on an integral method for creating DMs, for using DMs as a language to create other (domain-based) specifications, and for integrating multiple DMs and domain-based specifications. The ultimate goal is that a system is completely defined in domain-oriented specifications, and if other kinds of specifications are used, that they are also explicitly integrated.

5.2.1 MuDForM Objectives

Based on our vision and experience with domain modeling, architecture, and model driven development, we have defined a set of objectives for the development of MuDForM (see Section 1.1.3). We will refer to the objectives in Sections 5.4 and 5.5 when applicable. Not all objectives are addressed, because this chapter focuses on a part of MuDForM, *i.e.*, the making of prescriptive domain-based models, as stated in Objective O4.

5.2.2 Domain and Feature

This section is an extension of Section 3.5.2, which describes what we mean by the terms domain and feature.

Our notion of feature model differs from the definition in some other literature [90], where a feature model shows the relation between several features of a product line. Often, features correspond with high-level functions of a system, and consequently, a feature model can be seen as a functional decomposition of the product line.

MuDForM uses the feature structure viewpoint, as explained in Section 5.5.2.1, to show the decomposition of a feature in functions, or sub-features. A MuDForM feature model is, just like a MuDForM domain model, described in multiple views. The feature structure viewpoint is similar to what some other methods call a feature model.

5.3 Research Methodology

This chapter addresses RQ4 of this thesis:

What methodical support can be given for the specification of a feature, such that it is defined in terms of domain models?

This section describes the research methodology we have applied to gather the results presented in this chapter. Given the problem statement, the above definitions of domain and feature, and the fact that we are working on an integral method, we aim to address the following sub-questions:

(RQ4.1) How should methodical support for making feature specifications be integrated in a method that also supports the creation of domain models? The answer is provided in Section 5.4, in terms of how the modeling concepts and method steps fit with MuDForM's other modeling concepts and method steps.

(RQ4.2) What methodical support can be given for the specification of a feature, such that it is defined in terms of domain models? The answer is provided in Section 5.5, in terms of modeling concepts, method steps, and guidelines for the steps.

The development of MuDForM started as a project in which experience from industry practice is captured and made tangible in a method vision and definition, followed by a phase in which the method is applied to cases and adjusted based on case findings. The approach can be categorized as action research according to the description by Petersen *et al.* [121], which we use as the basis for explaining our study, resulting in the phases of Diagnosis, Action planning, Action taking, Evaluation, and Specifying Learning. The research method described below is based on the description in Section 3.2, but is limited to the part that is relevant to this chapter.

Diagnosis. Based on our experience with modeling, architecture, and model driven development, we have defined a vision on software development, which is mentioned in the introduction of Section 2. The vision is further refined in the form of the method objectives, defined in Section 5.2.1. When we started to work on MuDForM, we already knew of specification approaches from several books on domain modeling and domain-specific languages [53, 96, 92, 164, 107, 61]. We observed that those books did clearly not address all the MuDForM objectives. So, we performed a

systematic literature review (see Chapter 2) with the same objectives as starting point. We identified some clear gaps in the existing literature, which should be bridged to achieve the objectives. Especially, processing natural language to create models, dealing with multiple domain models, and using DMs and DSLs to make other specifications, are hardly addressed in existing literature. The latter corresponds to objective O4 and is the main topic of this chapter.

Action planning. As concluded in the mentioned SLR, the only method that comes close to meeting the MuDForM objectives is the KISS method [96]. That is why we used it as the starting point for the definition of MuDForM. We use the following characteristics of the KISS method: 1) the separation between domain model, feature model, and context, 2) having activities as first-class citizens, including the relation between activities and classes, 3) the modeling of lifecycles of domain classes and functions in a process algebra style, and 4) the use of natural language processing in model creation, and model validation. We have defined an explicit method model, a detailed method flow, and guidelines, which were lacking in the existing literature on the KISS method.

We have chosen UML for the representation of the metamodel and method flow. We also considered using metamodels like Ecore [155], MOF [119], and GOPPR [113]. The advantage, that these are more strictly and precisely defined than UML, did not outweigh the familiarity of UML and the availability of good tooling. Moreover, we only use a small subset of UML's modeling concepts in the MuDForM metamodel and method flow. For the sake of readability, we have chosen to use natural language for the specification of the guidelines. However, we have expressed the guidelines in terms of the MuDForM metamodel as much as possible.

The first step is to consolidate our experience in an initial method definition. After that, we foresaw three tracks: 1) incorporate method ingredients from other methods, especially the ones found in the SLR, 2) let peers review the method definition to comment on the metamodel and suggest method steps and guidelines, and 3) apply the method in practice via case studies to improve and extend the metamodel, method steps, and guidelines.

Action taking. We defined the initial versions of the metamodel, method flow, and guidelines. The metamodel and method flow are the result of our 25 years of experience in creating, using, and managing domain models, mainly with UML [120]. We have created and used domain models in the context of domain analysis, require-

5.3 Research Methodology

ments engineering, functional design, software architecture, process modeling, and test specification, in various business domains. We have started to record and generalize our experiences, and work them out in detail since the start of the MuDForM research program in 2015. We actively manage the metamodel¹ and the method flow in a UML model with the tool Enterprise Architect [152].

Meanwhile, we looked for possibilities in industry to apply MuDForM. We contacted industry partners and explained the MuDForM vision, the MuDForM modeling process, and what a case study could do for them. We defined the case-specific objectives together with the industry partner, and agreed on the timeline and availability of people and documentation. We explicitly chose not to prescribe a notation for MuDForM, because we want to stay close to a notation that is familiar to the partner.

Evaluation. Together with customer experts, we applied MuDForM to a subset of the ISO26262 standard [84], which led to the results described in Section 5.6. Typically, each analysis or modeling step started with an explanation of the relevant viewpoints and guidelines to the involved experts. We recorded examples of major analysis and modeling decisions, which included the identification of used guidelines and the creation of new guidelines. We did not record all decisions for the sake of manageability and project speed. Namely, the industrial partner had constraints regarding the duration and involvement of personnel. Logging all decisions would have decreased the process pace too much. During the case study, we also refined the method steps and their descriptions, and came up with new viewpoints and guidelines, partially based on feedback of the involved people. Moreover, the recorded analysis and modeling decisions were regularly discussed and (re)specified. We also asked people to review the method definition, *i.e.*, the metamodel, method flow, and guidelines, on which we reflect in Section 5.7.1.

After the modeling process, the recorded model was presented and explained to the employees of the industrial partner. This involved verbalizing the model in natural language, which makes it easy to check if the model corresponds to the original text. We were not actively involved in the usage of the model at the industrial partner's site. However, we received feedback on the usage of the model, which is discussed in Section 5.7.5 and verified by the industrial partner. We reflected on the case study from the perspective of the research questions and related work on feature modeling, which is addressed in Sections 5.7.1 through 5.7.4.

¹How the metamodel is constructed is a research topic in itself, and topic of a future paper. The latest version is available online [33].

Furthermore, to be clear about the scope, this chapter does not report on all the details of the above-mentioned evaluation activities, *i.e.*, all the modeling steps and decisions of the case study, and the decisions regarding the adaptations to the MuDForM method definition. It reports the execution of the method steps and the resulting feature model of the case study, and the method definition that corresponds with that execution.

Specifying Learning. We have received suggestions for guidelines from peers and, after acknowledging their validity and usefulness, fit them into the method flow and other guidelines.

After completing the case study, we first consolidated the method changes, *i.e.*, made adjustments to the metamodel, method steps, and guidelines. During the writing of this chapter, we also refined the definitions of method steps and guidelines, which we manage in a UML model as mentioned in the description of the *Action taking* phase above.

5.4 MuDForM Foundation

This section describes the foundation of MuDForM, which forms the framework for the feature modeling part described in Section 5.5.

MuDForM is based on the KISS method [96]. The major extensions that this chapter covers are the guidance for grammatical analysis for features, the guidance for identification and specification of elements in feature models, the method steps for feature modeling, and extra viewpoints for feature models.

MuDForM is defined according to the guidelines of Kronlöf as explained under objective O7 in Section 1.1.3. This has resulted in a method definition [33] with the following ingredients: (i) a metamodel containing classes, attributes, associations, specializations, and constraints, which define the modeling concepts and their relations, and (ii) a method flow containing steps, guidelines, and viewpoints, which guide the modeling process. Furthermore, the case study (Section 5.6) uses UML [120] syntax and semantics to the extent that it fits the MuDForM metamodel, to benefit from its familiar notation and available tool support. An explanation of the diagrams' syntax is given in cases that deviate from standard UML notation.

Section 5.4.1 explains the overall MuDForM modeling process. The method part about the phase from text to initial model is explained in Chapter 4. Section 5.4.2 explains how features models relate to other parts of a MuDForM compliant model. Section 5.4.3 concludes the MuDForM foundation with an introduction to the different types of specification elements that MuDForM offers for the specification of features.

5.4.1 MuDForM Modeling Process

This section is an extension of Section 3.5.4. Figure 5.2 shows the high-level steps of the MuDForM modeling process². Feature modeling is present in all four steps, and is explained in detail in Section 5.5. The steps to create a MuDForM model are:

1. **Scoping**: the scope of the targeted model is specified by defining the purpose, the boundaries, and the input text that is selected from the knowledge source. The knowledge source is often an existing document, or a document that is created from interviews with (domain) experts. For each piece of selected text, a domain expert is appointed to provide missing information and assist with inconsistencies. The goal of scoping is to have relevant input for the modeling process, in order to (i) prevent unnecessary modeling work, (ii) detect other relevant input, and (iii) keep the model and the modeling process manageable.
2. **Grammatical analysis**: the selected text is analyzed and transformed into a set of phrases with terms that are candidate elements for the model. The goal of this step is to maximize the knowledge elicitation from the source, and to make the resulting model traceable to the input, which supports objective O7. This method step itself supports the realization of objective O6.
3. **Text-to-model transformation**: the specification spaces, which form the top-level structure of a model (see Section 5.4.2), are identified, and the phrases are transformed into pieces of model, which each are allocated to one of the identified specification spaces. This transformation is the transition from working with text to working with models, and supports the traceability aspect mentioned in objective O7.

²For readability, the start- and end-nodes are omitted from the activity diagrams that depict the method flow in Sections 5.4 and 5.5. The first step is the step without incoming control flow. The end step has no outgoing control flow.

Chapter 5. Domain-based Feature Modeling with MuDForM

4. **Model engineering:** the model is completed and inconsistencies are solved by following the method steps and guidelines, and iterating over the different views. Domain experts are involved to answer questions and decide about missing concepts and model conflicts. The goal is to acquire an unambiguous specification that meets the MuDForM objectives. *Model engineering* consists of a step to *Manage the dependencies between the specification spaces*, and three steps for *Engineering* the different types of specification spaces, i.e., *contexts*, *domains*, and *features* (see Figure 5.3). Section 5.5.2 explains the step *Engineer feature*. There is no restriction to the order of the engineering steps. But typically, the focus is on engineering a domain or feature, and while capturing in a context the needed concepts that are not defined in the domain or feature of interest. When the focus is on engineering a feature, existing domain models and feature models might be used to specify the behavior of the functions in the feature, which also validates those existing models. Often during feature engineering, new domain model elements and context model elements are identified. They have to be added to the domain or context models, to assure that all the concepts used in the feature model are properly defined. Otherwise, the feature model would be inconsistent and ambiguous.

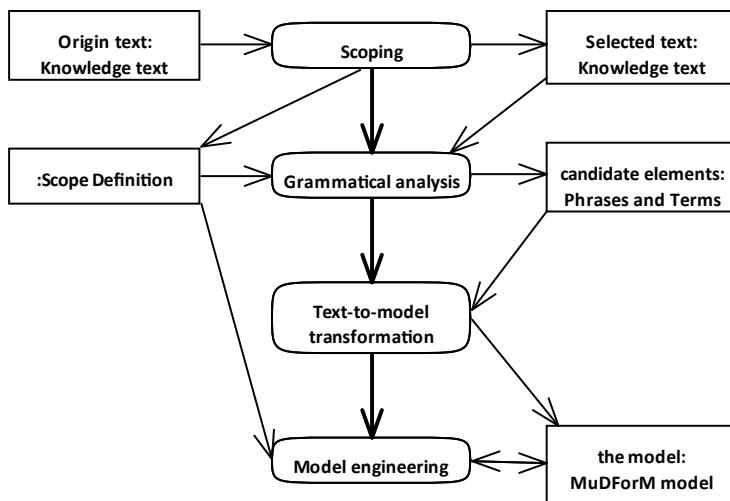


Figure 5.2: Main steps of MuDForM modeling process (UML activity diagram)

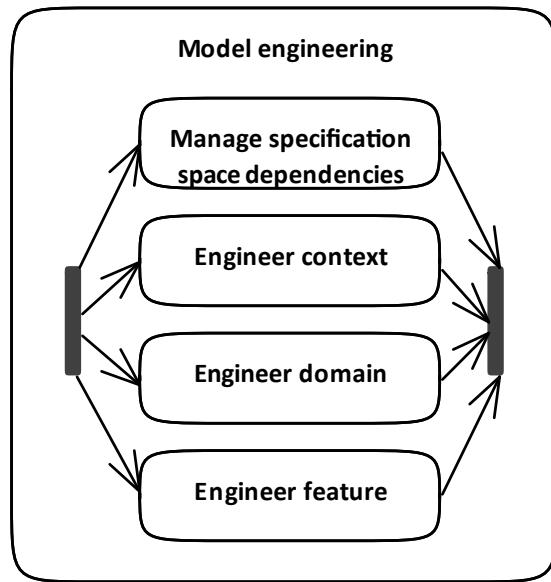


Figure 5.3: Main steps of method engineering (UML activity diagram)

Although the MuDForM outline is depicted as a *sequence* of steps, in practice they are carried out in *iterations*. For instance, often the grammatical analysis starts with a subset of the targeted input text, and then incrementally more text is analyzed. Moreover, when a modeling decision requires more information, it is possible to go back to a previous step to check if the required information was already present, yet overlooked.

When capturing the knowledge from a text in the model is the focus, the first three steps do cover feature, domain, and context simultaneously. *Feature engineering* becomes a separate step when the feature-specific viewpoints are made, which is in the *Model engineering* phase, and just before that, when the initial models are created at the end of the *Text-to-model transformation* step (see Figure 5.6).

5.4.2 MuDForM Model Structure

The top-level structure of a MuDForM model consists of a composition structure of Specification spaces, as depicted by the MuDForM metamodel fragment in Figure 5.4. A specification space contains specification elements, which can be a specification space of the same type, or one of the subclasses of specification element. Each type of specification space has its own specific types of elements, which will be listed in Section 5.4.3. MuDForM uses specification spaces (similar to UML packages) as containers for the specification elements that make up domains, features, or contexts. The separation of models in separate specification spaces supports objective O1. A specification space (child) may depend on other specification spaces (parent) with the following constraints:

- Features may depend on features, domains, and contexts.
- Domains may depend on domains, and contexts.
- Context are always independent; they form the relation of a MuDForM model with the world outside the model. As such, they enable the definition of self-contained domain models and feature models, which supports objective O3.

Although some MuDForM modeling concepts have the same name and similar semantics as a concept from the UML metamodel, they are not intended to be the same. MuDForM models are not UML models. But we have used the UML notation in the case study because of the available tooling and because in many cases it suffices regarding the semantics. Furthermore, for the focus of this chapter, the metamodel fragments in Figure 5.4 and Figure 5.5 are simplifications of the complete metamodel, which can be found online [33].

A domain model describes what can happen and what can exist in a domain. A feature model prescribes what shall happen and what shall exist. A context model captures assumptions and knowledge about elements that are needed to specify domains and features, but that exist outside those domains and features. A context model explicitly declares those elements and the properties that are needed to understand them. By defining the dependencies between specification spaces, specifications have no implicit semantics. So, when one wants to use a specification space, it is clear what other elements have to be incorporated, which supports the realization of objective O3. Domain models, feature models, context models, and the relations between them, make up a MuDForM model.

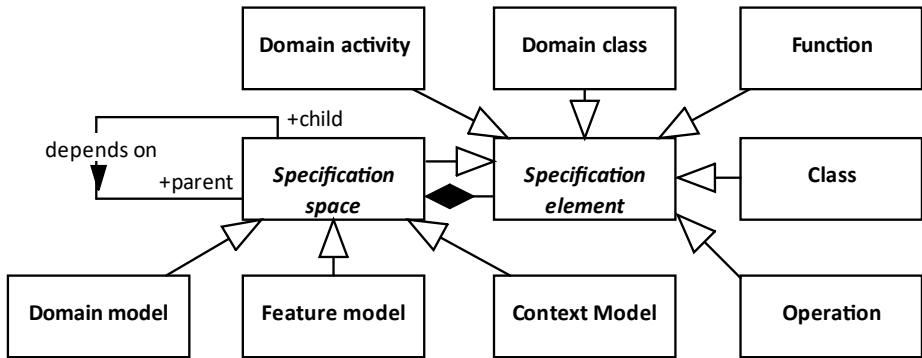


Figure 5.4: *Specification spaces and Specification elements* (UML class diagram)

For example, for a toll-road payment system, the **domain model describes**: vehicles can arrive and leave at a toll booth, someone can open and close the gate, and someone can pay a fee for a passing vehicle. The **feature model prescribes**: when a vehicle arrives at a toll booth, someone pays for that vehicle, someone opens the gate, and the vehicle leaves the toll booth. This assures that there will be no open payments, at the expense of blocking gates when someone cannot pay. The domain model is relatively stable because it describes all the possibilities. Feature models change over time because desired behavior changes over time and system environments evolve. For example, one could prescribe that someone pays after the vehicle leaves the toll booth, in favor of traffic flow, but with the risk of unpaid fees. In this case, the domain model remains the same, but the feature model is adapted. The **context model defines** external concepts like the payment service, fee, and license plate, to the extent needed to specify the elements of the feature and domain model.

5.4.3 MuDForM Specification Elements

MuDForM offers different types of *Specification elements*, as depicted in Figure 5.4. The type of *Specification space*, i.e., *Domain*, *Feature*, or *Context*, determines which types of *Specification elements* are allowed, and what is their semantics. All three types of *Specification spaces* have concepts to specify state, concepts to specify change, and concepts to specify the relation between state and change, which supports the realization of objective O5. Besides the concepts that are specific for a type of *Specification space*, almost all specification elements can have attributes and

specializations, and constraints attached to them. We list the different specification concepts below and clarify them with examples from a made up Banking domain.

Domain models contain the following types of concepts:

- *Domain activities* define what can happen in a domain. They are elements for the creation of composite behavioral specifications, *e.g.*, processes, scenarios, and system functions. Instances of *domain activities* are actions, which represent atomic (state) changes in the domain. Examples: to Withdraw money, to Transfer money, to Open.
- *Domain classes* define what objects can exist in the domain. They are elements for the creation of compositions and serves as the types of function attributes. Instances of domain classes are objects with a state. Examples: Client, Account, Cash register.
- *Interactions* define which objects can participate in which actions. Objects change state when participating in an action. All *domain classes* have an *object lifecycle* that expresses the order in which its objects may participate in specific actions. Examples: to Withdraw money from an Account at a Cash Register, to Transfer money from an Account to an Account, to Open an Account for a Client.

Feature models contain the following types of concepts (see Figure 5.5):

- *Functions* are *Specification elements* and *Behavior elements*. They specify what must happen when the *Function* is active, *i.e.*, when it is executed. A *Feature* is also a *Function*, *i.e.*, the top-level *Function* of a *Feature model*. Some *Functions* can be activated from outside the *Feature*, and some *Functions* are called by other *Functions*. Examples: Withdraw cash at an ATM, Transfer money on your phone, Open an account at the bank (office).
- A *Function* can use other *Behavior elements*, which can be other *Functions*, *Domain activities*, and *Operations*. Such usage is called a *Function sub-behavior*, and is typically only identified by the *Function* combined with the used *Behavior-element*, hence, does not have its own name. Some *Function sub-behaviors* are a *Function event*, *i.e.*, it is generated by the *Function* or the *Function* can react to it. Typically, one tree view is created with all the *sub-behaviors* of all *Functions* of a *Feature*. It has the *Feature* as the root and is called the *Feature structure*

(see Section 5.5.2.1). Examples: the *function* Transfer money on your phone uses the *domain activity* to Transfer money from an Account to an Account and uses the *function* Select an account in your banking app.

- *Functions* can have *attributes*, which have a (context or domain) *class* as a *type*. Typically, one view is created with all the *Function attributes* and all the *Function events* of a *Function*, which is called the *Function signature*, because it depicts how the *Function* is seen from the outside. Examples: the *Function* Withdraw cash at an ATM has an *attribute* withdraw amount with *type* Money amount, an *attribute* the ATM of *type* Cash register, and an *attribute* used card of *type* Bankcard.
- *Function lifecycles* describe what shall happen when a *Function* is instantiated. This so-called the control flow is specified in a process algebra style [60], *i.e.*, in terms of sequences, selections, concurrency, and iterations of *Function steps*. *Function steps* are *typed* by a *Function sub-behavior*, (and *sub-behaviors* are *typed* by a *Behavior element*). For each *Function step* is specified which *Function attributes* are *participating in* it. Examples: the *function* Withdraw cash at an ATM has *functions steps* verify card, enter withdrawal amount, validate amount for account, and then reject withdrawal, or dispense cash. These *steps* are *typed*, via a *Function sub-behavior*, by *Behavior elements*. For example, enter withdrawal amount is *typed* by the *operation* Enter money amount, and dispense cash is *typed* by the *domain activity* Dispense cash from a Cash register.

Context models contain the following types of concepts:

- *Classes* are types that define possible values (without a changing state). Examples: the *class* Money amount, which has an *attribute* currency and an *attribute* amount, *i.e.*, a positive number with two decimals, or the *class* IBAN, which represents the European wide definition of bank account numbers.
- *Operations* are types that define possible manipulations of values or comparisons between values. Examples: the *operation* Calculate interest on an account, or the *operation* Check if a string is a valid IBAN.

Context models typically contain two categories of concepts. First, physical quantities, like length, time, power, speed, and their operations. Second, concepts whose definition is not determined by the owners of the domains and features of interest,

Chapter 5. Domain-based Feature Modeling with MuDForM

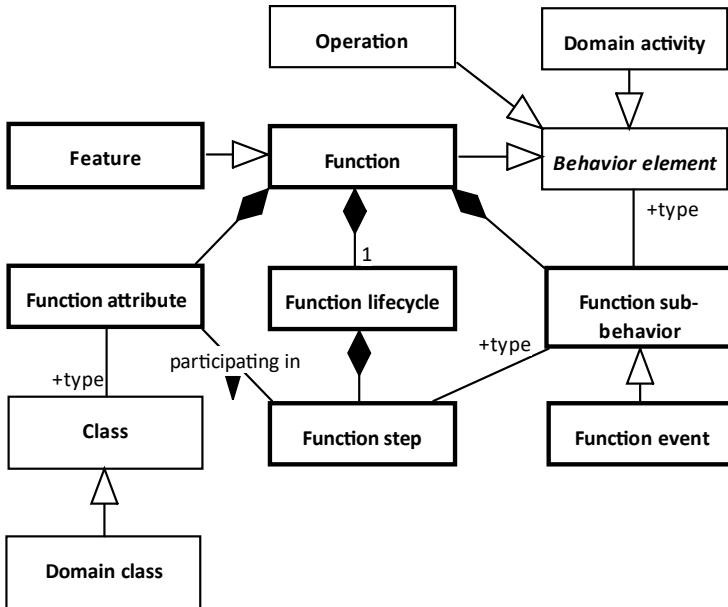


Figure 5.5: Feature modeling concepts (thick edge) related to other modeling concepts (thin edge) (UML class diagram)

like name, address, phone number, or an operation to determine the postal code of an address. These concepts might be needed to specify elements in domains and features, but their life (state changes) is not interesting. By explicitly defining needed concepts in a context model, the specifications of domains and features have no implicit semantics, which supports the realization of objective O3.

This section has presented an overview of the definition of MuDForM, the main structure of a MuDForM model, and what the main method ingredients are. It forms the context for the definition of the method part for feature modeling, which is explained in the next section. The modeling concepts, the method flow of this section and of Section 5.5, together with the guidelines of Appendix 5.A, support the realization of objective O7.

5.5 Feature Modeling

This section presents the parts of MuDForM that pertain to feature modeling. Following the outline from Section 5.4.1, Section 5.5.1 explains the method steps *Scoping*, *Grammatical analysis*, and *Text-to-model transformation*. Section 5.5.2 explains the method steps of *Engineer feature*, which is part of the *Model engineering* step depicted in Figure 5.3.

As explained in Objective O7 (see Section 1.1.3), guidelines are defined to support making analysis decisions and modeling decisions. Appendix 5.A presents the guidelines that are relevant for making feature models. Each guideline has a name, a description, and refers to the method steps in which it is applicable. The guidelines are ordered by the method steps.

Figure 5.6 details the method steps for each of the major steps presented in Figure 5.2. For the sake of clarity, we annotated the steps in the figure with notes containing references to the section in this chapter. The numbers after MD indicate the section in which the step is explained. (The numbers after CS indicate the corresponding sections of the case study.)

5.5.1 From Modeling Initiation to Initial Model

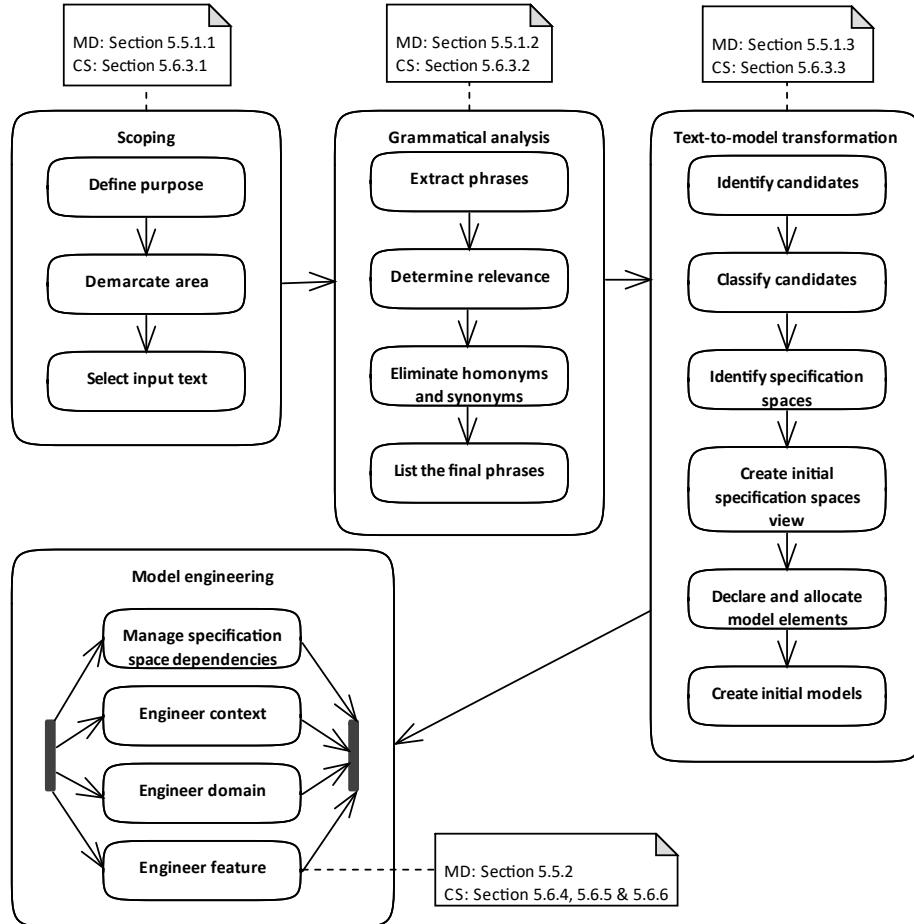
This section explains the method steps from the initiation of a modeling process, which starts with *Scoping*, followed by *Grammatical analysis*, and then *Text-to-model transformation*. These steps are generic for all models, and are not specific to feature modeling. That is why they are only explained to the extent that is relevant for feature modeling.

5.5.1.1 Scoping

The steps of *Scoping* are:

1. **Define purpose:** specify who the user/customer of the resulting specification is, and what they want to do with it.
2. **Demarcate area:** Name concepts that are in scope, and concepts that are out of scope.

Chapter 5. Domain-based Feature Modeling with MuDForM



3. **Select input text:** explicitly state which pieces of text (from a document) are the starting point for the specification process. A text can be the result of an interview with a (domain) expert. Typically, an expert is involved for each selected piece of text, to answer questions that arise during analysis and modeling.

5.5.1.2 Grammatical analysis

The steps of *Grammatical analysis* are:

1. **Extract phrases** from the selected input text and format them according to one of the phrase types: interaction structure phrase, static structure phrase, state structure phrase, and conditional phrase. (The explanation of these terms is out of scope for this chapter; Chapter 4 explains them. Table 5.1 in Section 5.6.3.2 shows some examples.) Typically, each sentence from the input text leads to one or more extracted phrases. The extracted phrases form a decomposition of the original sentence, and are processed in the next method steps, in which they can change in terminology or structure due to modeling decisions. Based on those decision, it is possible to rewrite the original sentence at any time during the modeling process, in order to check if the model still expresses the initially intended meaning. When no explicit input text is used, then sentences can be elicited directly from domain experts.
2. **Determine the relevance** of each extracted phrase from the perspective of the defined scope. Discard phrases that do not fit the scope definition, or that are duplicates.
3. Check all phrases for **homonyms and synonyms**, and **eliminate** them in consultation with the domain experts to assure that all terms (words) have exactly one meaning, and that all relevant meanings are covered by exactly one term.
4. This results in a **list of final phrases** which is used as input for the model. Make the list of phrases for the initial model via these criteria: all extracted phrases that are marked as relevant and not discarded, all newly added and rewritten phrases, and replacement of the possible homonyms and synonyms with the chosen term.

The KISS method [96], which is the starting point for the grammatical analysis in MuDForM, provides a more detailed description of this phase in the modeling process.

5.5.1.3 Text-to-Model Transformation

The *Text-to-model transformation* consists of the following steps:

1. **Identify candidates:** Determine the terms in the phrases, *i.e.*, nouns, verbs, adjectives, and adverbs, that are a potential *Specification element* for the *Model engineering* step.
2. **Classify candidates:** Select which type of element each identified term is. The possible types are: *domain class*, *domain activity*, *context class*, *function*, *attribute*, *domain*, *feature*, *context*, *operation*, *condition*, *function event*, *function step*, *activity operation*, *class relation*, or *specialization*.
3. **Identify specification spaces:** Identify *contexts*, *domains*, and *features*. Choose *specification spaces* for *specification elements* that are coherent. Each *specification space* should have an owner who is responsible for its content.
4. **Create initial specification spaces view:** Create a view, *e.g.*, a diagram, with all the *specification spaces*. Create relations (*dependencies* or *compositions*) between *spaces* if they are expected, or already known, complying with the rules specified in Section 5.4.2.
5. **Declare and allocate elements:** Place each *specification element* in the most logical *specification space*. An *element* can be reallocated during *model engineering*.
6. **Create initial models:** Create a first version of the models in the *specification spaces* from the list of final phrases. For a *feature model*, this means creating the initial *feature structure* and initial *function signatures*. (The other *specification spaces* are not the topic of this chapter.) In the case that there is no input text, and hence no grammatical analysis result, this is the starting point of the modeling process.

5.5.2 Engineer Feature

During the step *Model engineering*, the initial models are iteratively transformed into engineered *domain models*, *context models*, and *feature models*, as depicted in Figure 5.3. The two main modeling principles are 1) keeping the views consistent, and 2) acquire information from experts or documents to achieve a complete specification. This section zooms in on the step *Engineer feature*. Each feature engineering step

corresponds with a specific viewpoint. The rest of this section explains those steps. A feature is engineered by working in parallel on the *feature structure*, *function lifecycles*, and *function signatures*, as depicted in Figure 5.7.

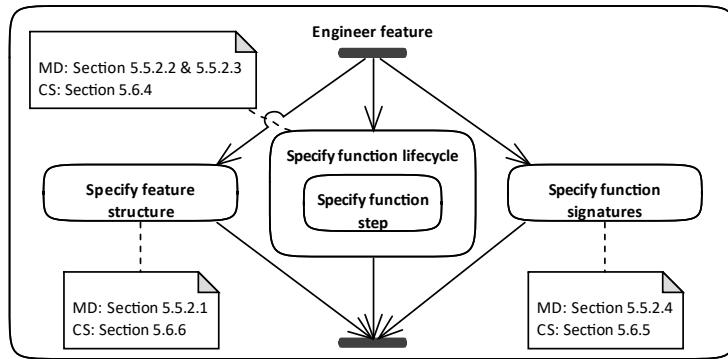


Figure 5.7: The steps of *Engineer feature* (UML activity diagram, MD: method definition, CS: case study)

5.5.2.1 Specify Feature Structure

During this step, the behavioral composition of the feature is managed. This means specifying the decomposition of the feature into functions, and possibly of each function into sub-functions, via the abovementioned concept *Function sub-behavior*. Additionally, the use of *behavioral elements* (*operations*, *activities*, *functions*) from outside the *feature* is specified. The *feature* is the root of the resulting tree structure.

5.5.2.2 Specify Function Lifecycle

During this step, the control flow, *i.e.*, *function lifecycle*, of each *function* is specified. This means describing the order in which the *function steps* must be executed. The *function steps* refer to *sub-behaviors* of the *function* (see Figure 5.5). All the *sub-behaviors* of the *function* must occur at least once as a *function step*. MuDForM distinguishes different types of ordering: sequences, selections, concurrency, or iterations. There are typically two ways to reason about a *function lifecycle*. The first way is to start with the main input *attributes* of the *function* and decide which actions should be performed on them going forwards in time. The second way is to start with

the end of the *function* in mind, which is a *postcondition* or a *domain activity* that must be performed, and then reason backwards about the order of the *function steps*.

5.5.2.3 Specify Function Step

During this modeling step, each *function step* is related to the context with respect to the objects (parameters) that play a role in the step. The following aspects must be specified:

- *Function attributes* are allocated to (the actual parameters of) the *step*. *Function attributes* must belong to the same *function* as the *step*, or they belong to a *function* that contains that *function*. Typically, a *feature*, which itself is also a *function*, has *attributes* that will be used in many *steps* of the *functions* within the *feature*.
- The *preconditions* for this *step*, *i.e.*, the constraints on the step participants. A condition is typically expressed in a logical language and may only use terms that are elements within the scope of the *function*. (Step preconditions are also called enter conditions.)
- The *postconditions* for this *step*, *i.e.*, the conditions that have to be true for this step to end. (Step postconditions are also called exit conditions.)

A specified *function lifecycle* must be consistent with the *domains* that the *function* uses, which means that for *function steps* that are an instance of a *domain activity*, holds that:

- The objects (identified via *function attributes*) allocated to the action (identified by the function step) have a type that corresponds with a *domain class* that is *involved* in the *domain activity*.
- *Function attributes* are allocated to each input *attribute* of the *function step*, and their types match.
- The *function lifecycle* does not violate the *object lifecycle* of an involved object (which can only be fully controlled at execution time, and not during modeling).

5.5.2.4 Specify Function Signatures

A *function signature* describes the interface of each *function* in terms of *attributes* and *events*, and frames the behavior of the *function*. *Attributes* are typed by a (*domain*) *class*, and *events* are typed by a *behavioral element*, as depicted in Figure 5.5.

All *function signatures* can be put into a single diagram (see for example Figure 5.18), or a separate diagram is created for important and complex *functions*. Each *attribute* can be an input, output, or local *attribute*. Local *attributes*, which are used to pass on data between *function steps*, could be omitted from the *function signature*, because they are not visible outside the *function*. But then, a different view would have to be created for the declaration of the local *attributes*. So normally, we put them in the same view.

The *function signatures* also specify the *preconditions*, and *invariants* that hold for the *function attributes*, *i.e.*, things that must be true in order to guarantee the proper outcome of the *function*. It is possible to specify *postconditions*, but this is not necessary, because MuDForM follows a whitebox perspective on *function specifications*. Although, a *postcondition* could help guide the design of the *function lifecycle*.

5.6 A Case Study: Modeling the Processes of ISO26262

This section presents the results from a case study in which we applied MuDForM to model the ISO26262 standard for functional safety together with automotive engineers from research and innovation institute TNO³. The ISO26262 standard was chosen as our case for the following reasons:

- It is mature and comes with an explicit glossary of definitions, which are a good starting point for analysis.
- It is clearly structured according to a set of processes, which are described in several documents (called parts).
- It is large and covers many aspects, which makes it relevant as a case for validating MuDForM.
- TNO Automotive has a need for an unambiguous, comprehensible specification of the ISO26262, which will be explained in Section 5.6.1.

³<https://www.tno.nl/en/>

- Functional safety is an example of a quality domain, which is specifically the target of MuDForM, as stated in Objective O2 in Section 1.1.3.

The goal of the case study is to evaluate the MuDForM support for specifying features in terms of domain models. This means using the *domain activities* and *domain classes* of the *domain model* (published by Khabbaz Saberi [93]), as the *types of function steps* and *function attributes*, respectively. The focus of the case study is not to validate how suited the resulting model is for a specific application.

Section 5.6.1 introduces the case. Section 5.6.2 gives an overview of the part of the case study that is the focus in this chapter, and explains its execution. Sections 5.6.3 through 5.6.6 present the resulting model and elaborate on the modeling decisions by explaining how the guidelines from Appendix 5.A are applied. The presented diagrams use a notation that is compliant with the UML metamodel [120], and are made with the tool Enterprise Architect [152]. We have added explanations for non-standard uses of the UML notation in the rest of this section.

5.6.1 Introduction to the Case

TNO Automotive is a research and innovation organization that develops new concepts and new approaches for the development of automotive systems for industry partners. The ISO26262 standard for functional safety in automotive systems prescribes the processes that must be executed, and the work products that must be produced to prove the absence of unreasonable risk in safety-critical systems in the automotive domain. The specification of ISO26262 consists of 10 separate documents, called parts, which add up to 470 pages. Each part consists of several clauses. The case study did not involve the entire standard, as it would require involvement of many experts over a wide range of the automotive supply chain. Section 5.6.3.1 specifies which part of the standard's text is selected for the case study.

In the automotive sector, there is an increase in the complexity of the systems, in the communication between these systems, and in the amount of safety-critical functionalities. Accordingly, the work needed to achieve functional safety and its certification, is becoming increasingly time-consuming and prone to human error. TNO and some of their customers, based on their own experience, expressed the need for a more predictable and uniform process. They find that a more objective certification process will help to reduce the risk of human errors during the system development process, decrease the certification costs, and increase the safety of

5.6 A Case Study: Modeling the Processes of ISO26262

automotive systems [93]. Moreover, they find that the use of a controlled language can help to achieve this [106].

In the development of automotive systems, people from different engineering disciplines and several companies cooperate very intensively. As explained by Khabbaz Saberi [93], these people often have different interpretations of the standard's text, because the terminology and phrasing are not always consistent or completely unambiguous, and there are assumptions in the standard about the meaning of terms that are not an intrinsic part of the standard, *e.g.*, the terms pertaining system design, such as system, element, and function. A MuDForM model that covers the domain concepts and the work processes of the standard, could be the basis for a shared understanding.

TNO has a research program in integrated vehicle safety [9], and offers services to automotive suppliers for acquiring the required ISO26262 certification for their products. Therefore, TNO embraces that the people involved in functional safety analysis and system design gain a **consistent and unambiguous understanding of the activities and artifacts** prescribed by the ISO26262 standard. Parts of the domain model and the process specifications resulting from this case study are explained in depth by Khabbaz Saberi [93]. That manuscript explains the reasons why TNO needs a domain model of the ISO26262 standard.

5.6.2 Case Study Overview and Execution

The case study was executed as a collaboration between a MuDForM researcher, several TNO Automotive engineers, and an ISO26262 committee member supporting the unraveling of unclarities in the standard's text. The case study was performed between January 2018 and January 2020. The case study is following the MuDForM method flow depicted in Figure 5.6, and the *feature engineering* flow depicted in Figure 5.7. During the modeling process, the most important decisions were recorded, and some of them are used in the explanation of the modeling results throughout this section. The complete model is not publicly available due to intellectual property rights. This chapter shows examples of the resulting model to illustrate how the method is applied.

A challenge for this case study is how to model the explicitly stated requirements from the text in the standard. Section 5.6.3.2 and 5.6.4 showcase how such a requirement is treated by MuDForM, and how it is captured in the model.

Section 5.6.3 presents the phase from modeling initiation to the initial model. It is limited to *feature modeling* as much as possible. We give some examples of how the grammatical analysis results look like, and refer to the guidelines for some of the made analysis decisions.

After that, we present the step *Engineer feature* by following the steps of Figure 5.7. Section 5.6.4 presents the modeling of *function lifecycles* for a few of the ISO26262 processes, including the details of *Specifying function steps* inside a *function*. This section concludes with the final *function signatures* and the final *feature structure* in Sections 5.6.5 and 5.6.6, respectively.

5.6.3 From Modeling Initiation to Initial Model

This section presents the steps *Scoping*, *Grammatical analysis*, and *Transformation from text to model*, applied to the case.

5.6.3.1 Scoping

The three steps of *Scoping* lead to the following results:

1. The **defined purpose** of the model (including the part that is described by Khabbaz Saberi [93]) is to have an unambiguous specification of the artifacts and processes that the ISO26262 standard prescribes. Following the guideline *Different specification spaces have a different purpose*, we distinguish two specific purposes for the specification of the processes: provide safety engineers with work instructions, and provide the requirements for a tool that supports the ISO26262 processes. These two purposes are both derived from the guideline *Common feature model purposes*.
2. The **demarcation of the area** concerns the clauses for item definition and hazard analysis. Examples of concepts that are in scope are: item, hazard, hazardous event, and malfunction. System design concept, such as system, element, and function, are out of scope, but might be needed as reference, and thus captured in a *context model*.
3. For this version of the model, we **select as input text** Part 3 of the ISO26262 (named the Conceptual phase) excluding the Functional Safety Concept clause. The text of Parts 1 (Vocabulary), 8 (Supporting processes), and 10 (Guidelines to the ISO 26262) to which Part 3 refers, will also be considered.

5.6 A Case Study: Modeling the Processes of ISO26262

The demarcation and *selected input text* have been adjusted during the process. Initially, we wanted to cover a larger part of the standard. But that appeared to be too much to start the modeling process with, and the available time of the domain experts was restricted. So, we had to narrow the scope to Part 3. We applied the guideline *Start with the foundation and the core* to come to the selection of the Item definition clause of Part 3, because that is the foundation. Then we selected the clauses Hazard Analysis and Risk Assessment (HARA) and Functional Safety Concept. After applying the guideline *Start small*, and because the output of the HARA clause is needed for the clause Functional Safety Concept, we came to the specified selection of the input text.

5.6.3.2 Grammatical Analysis

In most cases, and in this case as well, the *Grammatical analysis* phase delivers content for all three types of *specification spaces*, i.e., for *domain models*, *context models*, as well as *feature models*. As an example, we take the following sentence from clause 5.2 in part 3 of the ISO26262 standard:

This definition serves to provide sufficient information about the item to the persons who conduct the subsequent sub-phases: “Hazard analysis and risk assessment” and “Functional safety concept”.

In addition, this sentence from requirement 6.4.4.2 from the standard is analyzed:

If similar safety goals are combined into a single one, in accordance with 6.4.4.1, the highest ASIL⁴ shall be assigned to the combined safety goal.

Table 5.1 shows phrases that we extracted from the *input text* and that are relevant for *feature models*. The first column contains the *input sentence*. The second is the *phrase type*, as mentioned in the *Extract phrases* step of Section 5.5.1.2, followed by the *extracted phrase*. The last column explains the made analysis decisions. The table is the result of all four steps of *Grammatical analysis*, i.e., *Extract phrases*, *Determine relevance*, *Eliminate homonyms and synonyms*, and *List final phrases*.

⁴Automotive Safety Integrity Level

Table 5.1: Selection of the grammatical analysis

<i>Input sentence</i>	<i>Phrase type and Extracted phrase</i>	Decisions
This definition serves . to provide sufficient information about the item to the persons who conduct the subsequent sub-phases: “Hazard analysis and risk assessment” and “Functional safety concept”	<i>State structure phrase:</i> Item definition is a phase.	“This definition” refers to “Item definition” (confirmed by the domain expert). A phase is considered to be a part of the total safety lifecycle. It is not a safety concept itself, but a concept to organize the process for functional safety.
	<i>State structure phrase:</i> Hazard analysis and risk assessment (HARA) is a phase.	The same reasoning as for Item Definition.
	<i>State structure phrase:</i> Functional safety concept is a phase.	Functional safety concept is out of scope as stated in section 4.2. “to conduct” is out of scope, because it is about how the document is structured.
	<i>Interaction structure phrase:</i> Person conducts phase.	Apparently, different phases can be handled by different people. The verb “to conduct” is about the process definition domain, which is considered irrelevant.
	<i>Interaction structure phrase:</i> HARA follows Item definition.	This phrase says something about the order of behavior. The verb “to follow” is not considered as a relevant specification element itself for the same reason as “to conduct” is not.

5.6 A Case Study: Modeling the Processes of ISO26262

<i>Input sentence</i>	<i>Phrase type and Extracted phrase</i>	<i>Decisions</i>
If similar safety goals are combined into a single one, in accordance with 6.4.4.1, the highest ASIL shall be assigned to the combined safety goal.	<i>Conditional phrase:</i> If safety goals are combined into a single safety goal, then the ASIL of the single safety goal shall be equal to the highest ASIL of the combined safety goals.	<p>There are several decisions involved:</p> <ul style="list-style-type: none"> • To remove possible ambiguity, the word ‘then’ is added after the first comma in accordance with guideline <i>Standardize logical constructs</i>. • The part “in accordance with 6.4.4.1” is removed according to guideline <i>Ignore phrases about the document itself</i>. • The word similar is considered irrelevant for the meaning. The domain expert stated that determining similarity between goals is the responsibility of the safety analyst. The requirement is not influenced by goals being more or less similar. • More phrases were extracted from the input sentence, such as “To combine goals into goal”. These are not analyzed and discussed here, because they were phrases for the domain model.

5.6.3.3 Text-to-Model Transformation

This section discusses the creation of the initial models from the results of the *grammatical analysis*. Table 5.2 presents a subset of the *candidate elements* and their classification. It is the result of the steps *Identify candidates* and *Classify candidates* described in Section 5.5.1.3.

Chapter 5. Domain-based Feature Modeling with MuDForM

Table 5.2: Classification of candidates

Candidate	Classification
Safety Lifecycle	<i>Feature</i> , because that is the name used in the header and the guideline <i>Identify features and functions from text headers</i> is applied.
Item Definition	<i>Function</i> , because we applied the guideline <i>Define a function for coherent behavior that will be assigned to one actor</i> , and because Item Definition is a phase that can be conducted by a person as stated in the input text. Furthermore, the guideline <i>Identify features and functions from text headers</i> is applicable.
HARA	<i>Function</i> , for the same reason as Item Definition.
the Item	<i>Function attribute</i> , because of the guideline <i>Definite articles indicate a function attribute</i> and throughout the selected input text, “the Item” is used repeatedly.
If safety goals are combined into a single safety goal, then the single safety goal’s ASIL shall be equal to the highest ASIL of the combined safety goals	The whole phrase is a candidate <i>condition</i> , because it is a conditional phrase due to the if-then construct. Following the guideline <i>Auxiliary verbs indicate the specification space type</i> and the use of “shall”, we allocate it to the feature model.

To understand the relation of the Safety Lifecycle *feature* to its context, Figure 5.8 presents the *specification spaces* (represented as UML packages) and their relations. The Safety Lifecycle *feature* depends on the Design Specification *context*, which defines the concepts that are the input of a safety analysis, and depends on the Functional Safety *domain*, which defines all the concepts from the standard that are needed to define the *feature*. The Functional Safety *domain* has two subdomains: the Item Definition *domain*, and the Hazard Analysis and Risk Assessment *domain*. This view is the result of the steps *Identify specification spaces* and *Create initial specification spaces view*. In the step *Declare and allocate elements*, the specification spaces are populated with the identified *candidates* and with the phrases that involve those *candidates*. There are two views related to *feature modeling*, which are created in the step *Create initial models*: 1) the initial *feature structure*, and 2) the initial *function signatures* of the *feature Safety Lifecycle*.

5.6 A Case Study: Modeling the Processes of ISO26262

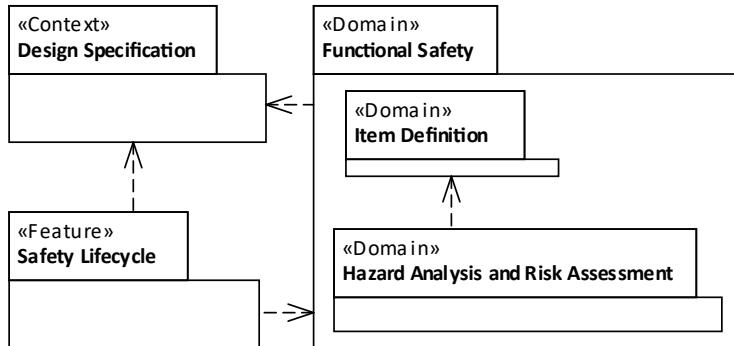


Figure 5.8: *Specification spaces* of the ISO26262 (UML package diagram)

From the output of the *Grammatical analysis* step, the initial *feature structure* of the *feature Safety Lifecycle* are created (see Figure 5.9). The *Feature Safety Lifecycle* has two sub-*functions*: 1) Item Definition, and 2) HARA, which correspond to clauses in the ISO26262 text.

As mentioned in the example sentences of Section 5.6.3.2, HARA consists of sub-activities, which can be performed by a different person. That is why we applied the guideline *Define a function for coherent behavior that will be assigned to one actor*. So, HARA has four sub-*functions*: 1) Hazard Analysis, 2) Hazardous Event Identification, 3) Risk Assessment, and 4) Safety Goal Determination. Those sub-*functions* correspond to sections in the HARA clause of the ISO26262 text. The relations have been described with a UML composition relation, because the instances of the composed activities are fully executed within the instances of the composing activities, e.g., Risk Assessment is executed within HARA, and Item Definition is executed within Safety Lifecycle.

Figure 5.10 presents the *initial function signatures*. For the *feature Safety Lifecycle*, which is a *function* itself, we have applied the guideline *Define attributes for sets of strong objects*, which has led to the set *attributes* all systems, all design objects, and all items. For HARA, we have applied the guideline *Define function wide attributes for central objects*, resulting in the *function wide attribute* the Item for HARA and for Item Definition. The four sub-*functions* of HARA are in the diagram too, because they are identified in the *feature structure*. However, there is no information to identify their *attributes* yet. The UML aggregation relation is used to declare the *function attributes*, because they can be seen as a reference from the *function* to a (*domain*)

Chapter 5. Domain-based Feature Modeling with MuDForM

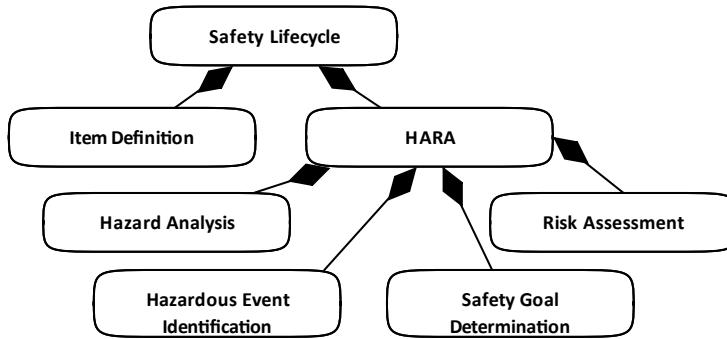


Figure 5.9: Initial *feature structure* of Safety Lifecycle (in UML notation)

class. (We did not use a composition relation because the usage of class instance is not necessarily restricted to this function.). The role name on the side of the *class* indicates the name of the *attribute*, e.g., Item Definition.all design objects has the type Design Object (from the *context model* Design Specification).

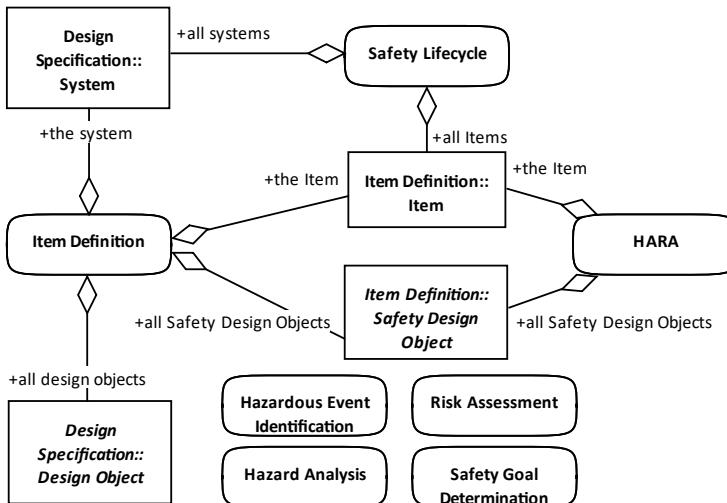


Figure 5.10: Initial *function signatures* (in UML notation)

5.6.4 Specify Function Lifecycles

This section describes the creation of a subset of the *function lifecycles* to demonstrate the steps and guidelines explained in Section 5.5.2.2.

Safety Lifecycle - The *function lifecycles* of the *function Safety Lifecycle* contains two *steps*: 1) Item Definition and 2) HARA. These *steps* can be executed iteratively in any order (see Figure 5.11) for as long the Safety Lifecycle *function* is active. We have identified two *function attributes*: all Items and all Systems (as already modeled in Figure 5.10). These are the result of the guideline *Introduce feature attributes for sets of existing objects*, and the fact that Item Definition starts with a set of Systems that can be chosen for Safety Analysis, and HARA starts with a set of Items to choose from. UML control flows are used to model the order of the steps, and are represented with thick arrows. UML object flows are used to specify that a *function attribute* *participates* in a *function step*, and are represented with a thin arrow. The direction of the arrow indicates if the *attribute* is *input* or *output* for the *step*. The stereotype «set» indicates that the *function attribute* does not contain one object of the *type System*, but can contain multiple objects of the *type System*.

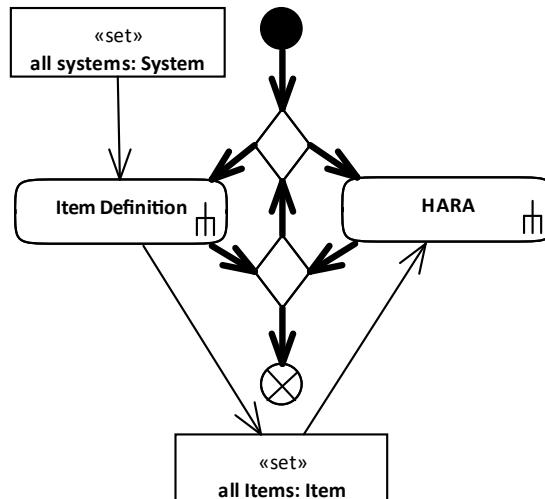


Figure 5.11: *Function lifecycle* of the *function Safety* lifecycle (UML activity diagram)

Item definition - Figure 5.12 shows the *function lifecycle* of Item Definition. The diagram is defined by following the guideline *Go with the flow*. The guideline *Check the domain models for unused activities* is used to check whether all the referred *domain activities* from the Item Definition *domain* are defined. We will not mention these two guidelines anymore, because they are used in the creation of every *function lifecycle*. After specifying the control flow of the *function lifecycle*, we specify the details of each *function step*.

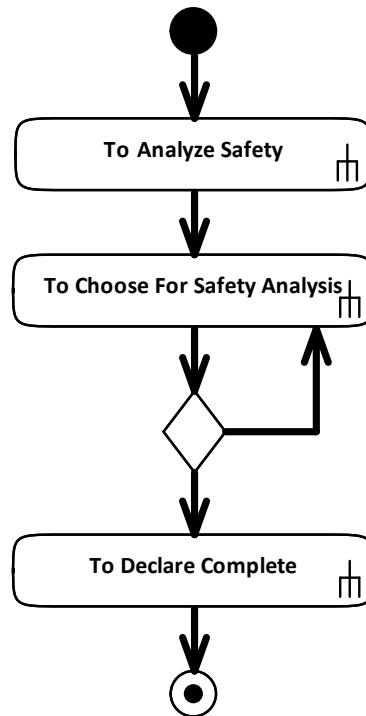


Figure 5.12: *Function lifecycle* of the *function* Item Definition (UML activity diagram)

Figure 5.13 shows which *function attributes* are participating in each *step*, as described in Section 5.5.2.3. The *feature attribute* all systems and the *function attribute* all design objects are *participating* in the steps To Analyze Safety, and To Choose for Safety Analysis, respectively. All three *function steps* (represented as UML actions) in the *function lifecycle* are invocations of *domain activities* (defined in the *domain*

5.6 A Case Study: Modeling the Processes of ISO26262

Item Definition). We gave the actions the same name as the corresponding *domain activity*. The *attribute* the Item is connected to all three *steps*. The *attribute* safety design objects of the Item is connected to the step To Choose for Safety Analysis. The statement between the ‘[...]’ indicates the *precondition* that must hold for the instances in the *attribute* all design objects. The flow ends with the *step* To Declare Complete when the analyst determines that all needed Design objects are *chosen for safety analysis*. After this, the Item is available for HARA, and the following (not modeled) steps of the Safety Lifecycle.

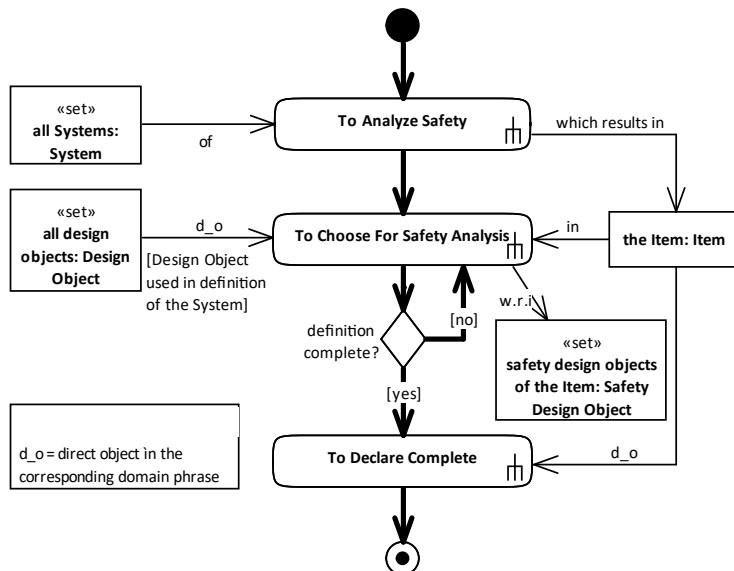


Figure 5.13: Item Definition with specified *function steps* (UML activity diagram)

HARA - Figure 5.14 presents how the sub-*functions* of HARA are ordered. The *function lifecycle* contains four *steps*, one for each sub-*function* of HARA. They can be executed iteratively in any order.

The next method step is to apply *Specify function step* to HARA. Figure 5.15 shows which *function attributes* of HARA are *participating* in each *function step*. To avoid too many crossing lines, which would hinder the readability of the diagram, the control flow arrows, as depicted in Figure 5.14, are omitted. HARA is centered around



Figure 5.14: *Function lifecycle* of HARA (UML activity diagram)

the *function attribute* the Item. Because all sub-*functions* are contained in the definition of HARA, as modeled in Figure 5.9, they can access the Item and it does not have to be passed onto the sub-*functions* via parameters. All the *attributes* in HARA have a type that is a *domain class* from the *domain model*, e.g., :Item intended Function has the type Item Intended Function. Each of those *domain classes* is a subclass of the *domain class* Safety Design Object. All the *attributes*, i.e., all the involved Safety Design Objects must be part of the Item, which is depicted in Figure 5.18. That a Safety Design Object is part of an Item, is modeled in the *domain model*, of which the corresponding fragment is presented in Figure 5.19.



Figure 5.15: HARA with *function attributes* connected to *function steps* (UML notation)

5.6 A Case Study: Modeling the Processes of ISO26262

For HARA, the *lifecycle* view has been split into two separate views, because otherwise there would too many control flow arrows and object flow arrows crossing each other, which would make the view messy. There is 1) a view with just the order of the *steps* and without any *function attributes* related to them (Figure 5.14), and 2) a view, which specifies which *function attributes* are *participating* in which *function step* (Figure 5.15). Another option is to use a textual notation for the *function steps*, as in a regular programming language, where a function call contains the link between variables and the actual parameters of the function call.

Bookkeeping - During the modeling of Hazard Analysis, Hazardous Event Identification, and Safety Goal Determination, we detected that they all required the use of *domain activities* To Reject and To Combine from the Functional Safety *domain*. These are *activities* to manage Safety Analysis Objects. Following the guideline *Define a function for recurring behavior in multiple functions*, we identified the *function Bookkeeping*, which is specified in Figure 5.16.

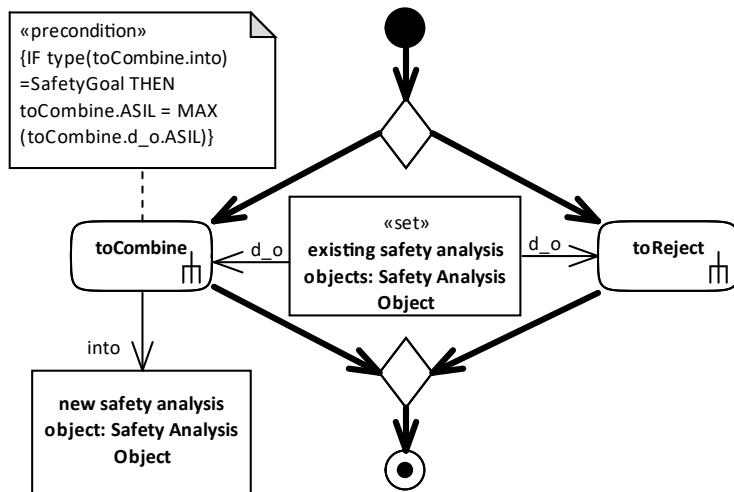


Figure 5.16: *Function lifecycle* of Bookkeeping (UML activity diagram)

Phrases that were classified as a *condition* should be captured in the model as *pre-condition*, *postcondition*, or *invariant* of a *function*, *domain activity*, or *function step*. The *function lifecycle* Bookkeeping contains an example of such a *condition*. To understand the *condition*, the elements from the *domain model* that are used in the

Chapter 5. Domain-based Feature Modeling with MuDFoRM

specification of the *condition*, must be clear. To understand the formal specification of the *condition*, we have shown the relevant part of the *domain model* in Figure 5.17. The association relations between the *domain activity* To Combine and the *domain class* Safety Analysis Object mean that instances of that *domain class* participate in an instance of that *domain activity*. The arrow at the end of the into-association indicates that the *domain activity*'s execution instantiates an object of the *domain class* Safety Analysis Object. So, To Combine requires two or more Safety Analysis Objects along the d_o association and gives a new Safety Analysis Object along the into-association. The aggregation relation between To Combine and ASIL Ranking indicates that To Combine has an *attribute* with the name ASIL and the *type* ASIL Ranking.

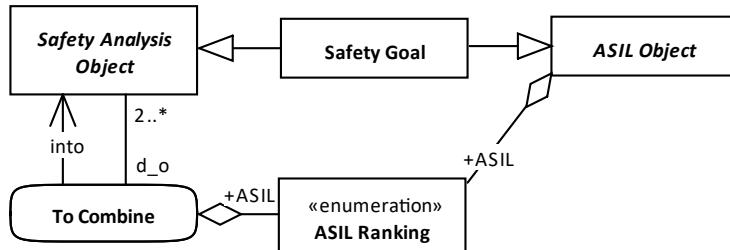


Figure 5.17: Fragment of *domain model* around to Combine (UML notation)

The candidate *condition* is: *If safety goals are combined into a single safety goal, then the single safety goal's ASIL shall be equal to the highest ASIL of the combined safety goals.* The referred model elements in this phrase are:

- *Domain class*: Safety Goal.
- *Domain activity*: To Combine.
- *Activity role*: To Combine into.
- *Activity role*: To Combine d_o (direct object) derived from “safety goals are combined” in the phrase.
- *Context class*: ASIL.
- *Attribute*: Safety Goal.ASIL with type ASIL.

5.6 A Case Study: Modeling the Processes of ISO26262

- *Operation:* highest (defined on ASIL values). There must be a ranking of ASIL values to be able to speak of the highest ASIL. This is covered in the definition of the class ASIL.
- Activity *attribute:* To Combine.ASIL with type ASIL.

Finally, the *condition* has to be added to the model. This is done in two steps: 1) determine the moment in the *lifecycle* that the *condition* must hold, and 2) formalize the predicate. In this case, the position is the To Combine step in Bookkeeping (see Figure 5.16). The predicate is:

```
If type(BookKeeping.toCombine.into.SafetyDesignObject) = SafetyGoal  
    then Bookkeeping.toCombine.ASIL =  
        MAX(Bookkeeping.toCombine.d_o.safetyGoal s: s.ASIL)
```

The IF clause about “type(...) = SafetyGoal” is added, because the *domain activity* To Combine is defined on the abstract *domain class* Safety Design Objects (see Figure 5.17) and not just on the *domain class* Safety Goal. The requirement about taking over the highest ASIL ranking is only valid for Safety Goals and not for other types of Safety Design Objects.

5.6.5 Specify Function Signatures

Figure 5.18 presents the final *signatures* of a subset of the *functions*. The major changes to the initial *function signatures* of Figure 5.10 are the addition of some, *i.e.*, a *precondition*, and two *invariants*. The *invariant* connected to Item Definition states that only the Design Objects that are used in the definition of the system, may be used in the definition of the Item. Similarly, in HARA, only the Safety Design Objects that are part of the Item, may be used in the sub-*functions* of HARA. The semantics for these *invariants* come from the Item Definition *domain model*, which is depicted in Figure 5.19. Another constraint is the *precondition* that only Items that have been declared complete in the Item Definition may be used in HARA.

5.6.6 Specify Feature Structure

At the end of the *feature engineering* phase, we revisit how the functions in the feature model relate to each other and to the context (see Figure 5.20). Most of the *functions* invoke *domain activities* of the domains that the Safety Lifecycle *feature* is

Chapter 5. Domain-based Feature Modeling with MuDFoM

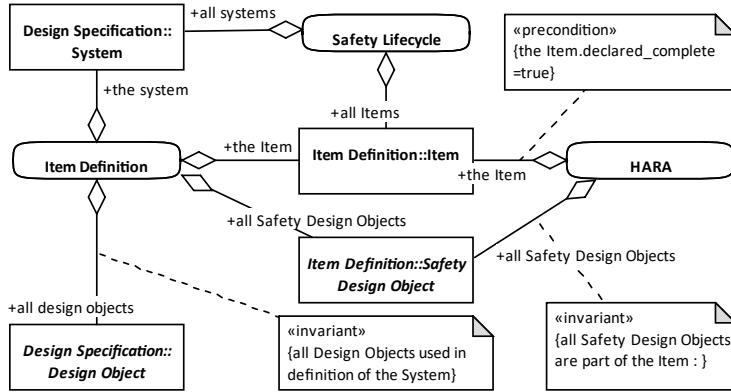


Figure 5.18: Final *function signatures* (UML notation)

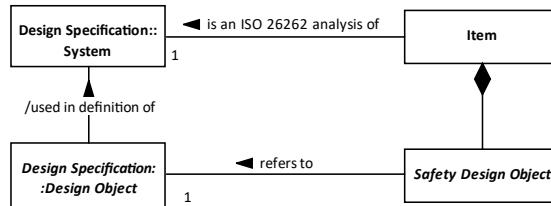


Figure 5.19: Fragment of the Item Definition domain(UML class diagram)

dependent on, e.g., Bookkeeping invokes To Combine and Item Definition invokes To Analyze Safety, which is expressed via an aggregation association. This means that the *function lifecycle* of those *functions* may contain *steps* that are instances of such a *domain activity*. The *function Safety Lifecycle*, which is the root of the structure, as well as the *function HARA*, only use other *functions* from the *feature model*.

5.7 Discussion

In this section, we reflect on the research questions and discuss our findings emerging from the case study. Section 5.7.1 discusses the support that MuDFoM provides for making domain-based feature specifications. Section 5.7.2 reflects on how MuDFoM helps bridge the gap between feature models (as meant in FODA [90]) and domain

5.7 Discussion

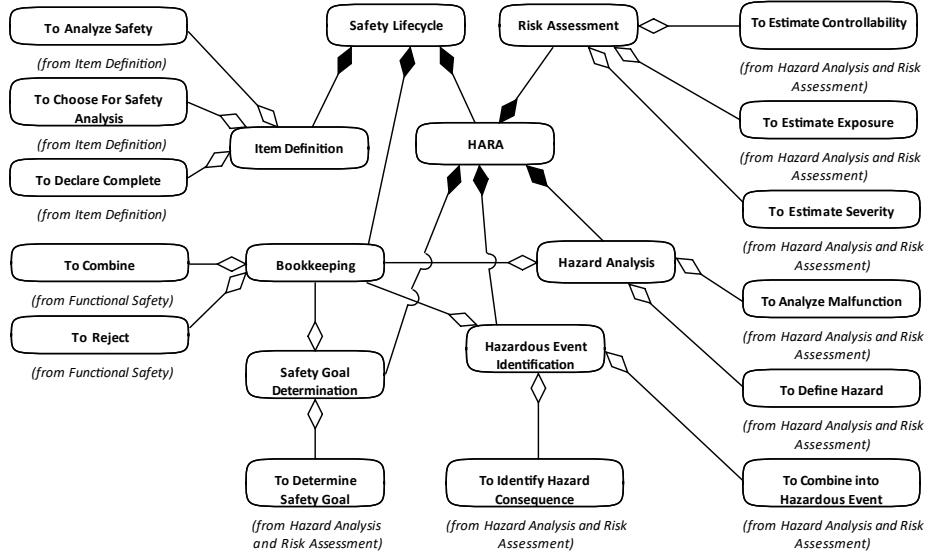


Figure 5.20: Final *feature structure* (UML notation)

models. Section 5.7.3 discusses how feature modeling fits into the rest of MuDForM. The usability of UML for MuDForM is discussed in Section 5.7.4. Finally, Section 5.7.5 discusses how the case study helped to realize the objectives of TNO.

5.7.1 Support for Domain-based Specifications

This section discusses the methodical support for specifying feature models in terms of domain models (RQ4.2 from Section 5.1.2). Objective O5 of MuDForM is to support the separation of what can happen from what shall happen, *i.e.*, distinguish descriptive domain specifications from prescriptive domain-based specifications. The SLR of Chapter 2 concluded that the existing domain-related literature provides little support for the methodical use of a domain specification (DM or DSL). This chapter shows how MuDForM provides such support.

In our case study, we modeled clauses of part 3 of the ISO26262 as functions in the feature “Safety Lifecycle”. We followed the MuDForM method steps and applied the guidelines. The results are 1) a feature structure, 2) a set of function lifecycles, and

3) a set of function signatures. The domain model elements have been applied in the specification of functions, *i.e.*, the domain activities are invoked via function steps in function lifecycles, and the domain classes are used as types of function attributes. The domain model elements are also used in the formulation of invariants and preconditions in the function signatures and function lifecycles. Moreover, the concepts of the System Design context model are used in the specification of the Item definition function. This function is the place where concepts from outside the scope of Functional Safety, *i.e.*, System and Design Object, are used to define the main concepts for the Safety lifecycle, *i.e.*, the Item and its Safety Design Objects.

The result is that all the terms in the feature model are expressed in terms of the domain models and context models, and all views have an unambiguous interpretation. This is an advantage above the textual description of the standard, because it enables safety engineers to have a shared and unambiguous interpretation of the standard. It also enables the engineers to demonstrate the application of the standard, through logging all the executed function steps, especially if this happens with a tool. Safety auditors can check the log file to validate if the standard is followed.

The clearest example of ambiguity reduction is the identification, specification, and integration of the Bookkeeping function, which is not explicitly present in the standard's text, and the formalization of the included constraints (see Section 5.6.4). Moreover, if some readers find the diagrams less comprehensible than the text of the standard, then it is possible to adjust the standard's text based on the inconsistencies and gaps that were detected and solved during the modeling process.

We observe that the resulting model of the case study resembles the input text in both structure and terminology. This resemblance is not surprising, because the ISO26262 has evolved for several years and is used in industry by automotive engineers, which has led to a mature structure and text. Furthermore, it is structured via processes and contains an explicit vocabulary, which respectively correspond to the structure of the feature model, and the terms in the domain model. This maturity of the standard could be seen as a disadvantage for the case study in the sense that the helpfulness of all the MuDForM steps and guidelines could not be fully demonstrated, because the text was relatively easy to translate into a MuDForM model. This situation differs from many industry projects where textual specifications are written specifically for such a project, and are not the result of years of reviewing and re-engineering. The maturity of the text was in a way also an advantage for the case study, because there was little delay caused by long discussions between domain experts, whose availability was limited.

The case study showcases how all the steps in the method flow are performed. In the modeling phase from case initiation to initial model, there are no separate steps related to feature modeling, because texts usually do not distinguish between contexts, domains, and features. But there are guidelines that specifically have an impact on feature models. During *Model engineering*, there is a separate step for *Engineering a feature*, including specific guidelines.

We find that it is possible to automate part of the text-to-model process, *i.e.*, the grammatical analysis and the text-to-model transformation. However, the involvement of domain experts in the grammatical analysis process is essential, because they do not only provide missing information and help eliminate homonyms and synonyms, but often feel more comfortable discussing natural language sentences than they are with discussing graphical models, because those have their own –unfamiliar to most domain experts– notations and semantics. The paper from Hoppenbrouwers *et al.* [161], which is based on the KISS method, makes a claim for partially automating the text-to-model phase, such that domain experts are still actively involved via natural language. So, the text-to-model phase can be partially automated, but not completely, because they require case-specific analysis decisions.

During modeling, we observed that some guidelines can be considered as separate method steps. For example, *Define a function attribute for central objects* is now defined as a guideline. It could, however, also be defined as a separate step *Define function attributes* with a guideline *Check for the central objects*. This refinement of the method steps could be useful, *e.g.*, for building a modeling tool that actively leads the modeler through the method steps. The content of the method would still be the same, though. At this moment, we do not have hard criteria to decide whether something is a step or a guideline, because in the development of MuDForM we started identifying steps and gathering guidelines separately. Later, we assigned guidelines to steps and discovered in practice that some guidelines could be seen as a step. We plan to develop the criteria and refactor the method flow and guidelines.

During the specification of the function steps of some functions, we identified an improvement for the tool support. Although the amount of possible function attributes that can be participating in each function step is not limited, there is often only one candidate attribute for each function step parameter. For example, the function steps To Analyze Safety, To Choose For Safety Analysis, and To Declare Complete in the function lifecycle of Item Definition all require an Item (see Figure 5.13) However, there is only one Item in scope, namely the function attribute the Item. Tool support could provide for the automatic allocation of function attributes to function step

parameters in the cases where there is only one candidate. This would reduce the manual modeling work for specifying function steps.

Although during the evaluation design, there was no explicit planning of reviewing the method definition, interested peers and people involved in the case study have been invited, and reviewed the method definition. The experience is that people find it hard to review the method definition, because many people find it hard to understand how the modeling concepts have to be used and what they exactly mean from just the UML class and activity diagrams in the method definition. The guidelines are easier to review. Reviewers have mainly helped to improve the clarity of the modeling guidelines. The most feedback came during face-to-face explanation of the method ingredients during the modeling activities, and not via reading the method definition documentation. We think there should be a handbook for practitioners, including more example cases, to convey MuDForM to a wider audience.

5.7.2 Bridging the Gap between Feature Trees and Domain Models

We argue that MuDForM feature modeling can be applied to bridge the gap between approaches that focus on modeling feature structures (top-down), like FODA [90] and ADOM [134], and approaches that consider domain models as a language to define other specifications (bottom-up), like the security domain model by Firesmith [58], and the many examples from van Deursen *et al.* [43]. The MuDForM part presented in this chapter can unite these perspectives. The functional decomposition in the feature structure viewpoint of MuDForM is similar to the feature model in FODA, and by specifying the functions in the feature in terms of the domain model, the gap is bridged. Via the function attributes and the function lifecycles, the two specification areas are connected. The advantage for the top-down approaches is that it enables their models to be the entry point for system development. Namely, the addition of function attributes and function lifecycles turns the feature structure into a system behavior specification. The advantage for the bottom-up approaches is that the path from their models (and languages) to a system specification becomes systematic.

In addition, MuDForM feature modeling and domain modeling can be used to formalize the process models from approaches centered around process modeling languages, such as BPMN [1, 30]. The MuDForM steps and guidelines help to organize the modeling activities. Furthermore, the use of an explicit domain model, with both domain classes and domain activities, facilitates the formalization of the process flows and process artifacts. As a result, the process models are fully expressed

in well-defined terms, like in the case study of Section 5.6, which improves their usability for process engineering and implementation.

The steps and guidelines for feature modeling could support modeling with the Object Process Methodology (OPM) [45, 46]. OPM integrates behavior concepts (processes) and state concepts (objects). It also allows composition for both types of concepts and has viewpoints similar to the feature signature and feature structure viewpoints of MuDForM. OPM has the Object Process Diagram, which is similar to the interaction view of a MuDForM domain model [93], and to the function signature viewpoint introduced in Section 5.5.2.4. To our knowledge, OPM does not have a viewpoint to model the internal behavioral structure of a process, like the function lifecycle of MuDForM. Nor does OPM explicitly distinguish contexts, domains, and features. Modelers using OPM could benefit from the steps and guidelines of MuDForM, and vice versa. As such, combining MuDForM and OPM could increase the application scope of both approaches.

It is the concept of domain activities, and the distinction between prescriptive feature models and descriptive domain models, that enable the unambiguous specification of the process flows. Due to making feature specifications and process models domain-based, they become unambiguous and fully integrated with data (object states). In combination with explicit context models, it follows that there are no undefined terms in a feature specification. Furthermore, the domain activities and their relation with domain classes enable an easy verbalization of a model, which improves its conveyance to people who are more text oriented. For example, in Figure 5.17 one can read the phrase “to combine several safety analysis objects into another safety analysis object”. Such a verbalization is very helpful in validating the model with domain experts, who might not easily understand a graphical notation like UML.

The use of domain activities and domain classes could be an addition to BPMN related methods [1, 30]. Those methods do not have normalized literals in their models, *i.e.*, they model process steps, but the steps have no explicit type with well-defined properties. For example, where in the example of Section 5.6.4, “to Combine” is the type of the step Bookkeeping.toCombine, in pure process-oriented methods, one could not build on the predefined feature-independent definition of something like the domain activity “to Combine”. We cannot, however, fully make this claim, as we did not perform a complete literature review on the use of types and references in process modeling approaches. We plan to carry out this review after having gathered more guidelines for function and process modeling from the existing literature.

Although not demonstrated in section 5.6, not only the processes, but also the process artifacts can be defined completely in terms of the domain model. Each artifact can be specified as a composition of domain model elements, similar to the use of domain model elements in the constraint example of Figure 5.16 in Section 5.6.4. For example, a HARA document can be seen as a query on a repository with a database scheme that is based on the domain model. Also here, having explicit domain activities plays an important role, because they facilitate the logging of all elementary changes during a HARA execution.

5.7.3 Feature Modeling is Part of MuDForM

This section discusses how the support for feature modeling fits into MuDForM (RQ4.1 from Section 5.3). MuDForM is based on the KISS method [96], which already has support for function modeling. Differently from MuDForM, however, the KISS method does not provide support for grammatical analysis related to functions, nor does it cover the notion of (MuDForM) feature, or provide detailed steps, guidelines, and viewpoints for feature modeling.

Section 5.4 presents the foundation of MuDForM and how the steps and guidelines for feature modeling fit into the overall steps and guidelines of MuDForM. The partial metamodel of Figure 5.5 addresses minimally how the modeling concepts fit.

We observed the following regarding the method flow: already in Scoping, feature modeling is addressed, because the purpose of a feature model mostly differs from the purpose of a domain model. During Grammatical analysis, MuDForM offers guidelines for the identification of functions, function attributes, function steps, and constraints. In the transformation from text to model, there are guidelines for the identification of typical function attributes, *i.e.*, for the central objects in a feature, and for the context objects that the feature uses as input. The transformation from text to model introduces the creation of two views: 1) the feature structure, and 2) the function signatures. During Model engineering, feature engineering is a step, which is typically organized separately. However, feature engineering might lead to changes in the domain models and context models. Namely, if a term in a feature model is not defined, then it is not the correct term and thus may not be used, or it must be integrated in a domain model or context model. This way, feature modeling validates the domain model and context model. This is the main reason for defining the feature model engineering step in parallel with the steps for domain models and context models, instead of ordering it sequentially. The latter would be more

logical when a domain model is stable and is used in the specification of multiple features, because then the feature modeling activities will not involve the adjustment of domain models. For now, this simply means that the domain modeling branch is not used in those cases. In conclusion, we did not detect a point where feature modeling does not fit or is inconsistent with the rest of the MuDForM definition.

With MuDForM, a domain object, *i.e.*, instance of a domain class, can only change state through participation in a domain action, *i.e.*, instance of a domain activity. The effect is that functions do not need to take care of preserving correct object states. But, they need to take care that the objects that participate in a function step comply with the preconditions of the step and the type of the step, which might be a domain activity. The general idea is that the what-must-happen specification in the feature model may not go outside the boundaries set by the what-can-happen specification in the domain model. In other words, the domain model forms a design space for the feature model.

The above discussion only pertains to the integration of feature modeling in MuDForM. We think that similar constructs should be applied when the support for feature modeling is integrated in other domain modeling methods. The following describes the general aspects of such an integration:

- **On the metamodel.** The metamodel has modeling concepts that are specific for feature modeling, which must be related to the concepts for domain modeling, and possibly to other modeling concepts as well, *e.g.*, analogous to the context modeling concept in MuDForM. This is not just a matter of relating concepts to each other, but also the semantics must be aligned. For example, most domain modeling methods do not have autonomous modeling concepts for specifying behavior, similar to the domain activity concept in MuDForM. They just model classes and attributes, and relations between classes, and often capture behavior in generic data-oriented operations like create, update, and delete. The feature modeling metamodel in this chapter uses behavioral elements, *i.e.*, operations, domain activities, and functions as elementary modeling concepts (see Section 5.4.3). They are used as the types of the steps in a function. If the domain model only has classes, then the function steps will be a combination of create, update, and delete operations. One can solve this issue, by modeling low-level functions that serve as a surrogate for the domain activities.
- **On the notation and viewpoints.** The case study uses UML for the notation of MuDForM concepts. But MuDForM itself does not prescribe a specific

notation. When feature modeling is integrated with another domain modeling method, it is possible to choose a notation that is close to the existing notation of that domain modeling method. For example, a text based notation can be chosen for the function flows, which resembles the notation of any imperative programming language like Java or Python. Of course, a notation must be defined for all the viewpoints described in Section 5.5.2.

- **On the method steps.** The four main steps of the MuDForM method flow (Figure 5.2 can be generalized into: Scoping, Discovery and Elicitation (for capturing specific knowledge from a knowledge source), Switch to modeling, and Model engineering. Scoping, and Discovery and Elicitation do not require specific steps for feature modeling. However, the scope of a feature model is inherently different from the scope of a domain model. So, the knowledge that is captured from a knowledge source is different for a feature model than for a domain model. The steps Switch to modeling and Model engineering will have detailed steps that are specific for feature modeling, because feature modeling has different viewpoints than a domain model, and the modeling steps typically correspond with viewpoints. We think the steps described in Figure 5.7 can be used unchanged in integrations with other methods.

To put the method part of this chapter in perspective of the whole method: the MuDForM metamodel contains 61 classes, the total method flow has 33 steps and sub-steps, there are 12 different viewpoints, and there are currently 125 guidelines. At this moment, the method's ingredients differ in maturity and completeness. Namely, the metamodel is quite stable, and the method steps sometimes change, but guidelines are still frequently discovered, discussed, and specified more concisely. The latest version of the complete MuDForM definition is available online [33].

5.7.4 Reflection on Using UML for MuDForM

In the case study, we used UML [120] and Enterprise Architect [152] as the modeling tool. We used the standard notation and semantics as much as possible. However, for some modeling concepts and viewpoints, we *misused* a UML concept, by giving it a MuDForM meaning. These are our findings:

- There is **no UML diagram to model the allocation** of function attributes to (actual parameters of) function steps. We used the UML concept Object Flow, but this does not cover the MuDForM semantics. Namely, a function attribute,

which is a reference to an object, participates in a function step; it does not flow in or out the function step. Moreover, there is no syntax to state explicitly which parameter of the function step a function attribute is allocated to.

- **Activity diagram coordinators, *i.e.*, fork-join and decision-merge are not suited** for the MuDForM metamodel, which is based on process algebra. Coordinators for selection and concurrency are already offered in other methods [168, 96], and they can be used. For example, the process algebra counterpart of Figure 5.11 is “(Item Definition | HARA)*”. So, we perceive UML activity diagrams to be cumbersome for representing processes, because they are derived from the concept of state transition diagrams, which were already present in earlier UML versions. The process algebra notation does not require arrows for the loop, nor an explicit “merge” symbol, which leads to diagrams with fewer nodes and edges.
- **The used UML concepts offer more freedom than MuDForM requires**, which implies that the modeler needs to manually guard if relations that can be drawn between two model elements in the UML tool, are really allowed according to the MuDForM metamodel. For example, a feature model may not directly contain classes. But, as we use packages to represent feature models, the used UML modeling tool (Enterprise Architect) allows declaring a class in such package. So the modeler needs to prevent this manually. A better solution would be to extend Enterprise Architect with consistency checks for MuDForM models. These checks could be based on formally specified consistency rules, for example in OCL. Some of these rules are now captured in the method definition [33] as postconditions of the corresponding method steps.

We find UML usable for a large part of the feature modeling view, but not so much for the aspects mentioned above. This is one of the reasons why we are currently collaborating with another company to build a MuDForM modeling tool in MPS (Meta Programming System) [89].

5.7.5 Reflection on Case-specific Objectives

The objective of TNO, which is mentioned in Section 5.6.1, is realized via several MuDForM characteristics:

- The process models, *i.e.*, steps and object flows, are unambiguous now, because they are completely defined via concepts in the MuDForM metamodel.

- The processes are completely defined in terms of the domain model and the context models:
 - The process steps are expressed as invocations of domain activities, or invocations of other functions, and the modeling process has eliminated homonyms and synonyms, which makes the process steps unambiguous.
 - The objects (data) that are handled by the processes are expressed as function attributes, which are instances of domain classes or context classes.
 - The requirements in the standard are expressed as constraints, and defined in terms of the context classes, domain classes, and their attributes. They are attached to the model as a function invariant, function precondition, guard, or pre- or postcondition of a function step.

TNO also used the model of the case study to build a tool for compliance testing. This tool gives two benefits. First, The domain classes, and their attributes and relations, are used for the data structure and user interface terminology of the tool. The artifacts prescribed by the standard are corresponding with the function attributes defined in the function signatures (see Section 5.6.5). This encoding of the model in the tool helps unify the terminology for the safety engineers of TNO and its customers. The second benefit is that the tool provides a formalization of requirements of the standard, *e.g.*, the constraint in Section 5.6.4. They are implemented in the tool as validity checks [17] on the imported functional safety artifacts, and allow the tool to automatically create a (partial) ISO26262 compliance statement. This has increased the objectivity and uniformity of the certification process.

5.8 Threats to Validity

As highlighted by Petersen *et al.* [121], action research yields three main validity concerns: 1) context dependency (which is intrinsic to action research and hence “*can never be really mitigated, but it can be reduced [...] when lessons learned may be transferred to similar contexts*”), 2) bias of the researcher (which can be mitigated by involving multiple researchers and/or feedback from external experts), and 3) the time factor, *i.e.*, learning and changes in the context (which can be mitigated, again, by involving multiple researchers and being aware of major changes).

Petersen *et al.* map the above threats to external validity and internal validity. Accordingly, we follow the definition of Wohlin *et al.* [170] to report the related potential validity threats that we identified, and the measures we adopted to mitigate them.

5.8 Threats to Validity

External Validity concerns the generalizability of the study. We identify two main limitations to external validity, regarding the set of guidelines we built, and the general applicability of MuDForM.

We make no claims to the general applicability or completeness of the guidelines. To mitigate this threat, we share them to be reusable and extensible so that they mature for general applicability. In fact, the guidelines are continuously adapted and extended. During modeling activities, we write down reasons for decisions when we notice them. We then let them review by others, in particular by people involved in the modeling process, and possibly formulate it as a guideline. The guidelines are linked to a modeling step, which makes it possible to only explain the guidelines to involved people at the beginning and during a specific modeling step. This ensures dosed learning of the guidelines.

The guidelines are written in terms of metamodel elements and, if necessary, complemented by commonly-used terminology from outside the metamodel. As such, we cannot ensure general-understandability. As a mitigation, however, we use common terminology that 'anybody in the field' should be able to understand. To verify this, the method steps have been applied in this and in other studies, and the guidelines are reviewed by several practitioners.

Concerning the general applicability of MuDForM, the method is designed to be domain independent (cf. objective O2 in Section 1.1.3). A possible limitation is that the method takes a textual description as input. As a mitigation of this potential threat, the step of extracting phrases can also be applied in interviews with domain experts, to acquire a set of usable input sentences. About domain experts, in this specific study we have involved a member of the ISO26262 standard committee to provide feedback. We are therefore confident that the modeled standard, output of the application of our method, is generally applicable to ensure standard conformity.

Finally, the study might not cover all the method parts, and thus some method parts might potentially be incorrect. More experimentation will be needed to increase method maturity. As a general mitigation, the steps have been executed in other cases too [114, 93, 36], and the guidelines have been reviewed by practitioners.

Internal Validity concerns the causality between the study and its outcomes [170]. We see a limitation regarding the potential bias that the direct involvement of the method developers might have introduced. For example, with respect to our observation that the used UML notation is unambiguous (Section 5.7.1), or that all process artifacts of the modeled ISO26262 processes can be defined completely in terms of the domain

model (Section 5.7.2). To mitigate this threat, we regularly involved other stakeholders to provide feedback, namely ISO26262 standard experts, experts in the automotive domain, ISO26262 users (*e.g.*, for feature engineering), and functional safety tool developers.

5.9 Related Work

Our SLR (see Chapter 2) did not uncover any work on methodical support for applying a created domain specification. However, the SLR started the literature search with a search string that included the term “domain”. So, it might have missed works that are not centered around the domain concept. To make sure we would not miss relevant work, we performed a search on Google Scholar on the following terms in the publication title: ((function) OR (process) OR (feature) OR (use case)). This leads to more than 100,000 hits. Adding “domain” as a mandatory term would not be helpful, as this is already covered by our SLR. For feasibility, we instead checked the top publications ordered by relevance. This resulted in a number of works related to ours, even though none of them covered methodical support for creating domain-based specifications. The following reports on them.

On process modeling. The Object Process Methodology (OPM) [45, 46] also has concepts for modeling state (objects) and change (processes), and has a viewpoint to specify how those are related (object process diagram). It also allows composition for both types of concepts and has some viewpoints similar to viewpoints of MuDForM. OPM distinguishes a descriptive aspect and a prescriptive aspect in a domain, but does not provide methodical support for the specification of the prescriptive models in terms of descriptive models. (Section 5.7.2 elaborates on the relationship between OPM and MuDForM.)

On feature modeling. Lee *et al.* [103] give guidelines for identifying features and state that a domain dictionary is required to assure uniformity across feature names. But they do not give steps and guidelines on how to use such a dictionary.

On use case modeling. Samarasinghe and Somé [115] discuss the extraction of domain models from use cases. But we could not find a paper that shows how to write use cases in terms of a domain model. A paper from Śmiałek *et al.* [151] presents an approach and metamodel to standardize sentences in use case scenarios. However, they do not show a method to do so, *i.e.*, no steps or guidelines are presented. Furthermore, they do not use a domain model, but just a vocabulary.

On functional modeling. We did not find any relevant papers that use a domain model as the vocabulary in functional specifications, except for the aforementioned KISS method [96]. But that book and other publications that refer to the KISS method, e.g., [80, 129] do not provide a metamodel, detailed steps, or guidelines.

5.10 Conclusion and Future Work

This chapter describes the MuDForM methodical support for using a domain model as a language to define other models in general, and feature models in particular; and reports on an industrial case study in the automotive domain.

We were interested in studying the following research questions: **(RQ4.1)** *How should methodical support for making feature specifications be integrated in a method that supports the creation of domain models?*, and **(RQ4.2)** *What methodical support can be given for the specification of a feature, such that it is defined in terms of domain models?* We conclude that the current way in which MuDForM is defined, is applicable to define feature modeling too. That is, we have defined modeling concepts, method steps, guidelines, and viewpoints, and integrated those method ingredients with the rest of the MuDForM definition. Furthermore, we conclude that it is possible to provide guidance for the creation of domain-based specifications. In doing so, we observe that the defined metamodel, and method steps are quite mature, as we did not detect relevant knowledge from ISO26262 that we could not capture. However, more case studies would help to perform a quantitative evaluation of MuDForM, to get clarity on which method ingredients are more or less mature, and which modeling decisions require better guidance. Furthermore, we would also like to have a more elaborate comparison on making domain-based specifications with and without MuDForM, which (of course) would require the involvement of modelers that have no previous knowledge of MuDForM.

The results of our study fill an important gap in the state-of-the-art, which to the best of our knowledge lacks in providing methodical support in the first place. It lays the foundation for our future work on building a significant, validated and reusable set of guidelines for which we plan the following:

- Building a community that actively validates, identifies, and manages guidelines. This would require that people from industry actually use MuDForM, which is not easy to realize. We think it starts with providing training and tooling.

- Searching for guidelines in existing literature, and extending the list of useful publications [96, 138, 2, 82, 47], which were found by our literature review (see Chapter 2).
- Conducting a literature review to find and analyze guidelines from process modeling approaches. We will also look for more literature on OPM, in addition to the already mentioned publications [45, 46].

Regarding method engineering, in our future work we plan to define criteria for deciding whether something should be a method step or a guideline. This could lead to refactoring MuDForM, as mentioned in Section 5.7.1. As suggested in Section 5.7.4, we will consider the formal specification of consistency rules, and maybe the guidelines too, in OCL or another suitable language. Moreover, we are working on a paper that covers the generic principles, design patterns, design criteria with which the method is constructed.

To facilitate industrial adoption, we plan to create a MuDForM handbook for practitioners. We are currently investigating the requirements and possibilities for a modeling tool that supports MuDForM, in order to replace the UML modeling tool that we currently use, *i.e.*, Enterprise Architect [152].

Finally, we want to mention that we, MuDForM researchers and employees of another company, have been working on combining MuDForM and Behavior Driven Development with the Gherkin language [150, 171], to create a language for model-based test specification, which involves combining the MuDForM metamodel with the Gherkin metamodel. This is an example of how to integrate MuDForM with another language. But in this case, the targeted specifications are not only based on MuDForM domain models, but also on MuDForM feature models. We plan to write a paper that reports on this experiment. We are also considering an experiment on domain-based requirements, in which requirements are completely expressed in terms of application domain models, and a requirements metamodel, *i.e.*, a requirements domain model.

5.A Guidelines Pertaining to Feature Modeling

5.A Guidelines Pertaining to Feature Modeling

The guidelines in the table below pertain to feature modeling. They are a subset of the complete MuDFoM definition [33].

Step	Name	Description
Scoping		
Define purpose	Common feature model purposes	<p>When in doubt about the purpose, check if these common purposes for feature models are applicable:</p> <ul style="list-style-type: none">• Provide the terminology for specifying other features.• Provide the terminology for specifying requirements for a system that implements the feature, <i>e.g.</i>, a software application, work process, or hardware.• Form the starting point for deriving specifications in another domain, typically a software domain. In other words, generate code (or models) for a specific target platform. In this case, the model would typically serve as the source model for transformation rules.• Provide terminology for specifying tests to verify if a system works according to the feature model.• Provide actors with work instructions. Actors can also be (software) systems, in that case the feature model can be seen as a functional system specification.
Define purpose	Different specification spaces have a different purpose	Write a purpose specification for each (expected) specification space. A domain model has typically a wider applicability than a feature model, which implies that it has a different purpose.
Select input text	Start with the foundation and the core concepts	When a text is too large to take in at once, then the selection can be narrowed (initially) by selecting the parts of the text that are needed for understanding other parts. This might require knowledge of the text, or at least some initial analysis to see the dependencies between parts (chapters, sections, paragraphs) of the text.

Chapter 5. Domain-based Feature Modeling with MuDForM

Step	Name	Description
Select input text	Start small	Limit to 50 sentences for a first iteration. This helps to quickly get an initial model. After the transformation from text to an initial model, one can choose to start the <i>model engineering</i> , or to first add more sentences.
Grammatical analysis		
Extract phrases	Keep subjects of interaction phrases if it is an object in other phrases	Check if the subject of an interaction phrase occurs as object in other phrases. In this case, the phrase often means “The actual subject/actor observes that”. One can maintain the original phrase structure, but the subject will not become a candidate actor, but most likely a candidate domain class. Example: In a toll registration system, “The vehicle passes the toll booth” could be rewritten as “The system observes the vehicle passing the toll booth”. But the original phrase is more natural and can be kept. And the subject “vehicle” will most likely occur as an object in other phrases, causing it to be a candidate domain class.
Determine relevance	Ignore phrases about the document itself	Ignore phrases that are about the document itself, like an explanation of the document structure, or sentences that “glue” paragraphs together. For example, an extracted phrase like “TO explain <some topic> in chapter”, or “TO summarize document in summary” can be ignored. (Unless the domain is about writing reports).

5.A Guidelines Pertaining to Feature Modeling

Step	Name	Description
Eliminate homonyms and synonyms	Replace generic verbs with a domain specific term	<p>Be aware of generic verbs. These are often data-oriented verbs or verbs that are easily applicable to a neighboring domain. Examples of data-oriented verbs are:</p> <ul style="list-style-type: none"> • Create, identify, enter, define, describe, register, select, add. • Update, adapt, change, modify. • Delete, terminate, erase, remove, end. <p>These verbs are typical for administrative and conceptual objects. Preferably, use a more meaningful term from the actual domain. For example, “change address of a person” is actually “person moves to a new address”, or “enter an order” becomes “place an order” or simply “to order”, or “change the color of the wall to blue” is really “paint the wall blue”. The verb term may be overloaded, when there is no good alternative available, according to the domain experts. For example, to describe a person could be seen the same as the same activity as to describe a dog. But in case you consider it to be different, you can postfix the general verb with the direct object, resulting in “to describe person” and “to describe dog”.</p>
Eliminate homonyms and synonyms	Standardize logical constructs	<p>Logical constructs are not always formatted uniformly in the input text. Sometimes punctuation is used to construct sentences containing the semantics “if-then-else”, “for all”, “implies that”, and “or”. By adding a clarifying keyword like “then”, or using parentheses, it becomes clear which interpretation is meant. It also holds for operations like “is equal to” or “has the same value as”. Use a uniform syntax to express conditions and operations in corresponding phrases. For example, replace “when it rains, take an umbrella” with “if it rains, then take an umbrella”.</p>

Chapter 5. Domain-based Feature Modeling with MuDForM

Step	Name	Description
Text-to-model transformation		
Classify candidates	Identify features and functions from text headers	Text headers often indicate the name of a function or feature, because texts are typically written as a coherent chronological series of events.
Classify candidates	Identify functions from use case interactions	If use cases, user stories, system (interaction) scenarios, are used as input text, then the steps that describe system behavior are often calls of system functions.
Classify candidates	Definite articles and indexicals indicate an (activity) attribute	A definite article, <i>i.e.</i> , “the”, might point to a role an object plays in a function or in a domain activity. Classify it as an activity attribute with a (domain) class as a type. The same holds for so-called indexicals. The standard list of indexicals includes pronouns such as “I”, “you”, “he”, “she”, “it”, “this”, “that”. They indicate that there is an object that participates in several steps in a lifecycle. Such an object must probably be represented by an activity attribute. In the case of a domain class, the indexical can also refer to the domain class itself, which is the container of the object lifecycle.
Declare and allocate elements	Auxiliary verbs indicate the specification space type	Auxiliary verbs can be an indication if a phrase belongs to a feature or to a domain. Verbs like “will”, “can”, and “be able” indicate that it is content for a domain model. Verbs like “must”, “shall”, “should”, and “ought to” indicate that it is content for a feature model.
Identify specification spaces	Begin with one context, one domain, and one feature	If there are no existing specifications spaces and there are no obvious boundaries, then start with one context, one domain, and one feature.
Identify specification spaces	Separate contexts for domain definition from contexts for feature interaction	Separate concepts for defining domain class attributes, domain activity attributes, and operations in activity models, from concepts that are needed to specify the interaction of features with their environment. Examples of the latter are external actors and operations that are the type of function events.

5.A Guidelines Pertaining to Feature Modeling

Step	Name	Description
Create initial model, Specify function signatures	Introduce feature attributes for sets of existing objects	A feature is often activated in an environment of sets of (independent) objects. Such sets of objects often serve as the pool of objects to select from for participation in function steps. Introduce set attributes for those objects in the feature or in the top-level functions.
Create initial model, Specify function signatures	Introduce feature wide attributes for the central objects	Features, and sometimes top-level functions in the feature, often center around one or more central objects, which are referred to via a function attribute. Such an attribute can be global in the feature (or the top-level function) to prevent that other functions must define it separately as a function attribute.
Model engineering, Engineer feature		
–	Start specifying functions for domain classes that are not a part of a composition or aggregation	If there are not already functions defined on a domain, then at least functions are needed to create and manipulate the objects that are not dependent on other objects; the so-called strong objects. Namely, those objects are needed to instantiate the weaker objects that are dependent on them.
–	Define a separate function for function steps sequences that occur more than once	Like normal functional decomposition used in programming or a function-oriented modeling method, a functional decomposition is handy when several higher-level functions contain the same sub-behavior. This common sub-behavior may be captured in a separate function.
–	Define a function for coherent behavior that will be assigned to one actor	Define functions for units of behavior, often called tasks, that will be assigned to and performed by one actor.
–	An atomic object manipulation might indicate a domain activity	During feature specification, one may find functions that manipulate a single object. Take into consideration if such a function should be defined as a domain activity. If so, it should be allocated to the domain model. If the behavior is about what can happen and not about what must happen or how does it happen, and if the behavior is independent of the feature, and any actor that performs the function, then it might be a domain activity.

Chapter 5. Domain-based Feature Modeling with MuDForM

Step	Name	Description
–	Find functions from system specifications	<p>System functions can be found from several perspectives:</p> <ul style="list-style-type: none"> • System use cases indicate a system function, and steps in the use case scenario indicate lower-level functions. • sub-systems in a system architecture often indicate a high-level function. • A decomposition of the system requirements may indicate functions and sub-functions. • Chapters or aspects in a requirements document indicate features or high-level functions.
Specify feature structure	Cover the relevant domain activities	<p>Go through the domain activities of the relevant domain models and check if they should be used in the feature. It is not that all activities must be used, because a feature might only cover a domain partially. But if activities are not used at all, they might be forgotten in the feature model, or might not be a domain activity at all.</p>
Specify function lifecycle	Go with the flow	<p>Begin with the major function steps:</p> <ul style="list-style-type: none"> • The activities and functions that must be executed in the function. • Their temporal ordering: sequence, selection, parallel, iteration. <p>Initially skip:</p> <ul style="list-style-type: none"> • Constraints of steps (enter criteria and exit criteria). • Decision logic of coordinators (guards of selections, forks, iterators). • Decision logic of step participants, <i>i.e.</i>, constraints on the function attributes that are allocated to step participants.
Specify function lifecycle	All function events must occur as a step in a function lifecycle	<p>All the function sub-behaviors must occur at least once as a step in the function lifecycle. Otherwise, it would not be a function event. In other words, if a function is dependent on other activities, then those activities should also be used in the behavior specification of the function, <i>i.e.</i>, in the function lifecycle.</p>

5.A Guidelines Pertaining to Feature Modeling

Step	Name	Description
Specify function lifecycle	Temporal words in the input text indicate temporal order	Temporal words in the input text are a hint about the passage of time or the position of an event in time, usually indicated with a transitional preposition (<i>e.g.</i> , after, before, during, until). Other temporal words can also be a hint, <i>e.g.</i> , now, eventually, suddenly, initially.
Specify function lifecycle, Specify function step	Make sure that used domain activities comply with the domain model	For each domain activity that is used (invoked) in a function step, immediately cross-check the domain activity with the domain model. Is the domain activity also present in the interaction view? Does it have the same objects associated with it (using the same prepositions, <i>i.e.</i> , the same activity roles)? Postpone other cross-checks with the domain model until the function lifecycle is completed. By doing the domain model check immediately, the domain model is validated as well, and it is assured that all the domain activities usages conform to the domain model.
Specify function lifecycle, Specify function step	Ensure the model consistency between constraints that involve the same domain class	Find all the constraints that are relevant for a function step for each of the involved (domain) classes. These are not only the constraints directly connected to that step, but also the ones that are specified at a higher aggregation level, <i>e.g.</i> , as an invariant of a container function. Then verify if they are free of contradictions.
Specify function lifecycle, Specify function step	Default allocation of function attributes to step participants	Often one domain class will only occur just once as the type of a function attribute within a function. That means that such an attribute is probably the attribute that will be allocated to all function step participants that have that domain class as type. In practice, this guideline suggests that those step participants don't need to be allocated manually, but could get a default allocation automatically.

6 From Vision to Method: the Cognition Perspective

“We strive for complete
comprehension
Search for meaning in the words
Relieve concepts from detention
Found in monkeys, wuggs, and birds”

Lingo, Neuro, Psycho,
Students of the summer school on
human language, July 2016^a

^aFull lyrics in appendix B

Chapter 6. Cognition Perspective on MuDForM

This chapter addresses RQ5 of this thesis.

Context. We envision a software development process in which involved people capture knowledge and decisions in a way that is close to how they communicate and think. Therefore, we are investigating a method, called MuDForM, to formalize and integrate knowledge of multiple domains into domain models and into specifications in terms of those domain models. We have defined and followed an approach for the creation of the method, which has led to the first version (see Chapter 3).

Goal. Put the method and its development in the perspective of cognition literature, in order to evaluate to what extent the method is cognition-based.

Method. We started reading and listening to literature on linguistics, philosophy, and cognitive science, mainly hinted by domain experts, and we iteratively added referred and citing literature to the reading list. We bookmarked sections that we considered relevant for the method and its development. We wrote a finding for each bookmark, and allocated it to one or more categories related to the method definition and its creation. We analyzed how each finding implicates the method and its creation, and grouped the results in different themes.

Result. We read and listened to about 150 books, of which 29 led to 80 bookmarks with findings. Their analysis led to 69 reflection snippets, which underpin or challenge method definition choices, or suggest considerations or improvements. The snippets are grouped in eight themes, which form a cognition-based reflection on each concept in the method definition. We conclude that the reflection breaks ground for making software development cognition-based; software development can benefit from synergy between method engineering and cognitive sciences.

Contents

6.1	Introduction	199
6.1.1	Problem Statement and Contribution	201
6.2	Method Creation Context	201
6.3	Research Method	202
6.4	Reflection on the Research Background	204
6.4.1	Use Domain-oriented Models to Convey Knowledge	204
6.4.2	Meaningful Cognition-based Models	205
6.5	Reflection on the Method Definition	207
6.5.1	Separation of Syntax and Semantics in Mind and Method	207
6.5.2	Supporting Analysis, Design, and their Context	208
6.5.3	Model Engineering through Method Engineering	210
6.5.4	Match Reality with Useful Categories	213
6.5.5	Time is Perceived through Events	215
6.5.6	Building Modeling Concepts from Natural Language	216
6.5.7	More Cognitive Aspects and Derived Modeling Concepts	218
6.6	Discussion	220
6.6.1	Cognition-based Method Definition	220
6.6.2	Different Types of Reflections	222
6.7	Related work	222
6.8	Limitations	223
6.9	Conclusion and Future work	224

6.1 Introduction

Software development is a process of integrating human knowledge and decisions. There is no transportation or transformation of material. All development activities and results are conceptual. In our experience, existing programming and specification languages do not support (most) people in capturing their knowledge and decisions. That is why we started our research to create a method, including a language. We set out to find fundamental concepts from human cognition, with natural language as an entry point. We learned about the notion of Mentalese as the language of the mind [125, 123]. But, Mentalese is not defined, and there is no consensus

Chapter 6. Cognition Perspective on MuDForM

among linguists and cognitive scientists on what its concepts or grammar are. As expected, the search for cognitive concepts was not that simple.

Meanwhile, we wanted to work on the actual method. So, we first looked into existing domain-oriented methods. We decided to start from a method that is close to natural language: the KISS method [96]. We defined its metamodel, method steps, and viewpoints, because they were not well-documented. From our 30 years of modeling experience, we documented guidelines and let them be reviewed. The KISS method provides no support for dealing with multiple domains, and for using existing methods and languages. We had designed solutions for these topics in industry projects, but we never generalized and documented them before. We applied and improved the method through case studies.

In parallel, we found elementary concepts in cognition literature, which fitted the defined method. We called those the cognitive aspects (see Section 3.4). Along the way, we produced findings for the MuDForM definition, which underpin, refute, or just challenge some method design choices, or suggest method improvements or future research. This chapter reflects on those. The result is far from complete with respect to the wide span of cognitive literature, and it raises many potential research questions. However, we believe that we opened the door to making software development cognition-based.

This chapter is not technical. It illustrates how the fields of software development and cognitive science can join forces. The vision of MuDForM is a development method that is based on human thinking and communication. It should capture human knowledge and decisions, and produce specifications suitable for system development, and software development in particular. It is fine if this is partially achieved by utilizing existing computer-based and mathematical methods, languages, and formalisms. But those are not the starting point or the foundation.

The chapter is organized as follows. Section 6.1.1 describes the problem statement and contribution. Section 6.2 clarifies the method creation concepts that form the context of this chapter. Section 6.3 explains the followed research methodology. Section 6.4 reflects on the vision, the main research goal of this thesis, and some of the method objectives. Section 6.5 reflects on the concepts that make up the MuDForM definition. Sections 6.4 and 6.5 are based on the reflection snippets from Appendix A. Section 6.6 discusses the results. Section 6.8 analyzes the limitations of this study. Section 6.7 discusses related work. Section 6.9 concludes this chapter, and lists the main suggestions for future work.

6.1.1 Problem Statement and Contribution

This section describes the problem that we address in this chapter, and our contribution. The goal of defining a specification method that is based on knowledge from cognitive and related sciences, is achieved in three tasks. The first task is the identification of useful concepts from cognitive science. To our knowledge, there is no published and agreed upon set of cognitive concepts that can be used to design a (specification) language or method. We need to identify those concepts ourselves. The second task is determining how identified cognitive concepts can be used in the method definition. The third task is defining the method, using the identified cognitive concepts.

This chapter contributes to the first and second task. It provides a reflection from a cognitive science perspective on the creation of MuDForM, *i.e.*, it identifies cognitive concepts, it underpins or refutes some of the method design choices, and it suggests topics for further research and improvements for the method. This reflection can be used for future work on MuDForM, or other cognition-based methods. Furthermore, this chapter shows how knowledge from cognitive science can be used in the creation of a specification method, which can be used by other method developers. Chapter 3, and in particular Section 3.5, addresses the third task.

This chapter does not claim to be complete regarding the potential of using cognitive science in software development, and method engineering in particular. Rather, it aims to break ground for more direct support for human thinking and communication in (software) development.

6.2 Method Creation Context

The method definition and development approach for MuDForM form the context for the reflection presented in this chapter. Although we have tried not to assume detailed knowledge of MuDForM, we cannot avoid that this chapter refers to concepts defined in other sections of this thesis. It might be useful to consult or read them to understand this chapter. The concepts are presented in Figure 6.1. The numbers point to the sections where they are described. We refer to instances of the concepts in the following ways:

- The **Method objectives (O_i)** are represented by their number, given in Section 1.1.3.

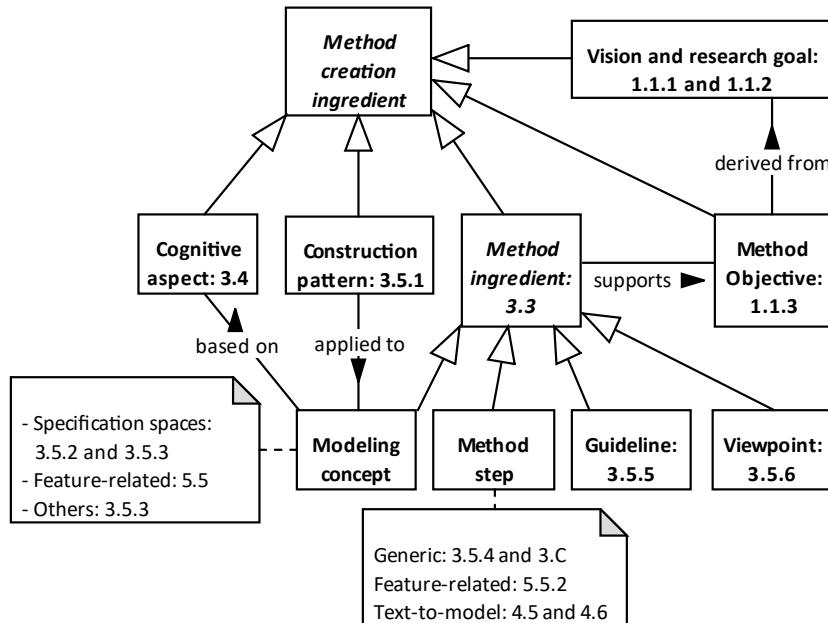


Figure 6.1: Method creation ingredients (UML class diagram)

- The names of **Cognitive aspects**, described in Section 3.4, are in *typescript* font, e.g., *Concept*, *Category*, *Analogy*, and *Statement* context.
- The names of **Modeling concepts**, introduced in Section 3.5.3, and defined in Table 3.3, are in *italics sans-serif* font, e.g., *Classifier*, *Activity*, and *Function step*.

6.3 Research Method

This chapter addresses RQ5 of this thesis:

What are common elementary cognitive concepts that people use to think and communicate, and how can they be applied in engineering a specification method?

To answer this question, we followed an approach that can be described as qualitative data analysis. This research method involves examining non-numerical data, such as texts, to uncover patterns and meanings (see, for example, the book of Huberman

6.3 Research Method

et al. [81]). The steps typically include data collection, data organization, coding the data, analyzing the data to draw conclusions, and finally, reporting insights.

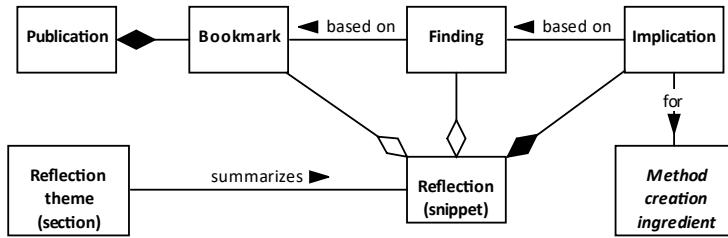


Figure 6.2: Concepts in the qualitative data analysis (UML class diagram)

Our qualitative data analysis is managed via the concepts depicted in Figure 6.2. It consists of the following steps:

- **Data collection:** We searched for literature in the field of philosophy, linguistics, and cognitive sciences, for which the studies from our SLR (see Chapter 2) that addressed the use of natural language, were used as starting point.

A general search on language philosophy and cognitive science, and talks with linguists and cognitive scientists, led us to books of Noam Chomsky [23], Steven Pinker [123], John McWhorter [111], and John Searle [139].

After that, we followed a snowballing approach (as described by Jalali and Wohlin [88]), by iteratively identifying and reading publications, mostly books, of authors that were mentioned in already included literature. We also read some general books about non software design and architecture [116, 68].

It resulted in reading and listening to about 150 books and recorded university lectures in the period April 2015 to December 2023. 29 of those *publications*¹ yielded about relevant 80 *bookmarks*. We considered a bookmark to be relevant when we could think of a way that it could be used in the definition of MuD-ForM, *i.e.*, we could define a finding that has implications for the definition of a method ingredient. When in doubt, we made a bookmark anyway, *i.e.*, we over-bookmarked, because it can always be discarded during the analysis step.

We have created a document titled Cognition Literature Analysis [34], in which we recorded our observations, considerations, and conclusions for each book-

¹ *Italic words* in this section refer to elements from Figure 6.2.

mark, to which we will refer from now on as *findings*. When we were unable define a sensible finding, we discarded the bookmark.

- **Data organization:** We put each bookmark and its finding in a separate *reflection snippet*, from now on simply called *reflection*. We defined categories based on the method creation ingredients presented in Figure 6.1.
- **Coding:** We allocated each reflection to one or more categories.
- **Analysis:** We analyzed how each reflection fits its categories and defined the *implications* for the corresponding *method creation ingredient*. This led to approximately 75 *reflections*, each consisting of one or more *bookmarks*, a *finding*, and an *implication*. When we concluded that multiple reflections had approximately the same implications, we combined them in one reflection, leading to 69 final reflections.
- **Reporting:** When we wanted to report on the results, we discovered that the grouping per category did not lead to a set of clear insights and sensible story lines. So, we regrouped all the reflections in a bottom-up process to come to eight *reflection themes*. Then, we wrote a conclusion and suggestions for each theme. At the start of each conclusion, we listed the used reflections, which are gathered in Appendix A. Considering that cognition-based method engineering is a new field, we have opted to emphasize future research directions more than in other parts of this thesis.

6.4 Reflection on the Research Background

This section reflects on the vision, research goal, and method objectives presented in Section 1.1. We first discuss the use of domain-oriented models to convey knowledge. Then, we introduce the notion of basing modeling concepts on cognitive concepts, with natural language as a starting point.

6.4.1 Use Domain-oriented Models to Convey Knowledge

Used reflections:

- A.1.1: ‘Use unambiguous models to convey knowledge’
- A.1.2: ‘People care about their domain. Computers do not know what they are doing’
- A.1.3: ‘Specifications must mean something to people’

6.4 Reflection on the Research Background

- A.1.4: 'Method steps and viewpoints focus the modeling process'
- A.1.5: 'Modeling is not an extra activity'
- A.1.6: 'Focus on domain-oriented methods'

Books of Pinker [123, 125] and Dennet [42] discuss the computational theory of Alan Turing. Computers do not know what is the relation between their actions and the world outside the computer. For a computer, software is nothing more than a set of instructions; computers just process specifications. To do so, those specifications must be unambiguously interpretable, which requires an unambiguous language. The computer does not require that specifications contain meaningful terms. Meaningful terms are only relevant for people, as addressed by Harari [75], because it enables them to relate specifications to their knowledge and decisions.

Specifications can be about many areas of knowledge, *i.e.*, domains, of a development activity and its context, and may cover many aspects. A modeler should be supported to focus on one or a few domains and aspects at a time. A method that guides a stepwise modeling process, and provides multiple distinct viewpoints, can provide such support.

The reflections mentioned above underpin our choice to investigate a domain-oriented method to define meaningful, unambiguous specifications.

6.4.2 Meaningful Cognition-based Models

Natural language has evolved for thousands of years to allow people to communicate about their world. People also have an internal language for thought, called Mentalese, which differs from natural language. Thought existed before language; it evolved for action.

Used reflections:

- A.1.7: 'Exploit the lexical hypothesis'
- A.1.8: 'Language awareness is innate'
- A.1.9: 'Cognition is more than language'
- A.1.10: 'Modeling concepts should be based on Mentalese'
- A.1.11: 'Focus on mental concepts; not on the brain or reality'
- A.1.12: 'On the evolution of thought'

Levitin [104] discusses the lexical hypothesis, *i.e.*, the most important things humans need to talk about eventually become encoded in language. Language awareness is

Chapter 6. Cognition Perspective on MuDForM

dominantly present in humans. Sigman [145] even concludes that it is innate; even babies are language aware. However, thinking is more than using language.

Pinker [126, 123] rejects genuine linguistic determinism, *i.e.*, a person's language determines what the person can think. He states this is clearly wrong. If it was right, then we could use a metamodel of natural language as the metamodel of MuDForM. But people can and do think without having words or a language-based grammar. They have a kind of internal language for their thoughts, called Mentalese [125]. The MuDForM metamodel should be close to Mentalese, but Mentalese grammar is not defined. The observation that thoughts are inherently unambiguous matches with the requirement that software (specifications) must be unambiguous. Natural language complicates the direct translation of thoughts into unambiguous specifications. The challenge is to exploit the power and maturity of natural language, while overcoming its ambiguity and indirectness that results from the discrepancy between speech and thought.

As explained by Marian [108], to find the concepts that are used in thinking, it is not useful to model the physical processes and elements in the brain or to try to find exact categories of things in the real world. Finding modeling concepts is not a matter of investigating how the brain or reality works, *i.e.*, it is not about neuroscience or physics. Accordingly, the method should support making models of how people perceive, think, and communicate about the world, which underpins the research goal.

Sloman and Fernbach [149] point out that thought existed before language; some animals clearly have thought without language. They mention different reasons why thought evolved: to create a mental model of how the world is, to enable language, to solve problems, or for specific purposes, like making tools or finding a mate. These reasons have one thing in common: thought evolved for action. MuDForM can be aligned with natural language, but should not be hindered by it. MuDForM must enable specifying how the world is and what actions happen in it. The specifications of actions must be relatable to the specification of conditions and consequences.

Furthermore, the reasoning about problems, solutions, and their relationships, should be supported. The current definition of MuDForM only allows to specify problems, solutions, and their relations. It does not explicitly support the reasoning process for solving a problem. To support the reasoning process itself, method steps and guidelines must be defined, and maybe some new modeling concepts. Its investigation is future work.

6.5 Reflection on the Method Definition

The section reflects on the method definition. It is organized around seven themes that we identified in the reflection snippets of Appendix A.

6.5.1 Separation of Syntax and Semantics in Mind and Method

Section 3.4 introduced the model of cognitive aspects as the basis for defining modeling concepts. The root of the model is Concept. All other cognitive aspects, except Symbol, are derived from it. Concepts stand for units of meaning. Symbols stand for units of syntax. It appears that this separation between meaning and syntax comes naturally to the mind. But, syntax, and choice of words in particular, plays an important role in understanding specifications.

Used reflections:

- A.2.1: 'Concepts have meaning'
- A.2.2: 'There are no colorless green ideas that sleep furiously'
- A.2.3: 'Meaningless symbols refer to meaningful concepts'
- A.2.4: 'The mind has representations of concepts too'
- A.2.5: 'A concept can have multiple representations'
- A.2.6: 'Eliminate homonyms and synonyms, but not always'
- A.2.7: 'Involved experts decide about model element names, not the modeler'
- A.2.8: 'Resembling word forms in models'

John Searle [140] discusses the notion of concept according to philosopher Gottlob Frege: a concept is a function with a truth value, *i.e.*, it is true or false. To determine its truth value, a concept must have meaning, which is covered by the root of the cognitive aspects model, *i.e.*, the class Concept. If a MuDForM model is specified correctly, *i.e.*, it follows the metamodel and model criteria, then it always has a defined meaning. This is a significant difference with natural language, where grammatically correct sentences can be meaningless.

The meaning of a model (fragment) is not determined by its syntax, but by the used modeling concepts. Used names and other symbols do not add meaning to a model. The involved (domain) experts must recognize, and preferably choose, the names of model elements, because they must accept, and preferably adopt, the model. To reduce ambiguity, there are steps and guidelines to eliminate homonyms and synonyms, and to standardize logical constructs. The reduction of ambiguity could go further, which requires more research, and is a suggestion for future work.

Chapter 6. Cognition Perspective on MuDForM

The use of different word forms, *i.e.*, morphology, helps to understand what kind of word a word is. English uses word endings *e.g.*, -ness, -ability, -ism, and conjugations of verbs. Morphology helps writers choose a word, and readers understand a word. As MuDForM does not prescribe a syntax, model users cannot benefit from this feature. It is future work to investigate whether there are relations between modeling concepts, which can be exploited syntactically to increase model usability. For example, to understand that the class ‘Order’ is the objectification of the activity ‘to Order’, or that the function ‘Ordering’ centers around the execution of ‘to Order’, or that the state ‘Ordered’ means that an instance of ‘to Order’ is executed.

6.5.2 Supporting Analysis, Design, and their Context

A development team must understand and decide on many aspects of a system and its environment. A specification method must support capturing knowledge about a target domain, decisions about to-be-developed systems, and assumptions about other knowledge and decisions. Section 3.5.3 explains how these concepts are related. *Context models* serve as an explicit border between a software development team and their environment. Some principles of lexicography are applicable in the definition of MuDForM.

Used reflections:

- A.3.1: ‘Support analysis and design for multiple domains’
- A.3.2: ‘Analysis and design reflect lexical defining and real defining’
- A.3.3: ‘Domain models give meaning to other specifications’
- A.3.7: ‘Formal propositions vs. propositions about something’
- A.3.4: ‘Context models form a foundation for contracts with other parties’
- A.3.5: ‘Context models support subjectivity’
- A.3.6: ‘Separate what must happen from what makes it happen’

There is no restriction to what kinds of domains are involved in software development. Software development involves many people, who have their own distinct knowledge, expertise, and goals. MuDForM supports capturing knowledge and decisions from different people, about multiple domains, and supports integrating them by using model elements from one domain in the definition of model elements in another domain. This contributes to the support for multiple domains, for any domain, and for architecture, as expressed in method objectives O1, O2, and O8 respectively.

Stamper [154] discusses lexical defining vs. real defining. Lexical defining is the attempt to describe how a word is used and what it means in a particular setting. Real

6.5 Reflection on the Method Definition

defining is the attempt to declare the nature of something via defining it. Modeling a *Domain* is describing what can happen and can exist. It can be seen as an analysis activity and is comparable to lexical defining. Modeling a *Feature* is prescribing what must happen and must exist. It can be seen as a design activity, and is comparable to real defining. The distinction between the two is expressed in method objective O4. Furthermore, *Feature* models are fully expressed in terms of *Domain models*; *Domain models* are the dictionary for *Feature models*. As a consequence, the meaning of a *Feature model* is always clear.

An example: the domain model defines that you can pour beer from a bottle into a glass, and that the amount of beer that goes out of the bottle equals the amount that goes into the glass. This allows you to specify in a feature model how much beer you should pour and how fast. The context model defines for example what liquid amounts are. If you also want to express requirements about the amount of beer that may be spilled during pouring, then you made the wrong domain model. In the words of Searle [140], the meaning of the concept was there, but you did not have the language to express it. By capturing concepts in a model, you have the language to use their meaning.

Domain models are intended to describe something that exists. As such, they can be wrong, *i.e.*, they can be inconsistent with the world they describe. This is different for *Feature models*, because they prescribe what should be true in the world. A system can be an incorrect implementation of a *Feature model*, and as a result, the model is an incorrect description of what really happens. But in such case, we do not say the *Feature model* is wrong; we say the implementation is wrong.

Context modeling is neither analysis nor design. Context modeling is just declaring terms. *Context model elements* have a black box definition, *i.e.*, only their externally observable properties are specified, *e.g.*, *signatures*, *invariants*, *preconditions*, and *postconditions*, but their implementation is not. A *Context model* can serve as the foundation for a contract between the architect (and their organization) and the party that provides the implementation of the *Context model elements*.

Formalisms, like logic and mathematics, can be embedded in *Context models* via the Formalisms pattern (see Section 3.A.2). This means that one can use the declared formalisms in models. The metamodel of the MuDForM definition is itself also a formalism. For example, if you consider a *Step* in an *Object lifecycle*, you always know that such a *Step* refers to an *Activity role* of a defined *Domain activity*. This is not caused by the decisions of the modeler; the metamodel enforces it.

Chapter 6. Cognition Perspective on MuDForM

Context models can also be used to capture subjectivity. For example, if you have a criterion that you only want to buy ‘beautiful’ books, then you define a Boolean *Operation* that determines if a book is beautiful. You can declare an *Actor* that has the capability to execute that *Operation*. This may lead to the situation where the experienced meaning of the model is nondeterministic, because what books are bought depends on what the assigned *Actor* finds beautiful. The model itself is unambiguous, though.

An explicit step in the MuDForM modeling process is the transition from specifying what must happen in a function execution, to specifying what makes it happen, *i.e.*, assigning *Actors* to *Function steps*. The *Actors* are declared in a *Context model*. Their capabilities, specified in their *Behavior composition*, must match all the *Function steps* they are assigned to. There is currently a gap in the metamodel concerning the allocation of *Actors* to *Function steps*. Solving this gap is future work.

6.5.3 Model Engineering through Method Engineering

The methodical engineering of models, as opposed to ad hoc modeling or modeling through experience, is enabled by the maturity of the method ingredients. This section reflects on the application of lexicographical principles, deductively choosing model elements of the right granularity, and how guiding and tracing knowledge elicitation enhance model validity.

6.5.3.1 Apply Principles for Defining Words

There are different ways to define and learn words, which can also be applied in defining model elements.

Used reflections:

- A.4.1: ‘Support different types of defining a term’
- A.4.2: ‘Support extensional and intensional defining’
- A.4.3: ‘Support principles of vocabulary learning’

Stamper [154], Hofstadter and Sander [79], and Flanigan [59] mention different ways of defining words. We analyzed their applicability to our method. We conclude that analytical defining (aka intensional defining) is the main definition form for model elements. This way of defining consists of giving a model element a type, and relating it to other model elements, similar to analytical defining in dictionaries.

6.5 Reflection on the Method Definition

A MuDForM model is normalized, *i.e.*, it does not contain redundant model elements. Sometimes experts use a term that does not correspond with a normalized model element, but that is meaningful to them. Such a term must be allowed to increase model acceptance. This can be implemented via derived model elements, which are completely defined in terms of other model elements. For example, let ‘car’ be a normalized class with an attribute ‘number-of-wheels’, and suppose experts want to use the term ‘tricycle’. By defining that ‘tricycle’ is a ‘car’ with ‘number-of-wheels’ equal to ‘three’, such ‘tricycle’ can be used without violating the normalization rules. Extending the metamodel to support derived elements is future work.

All *Named elements* can have a list of *aliases*, which supports synonymous defining. Although real synonyms are scarce in a real-life language, in a system development context, people from different organizational units might use different terms for the same thing. They might also consider different properties of the same thing. In that case, the different terms are not exact synonyms, but it must be possible to state that their instances have the same identity. It should be possible to state that two model elements are the same, *i.e.*, to merge two model elements. Investigating how to realize this in MuDForM is future work.

As discussed in Section 6.5.1, MuDForM does not support morphology, but it could definitely help in learning; in making models, as well as in understanding models.

6.5.3.2 Deductive Modeling

The combination of metamodel, method steps and viewpoints, and guidelines help to deductively decide about what becomes a model element and what not.

Used reflections:

- A.4.4: ‘Avoid different ways of modeling the same meaning’
- A.4.5: ‘There should be guidelines for determining the right granularity of model elements’
- A.4.6: ‘Guidelines for deciding which concepts become a model element’
- A.4.7: ‘Guidelines for distinguishing model elements’

The MuDForM guidelines and criteria assure that models are normalized, *i.e.*, they do not have redundancies. For example, for defining family roles you only need the concepts of person, being a parent, and gender. You can specify the concepts father, mother, sibling, grandparent, cousin, etc., by expressing them in terms of parent relations and gender.

Chapter 6. Cognition Perspective on MuDForM

Furthermore, there are guidelines to prevent that knowledge can be modeled in different ways. They are gathered in Reflection A.4.7: ‘Guidelines for distinguishing model elements’ and their description can be found in the full method definition [33]. It is unclear how many of the possible choices for each method step can be directed by the current set of guidelines. We suspect there can be more, *e.g.*, for distinguishing *Attributes*. Finding out requires more investigation and is future work.

6.5.3.3 Guiding and Tracing Knowledge Elicitation Enhance Model Validity

Models represent situation-independent knowledge, which might be elicited directly from experts or other sources, or derived from situation specific information. Explicit method steps and viewpoints help to ease the traceability of analysis and modeling decisions, which enhances the validity of models.

Used reflections:

A.4.8: ‘Models represent situation-independent knowledge’

A.4.9: ‘Support knowledge elicitation from both semantic memory and episodic memory’

A.4.10: ‘Traceability helps to prevent falsities’

A.4.11: ‘Traceability and multiple viewpoints help to detect biases’

Goleman [69] discusses that people can have situation-independent thought. They can think and talk about situations that they are not in right now, or they can generalize knowledge from the situation that they are in. Pinker [125] states that compositional thoughts are often about individuals. The thought that a particular person eats an apple is different from the thought that people eat apples in general. People are capable of thinking things like ‘there is an X that Y’ without knowing such X, or things like ‘for all Xs: Y’ without knowing all Xs. Corballis [26] writes about the notions of episodic memory and semantic memory. Episodic memory is memory for events or episodes in our lives. For example, remembering that you ate an apple this morning. Semantic memory is memory for statements about the world. For example, remembering that Amsterdam is the capital of the Netherlands, or that people can eat apples.

Experts can be asked to share general knowledge about the targeted domain or system (from their semantic memory). They can make statements that are generic, as in ‘People can order a book’. This type of statement can be used directly as model input.

6.5 Reflection on the Method Definition

Experts can also be asked about events that happen(ed) in their domain (from their episodic memory). They can make instance-specific statements, as in ‘John ordered Lord of the Rings’. Such a statement needs to be generalized to become model input, *i.e.*, by asking what kind of thing ‘John’ is, what kind of thing ‘Lord of the Rings’ is, what kind of thing ‘orders’ is, and what other things can ‘order’ or be ‘ordered’. The guideline ‘Ask for examples when generic knowledge is hard to phrase’ appeals to the episodic memory of the involved expert.

A model can be seen as a representation of semantic memory. When a model is executed, it creates events, which can be logged and form a representation of episodic memory.

The knowledge that experts share, can be false and incomplete. It does not necessarily mean they are lying. They might simply be biased, or do not possess the right knowledge. All decisions during grammatical analysis and modeling can be logged in terms of the involved model elements, taken steps, and applied guidelines. This means that any inconsistency in the resulting model can be traced back to the input. The model viewpoints are defined on one metamodel, which also supports the detection of inconsistencies across views. Traceability, in accordance with method objective O7, reduces the chance that a model contains falsities.

Of course, it is still possible that falsities end up in the model. A bias in an input text that is transformed into a model, is probably also present in that model; bias in, bias out. Letting other experts validate a model, either directly via a review, or indirectly via a translation into natural language or another representation, prevents that a model is based on the biases of one person. It is future work to investigate if it is possible to find analysis and modeling guidelines that explicitly target a specific type of bias.

6.5.4 Match Reality with Useful Categories

Modeled categories must be consistent with the real-world concepts (objects) that they describe. Useful categories are not determined by observable properties of objects, but through the actions that their objects undergo, *i.e.*, useful categories are functional categories. Objects can belong to multiple categories throughout their existence. Whether multiple objects in the real world might correspond with one instance of a model element depends on their distinguishability. Changes in the real world may impact instances of a model, and sometimes also the model itself.

Chapter 6. Cognition Perspective on MuDForM

Used reflections:

- A.5.1: 'Categories enable inference of properties'
- A.5.2: 'Prevent mismatches between reality and model'
- A.5.3: 'Classifiers must represent functional categories'
- A.5.4: 'Support multiple functional categories for one instance'
- A.5.5: 'A model is a complete specification of the world it describes'
- A.5.6: 'It must be unambiguously determinable if an object belongs to a category'
- A.5.7: 'Different objects in the real world can correspond with one model instance'
- A.5.8: 'Objects differ if and only if they are distinguishable'
- A.5.9: 'Objects are not tied to one category'
- A.5.10: 'There are no exact category definitions in the real world'

In "How the mind works" [125], Steven Pinker concludes that 'the mind has to get something out of forming categories'. That something is inference. Obviously, we can not know everything about every object. But we can observe some of its properties, assign it to a category, and from the category predict properties that we did not observe. 'If Mopsy has long ears, he is a rabbit; if he is a rabbit, he should eat carrots, go hippity-hop, and breed like, well, a rabbit.'

The cognitive aspect **Category** represents the notion of category as meant by Pinker. **Category** is a subclass of **Concept**. It is implemented via the modeling concept **Classifier**, which can have *Generalization* relations to other **Classifiers**. **Classifiers** can have instances, *i.e.*, objects, and actions. If you know an instance of a **Classifier**, then it must have values for the specified properties of that **Classifier**. If it does not, then either the **Classifier**'s definition is incorrect, the classification is incorrect, or your knowledge about the instance is incorrect. Correct the model or the instance such that the consistency between them is ensured. This reasoning is captured in the modeling guideline *Ensure consistency between classifiers and known instances*.

Levitin [104] and Pinker [123] discuss the notion and importance of categorizing: perceptual and conceptual. We categorize things because they look, sound, smell alike. But also because they can fulfill some function in some context. The latter categories are called functional categories. MuDForM follows the notion of functional categories. Objects do not belong to the same category because they look (or sound, or smell) the same, but because you do the same things with them. Vice versa, objects belong to a different category when you do something different with them, *i.e.*, they undergo different types of actions. MuDForM has several guidelines driven by the notion of functional categorizing, which help to identify the right categories and their properties (see Reflection A.4.7: 'Guidelines for distinguishing model elements').

6.5 Reflection on the Method Definition

It is impossible to specify a concept in the real world completely. This is not the case for concepts that are an instance of a model. A model of a concept can be seen as a complete specification of that concept. For example, if a car is defined as a thing with wheels, then a bicycle and a shopping cart are also cars. This is correct if only the wheel-having property is important. If you want to know how many passengers it can transport, it is clearly an incomplete definition.

Searle [140] and Pinker [126] point to the difference between being countable and having identity. Two objects are identical if and only if they have the same properties. MuDForM follows this rule, called Leibniz's law. This means that two real-world objects without distinguishing properties correspond with one model instance. For example, two plastic cups that look exactly the same may correspond with one model instance that has the properties of that type of plastic cup. To distinguish the two cups requires them to have distinctive properties.

Several authors (Spinoza [101], Pinker [124], Hofstadter and Sander [79], Lakoff [99]) state that there is no precisely defined category for each object in the real world, as was presumed by classical philosophers. Categorizations of objects change, because both object properties and category properties can change. MuDForM currently covers the former situation. Objects belong to *Classes*, which can have *Generalization* relations. Objects may switch *Classes* throughout their life, as long as the those *Classes* are related via a *Generalization* relation.

The change of category definitions is covered partially. All instances have a model element as their type, and all model elements are *Classifiers*. But MuDForM does currently not have explicit operations for allocating an instance to a different model element or to change a model element. Those operations can be determined when the metamodel is extended with modeling activities, such as '*to Objectify an Activity into a Class*', or '*to Retype an Attribute with a Class*'. Currently, only the metamodel part for grammatical analysis contains activities (see the full method description [33]). Extending the metamodel with activities to deal with model changes is future work.

6.5.5 Time is Perceived through Events

Time is not an innate mental concept. Time is perceived through events. Events are ordered relative to each other. The method supports different options for ordering events.

Chapter 6. Cognition Perspective on MuDForM

Used reflections:

- A.6.1: 'Events are ordered in time, and have a duration'
- A.6.2: 'The past is determined by the events that have happened'
- A.6.3: 'Lifecycles enable stepping through time'

Pinker [126] observes that time is encoded in grammar in two ways. First, in tense, *i.e.*, past, present, and future. The other is called aspect, which is about the shape of the event. For example, instantaneous, as in 'swat a fly', open-ended, as in 'running around', a marking of completion, as in 'draw a circle'. A third one is called reference time, *i.e.*, an event to which is referred. For example, 'I ate before I showered', in which, 'I showered' is the reference time.

The cognitive aspects model has an *order* relation between *Events*. The *changes* relation between *Action* and *Object* has a timing aspect, because one can speak of the object state before and after an action. Regarding the event shape, a modeled *Activity* can be *atomic* or *non-atomic*. Actions that are an instance of an *atomic Activity* have by definition a timestamp. Instances of a *non-atomic Activity* always have a starting time, and if they are finished, they also have a stopping time.

Object lifecycles and *Function lifecycles*, allow, as a matter of speaking, to virtually travel through time. An object and a function execution are always at some point in their *lifecycle*. One can speak of the actions that happened before, the actions that happen now, and the actions that can or must happen. The Flow structure pattern, which is the foundation for *lifecycles* offers several possibilities to order *Steps*, *e.g.*, sequential or parallel, and optional or obligatory. A *lifecycle*, combined with a log of all actions that happened in the life of an object, enable reasoning about the next actions of that object.

6.5.6 Building Modeling Concepts from Natural Language

We discuss some common language constructs and generic word categories that we found in literature, and how they are used in the definition of MuDForM. We conclude that more natural language concepts can be used as the foundation of modeling concepts or in the transition from text to model.

Used reflections:

- A.7.1: 'Language is common, but there is no common language'
- A.7.2: 'Support for words that are common in all languages'
- A.7.3: 'Support part of speech concepts'

6.5 Reflection on the Method Definition

- A.7.4: ‘Support modeling verbs, and their temporal contour’
- A.7.5: ‘Support to be, to have, to do, and to go’
- A.7.6: ‘Support different meanings of to be’
- A.7.8: ‘Support different types of speech acts’
- A.7.7: ‘Support specification of location, force, agency, and causation’
- A.7.9: ‘Support Subject-Verb-Object sentences’
- A.7.10: ‘Compound names correlate with hierarchy between concepts’
- A.7.11: ‘Attributes and steps support indexicals’
- A.7.12: ‘Support word definitions and word usage’

Section 6.4 mentioned that language is common, and language awareness is innate. But according to Searle [139], linguists do not agree on universal language rules. To define modeling concepts, we searched concepts that are common in most languages. The search results can be divided in word categories, and language constructs.

Pinker [125] lists word categories that are common in all languages. The MuDForM support differs per category. For example, logical connectives, *e.g.*, ‘not’, ‘and’, and ‘same’, are directly supported, but motion and space are not.

We analyzed how the main parts of speech of the English language, *i.e.*, noun, pronoun, adjective, determiner, verb, adverb, preposition, conjunction, and interjection, are covered by MuDForM (see Reflection A.7.3: ‘Support part of speech concepts’). All relevant parts are currently addressed in the *Grammatical analysis* step. There are guidelines for choosing the right conversion, but we think that the *Text-to-model transformation* step can be extended with more guidelines.

We specifically analyzed the modeling of verbs, as hinted by Hofstadter and Sander [79], and in accordance with method objective O5. We analyzed how different types of verbs, and the common verbs to be (in different meanings), to do, to have, and to go, are supported (see Sections A.7.4, A.7.5, and A.7.6). We conclude that the different verbs and meanings are supported directly by modeling concepts, except for verbs relating to location and motion. We think that these should also be supported directly by modeling concepts, and we suggest how in Section A.7.7. The corresponding method definition part and its validation are future work.

In the lectures on “Philosophy of Language” [140], John Searle discusses different types of speech acts (assertive, directive, commissive, expressive, and declaration). We have analyzed the modeling possibilities for the different type of speech acts and conclude that all, except expressive, are addressed in some way. But it is not clear if they are covered correctly. More investigation is needed to see if it is useful to

Chapter 6. Cognition Perspective on MuDForM

support different types of speech acts with modeling concepts and guidelines. This is a suggestion for future work.

With respect to language constructs, we analyzed the support for the Subject-Verb-Object (SVO) structure of sentences, for compound names, and for indexicals². Each of them is supported explicitly by modeling concepts and guidelines.

Dennet [42] discusses the notion of words and tokens of a word, *i.e.*, the word and its meaning, and the use of the word in sentences. The word ‘word’ has four tokens in the previous sentence, and two in this one. This notion is present in many places in the metamodel, *e.g.*, the type of an *Attribute*, is a token of the *Class* that it refers to, and a *Function step* in a *Function lifecycle* refers to an *Activity*, hence, the *Function step* is a token of that *Activity*. The *Structure* pattern and its derivatives manifest the notion. In general, a *Reference* in a *Structure* is a token of the *Referred item* (see Section 3.A.3).

6.5.7 More Cognitive Aspects and Derived Modeling Concepts

Besides the cognitive aspects and corresponding modeling concepts that are discussed in the previous section, we identified some cognitive concepts that are not directly based on language constructs: the three principles of connections between ideas from David Hume, the generic concept of hierarchy between concepts, the mental concept of analogy making, and the difference between performative and constative speech acts.

Used reflection:

A.8.1: ‘Support possible connections between ideas’

Steven Pinker [127] refers to the 1748 book on human understanding of philosopher David Hume: ‘There appear to be only three principles of connections between ideas: similarity, contiguity in time or place, and cause or effect.’ Those connections are expressed as relations between sentences or partial sentences.

- Regarding Similarity: the cognitive aspect *Analogy* reflects the specification of similarity. Stating that something belongs to a *Category* is also an expression of similarity. What it means to state that two model fragments are similar, is not obvious. However, it could be useful to model, *e.g.*, to analyze if properties

²An indexical refers to something in its context, *e.g.*, “I”, “here”, and “now”.

6.5 Reflection on the Method Definition

of one model element also hold for another model element. For example, you can say that a car is similar to a horse. You can carry stuff with a horse. Can you also carry stuff with a car?

- Regarding Contiguity: the cognitive aspect Event has an order with itself. *Function lifecycles* and *Object lifecycles*, which are based on the *Flow structure* pattern, are used to specify contiguity in time. Contiguity of place is not supported at the moment, It is future work, as suggested in Section 6.5.6.
- Regarding Cause effect: an Event can cause another Event. An Agent may perform an Event, which is a kind of causation too. *Function lifecycles* specify what should happen when. Via the *behavior modality* of *Function steps*, the observation and generation of events can be specified, i.e., what event causes a *Function step* to execute, and what event is generated by a *Function step*. The effect of a *Function step* is defined by the *Activity* it refers to.

Used reflections:

A.8.2: 'Support the specification of hierarchy between concepts'

A.8.3: 'Support making abstractions'

Hofstadter and Sander [79] observe that concepts are hierarchical. We are building concepts from other concepts throughout our whole life. The cognitive aspect Property reflects hierarchy. The main mechanism in the metamodel for capturing hierarchy is the Structure pattern and its derivatives (see Section 3.A.3). It is applied to most relation types in the metamodel, such as *Behavior composition*, *Generalization*, *Attribute*, and *Object lifecycle*.

One form of hierarchy, that Hofstadter and Sander discuss, is making abstractions. 'People are constantly abstracting. This is not stupidity, but intelligence.' To make abstractions, MuDForM offers the concept of *Generalizations* between *Classifiers*. Furthermore, the modeling concept *Classifier* is itself an abstraction mechanism; a *Classifier* is an abstraction of a set of instances. For example, the *classifier* Apple can be seen as an abstraction of one or more specific apples. Generally speaking, making models is making abstractions.

Used reflections:

A.8.4: 'Support the specification of analogies'

A.8.5: 'Any model fragment can be seen as a pattern'

Chapter 6. Cognition Perspective on MuDForM

Hofstadter and Sander [79] dedicated a whole book to analogies being at the core of mental processes. Analogy is identified as one of the cognitive aspects, but MuDForM does not support it yet with specific modeling concepts. Of course, you can select a model fragment, copy it, rename the elements, and paste it in the target model, which reflects making an analogy. We think that guided analogy making would be a useful extension to MuDForM. How to support it properly requires more investigation and is future work.

Used reflection:

A.8.6: 'Distinguish performative from constative speech acts'

Searle [140] elaborates on the difference between performative speech acts and constative speech acts. A performative speech act is performing an action by saying something, *e.g.*, 'waiter, get me a beer please' is the act of ordering a beer. A constative speech act would be 'the waiter is getting me a beer'.

A software system can perform an action if it is a performative speech act, *e.g.*, order a beer via an app on your phone, which sends the order to a back-end system. Constatative speech acts can only be controlled by a software system via a call to some actor (operator or actuator) to perform the action, *e.g.*, the back-end system tells the waiter (or a robot) to deliver a beer. Constatative speech acts can also be observed by a software system via an interface on which an actor (user or sensor) indicates that an action happened, *e.g.*, the waiter confirms the delivery of a beer.

Modeling concepts to distinguish conceptual and physical activities could be useful. Investigating the support and benefits of different types of speech acts –there are more types, *e.g.*, see Section 6.5.6– is future work.

6.6 Discussion

This section discusses the reflections from Appendix A and their summaries in Sections 6.4 and 6.5. We first discuss the reflection on the method creation ingredients, and then the different types of reflections.

6.6.1 Cognition-based Method Definition

Section 6.4 and A.1 discuss how the studied literature relates to our vision, the main research goal, and method objectives, which are explained in Section 1.1. The primary

insight is that the need for unambiguity in software languages coincides with the inherent unambiguity of human thought. Building a metamodel from thought-based concepts does not raise a contradiction. Of course, those concepts must still be identified.

Seven of the eight method objectives are involved at least once in one of the reflections from Appendix A. This does not prove that the method objectives are correct or complete. It demonstrates that cognition and linguistics literature can be related to them, and that maybe more literature can be studied to fine-tune the method objectives. Objective O8, *i.e.*, the support for architecture, is an exception, because we only reflected on it through bookmarks in design and architecture literature.

Section 3.3 introduced the ingredients that make up the definition of MuDForM. We found content related to each of the method ingredients in Figure 6.1. We made reflections that implied modeling concepts, method steps, and guidelines. We found content related to viewpoints in general, but not to specific viewpoints. We also identified potential method ingredients that are not yet covered by MuDForM. Their addition is future work.

Section 3.4 introduced the notion of cognitive aspects, presented in Figures 3.2 and 3.3. We made reflections for each of the elements in Figure 3.2. We conclude that it makes sense to let all cognitive aspects, except *Symbol*, be a subclass of *Concept*. The reflections also discussed which modeling concepts were based on each cognitive aspect.

The cognitive aspects *Statement* and *Statement context* from Figure 3.3 are not mentioned in any reflection. Linguists and philosophers use other terms, such as *speech act*, *concept*, *symbol*, *idea*, *sentence*, *word*, and *phrase*. We think that *Statement* resembles the term ‘speech act’ from Searle [140]. We suggest to study more work about speech act theory, to determine if the concepts around the cognitive aspect *Statement* should be changed or extended.

Section 3.5.1 explains the construction patterns that we used in the definition of the MuDForM metamodel. Each of the three patterns occurred in at least one reflection. The *Named Elements* pattern is addressed in Section 6.5.1. The *Structure* pattern is mentioned at several places, mostly in Section 6.5.7. The *Formalisms* pattern is mentioned in Section 6.5.2. This does not mean that the patterns are optimal or complete. It shows that they correlate with cognitive or linguistic concepts.

6.6.2 Different Types of Reflections

During the analysis of the *bookmarks*, we discovered four different types of *findings* and *implications*:

- Some underpin a method design choice, A.7.11: ‘Attributes and steps support indexicals’ or A.7.3: ‘Support part of speech concepts’.
- Some challenge a method design choice, *e.g.*, A.5.10: ‘There are no exact category definitions in the real world’.
- Some suggest an improvement for the method, *e.g.*, A.8.6: ‘Distinguish performative from constative speech acts’.
- Some bookmarks just inspired a finding that made us consider a cognitive concept, *e.g.*, A.4.11: ‘Traceability and multiple viewpoints help to detect biases’, and A.3.2: ‘Analysis and design reflect lexical defining and real defining’.

Furthermore, we see a distinction between reflections that address the vision, main research goal, and method objectives, and reflections that target a specific method ingredient. The reflections of the former category are generic and more subjective. They can raise more method objectives and generic research questions. The reflections in the latter category are more specific and less subjective. They can serve as entry points for more specific research questions, *i.e.*, they could serve as motivation for investigating how a specific topic from cognitive science can serve as foundation for method ingredients.

6.7 Related work

To our knowledge, there is no literature that clarifies how a definition of modeling method can be based on cognitive concepts. However, we found two publications that suggest it, and provide a partial demonstration.

Siau states that despite the pivotal role of modeling methods, most method designs are just based on common sense and intuition of the method designers [144]. He proposes to use cognitive psychology as a reference discipline for information modeling and method engineering. He discusses the different types of memory and knowledge from the Adaptive Control of Thought theory from Anderson [6, 5], and gives an example of how this can impact and restrict a method design. He does not concretely

state how it can be used to define method ingredients. The MuDForM definition realizes explicit method objectives, and its modeling concepts are based on cognitive aspects, which demonstrates the use of knowledge from cognitive psychology as Siau suggests.

The role of cognitive aspects in the MuDForM definition resembles the idea of ontology-based metamodeling from Wand [166]. He distinguishes ontological elements and metamodel elements, and demonstrates and concludes that they can be related, but should not necessarily be the same. Wand does not state what the proper ontological elements are, nor does he define an approach for finding them. We think that the cognitive aspects in Section 3.4 can be seen as elements of an ontology of human cognition, which matches the idea of Wand.

The MuDForM definition is based on cognitive concepts from a large range of literature, cited in the reflections in Appendix A. We did not use explicit criteria for finding the right literature and the right cognitive aspects, and for deciding if they should be used in a method definition. This is a topic for future research. In this chapter, we wanted to sketch how cognition-based method creation can be implemented, and most of all, argue that it is possible.

6.8 Limitations

Following the reflection provided by Verdecchia *et al.* [162], we have identified three limitations to the validity of our study.

First, we do not make any claims regarding the completeness of the literature study, hence, we also do not do that about the completeness of the method ingredients from a cognition perspective. We think that we tackled some important authors in the field, but are aware that we missed works. For example, we think that “The Metaphors we live by” by Lakoff [100] and “How language began” by Everett [54] would be useful to study. There is simply too much literature, even for the nine years that this study took. As we are not cognitive scientists, linguists, or philosophers, more substantial involvement of people from those fields could help to mitigate this limitation.

The second limitation is that, naturally, it was not possible within the scope of this research to study all potentially important fields of science. A mitigation to this limitation, however, is that we studied about five times more books (about 150) than the ones cited by the reflections, and these have covered other scientific fields like

evolution theory, psychology, anthropology, evolutionary psychology, neuroscience, and even quantum physics.

The third limitation is that, potentially, we suffered from confirmation bias (*i.e.*, the tendency to notice evidence supporting one's beliefs and ignore evidence contrary to one's beliefs [132]) when we studied the literature, because we did this during the creation of MuDForM. This implies that we might have overlooked useful content from the literature, and thus, did not bookmark it. The mechanism that we used to mitigate this limitation is that we made much more bookmarks than the ones that finally ended up in Appendix A, *i.e.*, we over-bookmarked. When we started to revisit the bookmarks, we concluded that some of them were not useful, and did not put them in the intermediate analysis document [34]. More bookmarks were rejected during the analysis stage, as described in Section 6.3.

6.9 Conclusion and Future work

This chapter presents the results of a qualitative data analysis on literature in philosophy, cognitive science, and linguistics. The analysis led to 69 reflection snippets, described in Appendix A. The reflection snippets are summarized around eight themes, presented in Sections 6.4 and 6.5. The starting point was the following research question: *What are common elementary cognitive concepts that people use to think and communicate, and how can they be applied in engineering a specification method?*.

We have identified many cognitive concepts and showed how they can be used in the definition of MuDForM. We also conclude that cognitive science can be used to identify and drive improvements for MuDForM. Although the current version of MuDForM is quite stable and already practically applicable, there is always room for improvement. Below are some suggestions related to possible future improvements of MuDForM from a cognitive perspective. We distinguish should-haves, of which we concluded that they are an improvement, and could-haves, which require much deeper investigation to determine if they are feasible and would be an improvement.

The should-haves are:

- There should be explicit steps and guidelines for changing models, *e.g.*, merging two models. Furthermore, there could be default operations to specify what a model change entails for existing model instances.
- There should be modeling concepts to specify derived model elements.

6.9 Conclusion and Future work

- There should be method ingredients that directly support the specification of location and motion.
- There should be specific method ingredients that support analogy making.

The could-haves are:

- MuDForM supports the specification of the outcome of a reasoning process. It does not support the reasoning process itself. The support of reasoning would include explicit steps, viewpoints, and guidelines, and concepts like decision, alternative, trade-off between alternatives, and rationale behind a decision, which could form a useful addition to MuDForM.
- There are guidelines to eliminate homonyms and synonym, and guidelines to guide the definition of model elements, their granularity, their generalizations, and in which domain they belong. We think there could be more guidance possible to reduce ambiguity in models, and in the modeling process.
- MuDForM does currently not prescribe a syntax. But just like morphology in natural language helps to understand the type of a word, it could also be applied to better understand models. There could be guidance for defining a syntax that increases the understanding of MuDForM models.
- There could be explicit guidelines to prevent models being polluted by specific biases of involved experts. Different biases probably require different guidelines.
- It could be useful to have method ingredients that support the different speech acts. After all, making models is a speech act itself.

Appendix A contains more suggestions for future work. We did not use explicit guidelines or decision criteria for finding cognitive concepts in literature, which itself is a topic for future research. In this chapter, we wanted to demonstrate how cognition-based method engineering can be implemented. But most of all, that it is possible, and that it provides a useful addition to the engineering of methods and languages for software development. It demanded us to look over the fence beyond our own expertise, which was not always easy, but very interesting and educational. We invite you to look further, too.

7 Conclusion and Future Work

“The future’s uncertain and the end is
always near”

Roadhouse Blues, The Doors, 1970

Chapter 7. Conclusion and Future Work

In this concluding chapter, we (i) revisit the research questions that form the foundation of this thesis, based on the results from the previous chapters, (ii) apply the classification framework resulting from the systematic literature review in Chapter 2 to MudForM, (iii) summarize the suggested future work, and (iv) reflect on the main research goal.

Contents

7.1	Introduction	229
7.2	Research Questions Answered	229
7.2.1	RQ1: State-of-the-Art Specification Techniques	230
7.2.2	RQ2: Cognition-based Method Engineering	230
7.2.3	RQ3: From Text to Model	231
7.2.4	RQ4: Domain-based Feature Modeling	232
7.2.5	RQ5: Cognition Perspective on MuDForM	232
7.3	Classification of MuDForM	233
7.3.1	Application Scope	233
7.3.2	Method Engineering	235
7.3.3	MuDForM Specific	239
7.4	Future Work	241
7.4.1	Method Ingredients	241
7.4.2	Method Engineering	242
7.4.3	Putting MuDForM in Practice	243
7.5	Main Research Goal	244

7.1 Introduction

This chapter concludes this thesis. Section 7.2 revisits the research questions presented in Section 1.2, by summarizing the conclusions from the previous chapters. Section 7.3 reflects on MuDForM through the perspective of the classification framework resulting from the systematic literature review in Chapter 2. Section 7.4 summarizes the suggestions for future work. Finally, Section 7.5 reflects on the main research goal, presented in Section 1.1.2.

7.2 Research Questions Answered

This section discusses the answers to each of the research questions from Section 1.2.

Chapter 7. Conclusion and Future Work

7.2.1 RQ1: State-of-the-Art Specification Techniques

Chapter 2 answered the following research question:

RQ1: What are existing techniques to create domain-oriented specifications, and to what extent do they support the method objectives?

The chapter describes a systematic literature review (SLR) on domain-oriented specification techniques, based on the method objectives from Section 1.1.3. It provides an overview of the state-of-the-art in specification techniques to create domain models and domain-specific languages, *i.e.*, domain specifications, and their use in the creation of other specifications. It included in total 53 studies.

The SLR also identifies shortcomings in existing specification techniques. We found that no method covers all the method engineering aspects framed in our classification framework, which were derived from method objective O7. We think, of course, that the MuDForM definition should cover all those aspects.

Most of the included studies focus on creating domain specifications. None of them provides guidance for using them to write other specifications, *i.e.*, domain-based specifications. Similarly, no method for creating domain specifications has a method part that addresses how to deal with multiple domains. This implies that the application of the domain specification, and the integration of multiple domain specifications, has to be dealt with in a specific development context. The lack of guidance for using a domain specification to create other specifications has led us to formulating RQ4.

Some studies provide guidelines for the transformation of natural language text into a model. But many grammatical constructs are not supported. In general, a modeling language and natural language are not isomorphic, leading to differences in their expressiveness. The transformation of natural language text into a model, and vice versa, suffers from loss of semantics. *e.g.*, when entities or classes are used to represent the active verbs from a text. Active verbs represent behavior. But most studies neglect the specification of behavior, or integrate it poorly with the specification of state.

7.2.2 RQ2: Cognition-based Method Engineering

One of the shortcomings identified in the existing literature is that definitions of domain-oriented specification methods lack in their engineering maturity level.

Furthermore, to realize our vision, we want to create a method definition based on cognitive concepts. This led to the following research question:

RQ2: How to define a specification method that is explicitly cognition-based?

Chapter 3 answers this question by explaining the MuDForM definition. It clarifies how the modeling concepts are constructed from a few patterns, and how they are based on cognitive aspects. It also introduces the method steps, guidelines, and viewpoints of the complete method. The chapter shows how some of them are applied in a case study from industry.

A big part of the method definition is contained in appendices: construction patterns (3.A), all major modeling concepts (3.B), all method steps (3.C), all viewpoints (3.D), and the guidelines for feature modeling (5.A). The complete method definition is continuously under development. The latest version can be found online [33]. Besides the method ingredients mentioned above, it contains all guidelines, and the descriptions of all modeling concepts. We conclude that the definition is more mature and detailed than any of the methods that we included in our SLR from Chapter 2.

7.2.3 RQ3: From Text to Model

Another identified shortcoming in existing domain-oriented specification techniques is the lack of methodical support for using natural language in the specification process. This led to the following research question:

RQ3: What methodical support can be given for the conversion of text into elements of a domain-oriented model?

Chapter 4 answers this question by explaining the method step Grammatical analysis, including the involved analysis concepts. The explanation of the method step Text-to-model transformation clarifies how grammatical analysis results are used to create an initial model. Both steps are derived from the KISS method [96], which is the only approach from the SLR that has an explicit grammatical analysis step and corresponding concepts, and that distinguishes domain and feature.

The KISS method documentation does not provide a metamodel, fine-grained method steps, or guidelines. Chapter 4 and the full method definition [33] fill this gap, including the results from a case study. We observe that the defined analysis concepts and method steps are quite mature. However, the guidelines are far from complete, because we easily found new ones during the case study.

Chapter 7. Conclusion and Future Work

7.2.4 RQ4: Domain-based Feature Modeling

Another identified gap between the method objectives and the available domain-oriented specification techniques, is the lack of support for using domain models as the terminology for other specifications, and feature specifications in particular. This led to the following research question:

RQ4: What methodical support can be given for the specification of a feature, such that it is defined in terms of domain models?

Chapter 5 describes the MuDForM support for using a domain model as a language to define other models in general, and feature models in particular, and reports on an industrial case study in the automotive domain.

We conclude that the way in which MuDForM is defined, is also applicable to define feature modeling. Namely, we have defined modeling concepts, method steps, guidelines, and viewpoints, and integrated those method ingredients with the rest of the MuDForM definition. Furthermore, we conclude that it is possible to provide guidance for the creation of domain-based specifications. In doing so, we observe that the defined metamodel, and method steps are quite mature, as we did not detect relevant knowledge from the ISO26262 that we could not capture.

The results from the chapter fill an important gap in the state-of-the-art in domain-oriented methods, which to the best of our knowledge lacks in providing methodical support in the first place.

7.2.5 RQ5: Cognition Perspective on MuDForM

To fully address RQ2, we also needed input from cognitive science literature on what concepts should be implemented in the method definition. We decided to read books to look for knowledge that could play a role in the creation of MuDForM. We decided not to limit ourselves to just the definition of the method, but to search knowledge that covers the whole spectrum of method engineering. This led to the following research question:

RQ5: What are common elementary cognitive concepts that people use to think and communicate, and how can they be applied in engineering a specification method?

Chapter 6 answers RQ5, by reporting the results of a qualitative data analysis on literature in philosophy, cognitive science, and linguistics. The analysis led to 69 reflection snippets, described in Appendix A. The reflection snippets are summarized in the chapter around eight themes. The analysis identified many cognitive concepts, and showed how they can be used in the definition of MuDForM.

In this chapter, we wanted to showcase how cognition-based method engineering can be implemented. But most of all, that it is possible, and that it provides a useful addition to the engineering of methods and languages for software development. It demanded us to look over the fence, beyond our own expertise, which was not always easy, but very interesting and educational. We invite the interested reader to do the same.

7.3 Classification of MuDForM

Based on our vision, and our experience with domain modeling, architecture, and model driven development, we defined a set of method objectives for MuDForM, which are explained in Section 1.1.3. Chapter 2 defined a classification framework, based on those objectives, and used it to classify existing domain-oriented specification techniques. This section uses that framework to classify MuDForM.

The rest of this section is organized via the three categories of classification aspects: application scope (Section 7.3.1), method engineering (Section 7.3.2), and MuDForM-specific (Section 7.3.3). We have analyzed how MuDForM addresses each aspect via the results from Chapters 3, 4, and 5.

7.3.1 Application Scope

The Application scope category of the SLR's classification framework covers the objectives about being domain independent (O2), producing self-contained specifications (O3), and having a clear relation with architecture (O8). It considers the following aspects:

- **Domain dependence.** MuDForM can generally be applied to any domain or system that can be described. But, we do not recommend to always use MuDForM. For example, there are better languages for describing the blueprint of a house, a schema of an electrical circuit, or a knitting pattern. MuDForM

Chapter 7. Conclusion and Future Work

can be used to create the underlying model of such languages, though. So far, we did not encounter any conceptual system, like a software system, in which MuDForM was not applicable.

Via explicit context models, MuDForM supports the use of concepts that are defined outside the targeted domain and feature models. A context model contains elements that you assume to exist, and of which the internal properties are not required to be known. For example, in the specification of an embedded system, a context model can declare the functions of a hardware driver, which then can be invoked in the specification of the control software. Or, in the modeling of a book ordering domain, a context model declares a function to convert a postal code into a street name, which can be invoked when an address must be found.

By explicitly defining on which contexts a feature or domain depends, models have no implicit semantics. So, when a domain or feature are used, it is clear what other elements have to be incorporated to make the model self-contained, which supports the realization of Objective O3.

- Suitability for the **specification of quality**. Quality specifications are made in terms of quality-domain models, like in our case study about functional safety in the automotive domain (see Chapter 5). The only restriction to modeling a quality domain with MuDForM is that there is existing terminology. For example, MuDForM can be used to model quality attributes defined in the ISO/IEC25010 standard [86], by using the standard's description as input text.
- **The relation with architecture.** Architecture is supported in several ways. The expression of the metamodel in UML facilitates the implementability of a MuDForM model, because it is possible to unambiguously interpret all the UML concepts that are used in the metamodel (classes, attributes, associations, and specializations). Of course, this claim does not solve the possible incompleteness of a model. This aspect is covered via consistency rules, which are defined as postconditions of modeling steps.

Furthermore, it is possible to create a domain-specific language from a MuDForM domain model, and specify features as models in terms of such a language, which fits architectures based on model-driven development principles.

Applying MuDForM to quality attributes enables the specification of quality requirements, which are relevant for defining architecture, *e.g.*, the quality attribute scenarios introduced by Bass *et al.* [15]. More research is needed to investigate how this works when several qualities need to be combined in one

architecture. This would entail the specification of several qualities, integrated with the specification of a software design.

Let us consider an example of a software system that must meet privacy requirements, user-friendliness requirements, and availability requirements. This requires the presence of domain models of security, user-friendliness, and availability. These are used to specify the requirements for each quality attribute, which are captured in feature models. There must also be a domain model that can be used to specify the state and behavior of a software system, with concepts such as function execution, managed data, and operational state. There are several options to consider next. To denote a few: combine domain models in a new domain model, merge quality requirements that are about the same system domain objects, specifying consistency rules between the quality domains and make software that guards them at runtime. Which options work under which circumstances is the subject of the required research work.

The notions of model and view in MuDForM are the same as in the ISO/IEC42010 standard for architectural descriptions [85]. In ISO/IEC42010 terminology, a MuDForM Model is governed by one Model Kind, *i.e.*, the MuDForM metamodel. The postconditions of method steps that pertain to concepts in different viewpoints, can be seen ISO/IEC42010's correspondence rules. Extra correspondence rules can be defined when a MuDForM model is combined with other types of models, to form one consistent architecture description.

7.3.2 Method Engineering

The Method Engineering category of the SLR's classification framework is all about Objective O7. It considers the following aspects:

- **Support for specification consistency.** The viewpoints are defined on one metamodel, which supports the detection of inconsistencies across views. The detection of inconsistencies in feature models is supported by the rule that they are fully expressed in terms of domain models. Namely, the feature models may not make statements that conflict with the domain models. However, there is no guarantee that a model prevents all possible inconsistencies at instance level.
- **Support for traceability** from (intermediate) specifications back to the input. All decisions during grammatical analysis and modeling can be logged in terms

Chapter 7. Conclusion and Future Work

of the involved model elements, taken steps, and applied guidelines. This means that any inconsistency in the resulting model can be traced back to the input.

The metamodel of the discovery domain, introduced in Chapter 4, and fully explained in the complete method definition [33], contains explicit analysis activities, which can be used in logging decisions. The metamodel of the modeling domain does not contain activities. So, logging actions is limited to the implicit activities Create, Update, and Delete, just like with any other metamodel that only provides structure and state concepts. Adding modeling activities to the metamodel would be a significant improvement for the MuDForM definition.

- **Support for detecting incompleteness** in the targeted specification, and in the used input. Viewpoints overlap and are all based on the metamodel and all metamodel elements are present in at least one viewpoint, which supports the modeler to capture all relevant model aspects. Steps in the method flow iterate over viewpoints. The steps and guidelines comprise the elicitation of knowledge and decisions from involved (domain) experts, and can be used for logging and tracing decisions.

For example, when making an object lifecycle view of a domain class, one starts with referring to each of the related domain activities from the interaction view. During the modeling of the object lifecycle, new domain activities might be discovered, which then have to be added to the interaction view. The effect is that the two viewpoints and the corresponding method steps support the completion of a model.

- The method definition contains an **underlying model**, *e.g.*, metamodel, core model, or abstract syntax. The metamodel is fully defined in one UML model consisting of 65 classes. Additional consistency rules are defined as postconditions of modeling steps. Those rules are currently defined in natural language, which could be formalized, *e.g.*, in OCL [118].

The structure patterns (Section 3.A.3) are used throughout the whole metamodel, which brings several benefits. They provide expressive relations between metaclasses, dealing with aspects like multiplicities, nesting, ordering, and (conditional) selection between the elements in an N-ary relation. They provide uniformity in relations, which simplifies their representation and implementation. The disadvantage is that sometimes not all possible instances of a used pattern are allowed, and that an extra constraint is required to express the restriction.

- The method definition contains a **notation**, possibly used in different viewpoints. There are currently 15 viewpoints defined (see Table 3.4 in Appendix 3.D), but there is no syntax prescribed. All case studies in this thesis have used a UML-based notation (see Sections 3.6, 4.7, and 5.6). We also have a great deal of experience with using the KISS notation, and have experimented with a textual notation embedded in Gherkin scenarios [150], but they are not yet presented in academic publications.
- The method definition contains **method steps** that guide the modeling process. The method flow consists of 33 steps and sub-steps. Each step is currently described in terms of the modeling concepts. The definition of steps could be more uniform and precise if the metamodel itself is modeled as a MuDForM domain model, and contains modeling activities, as suggested under the traceability aspect above. This enables the specification of the method steps in terms of those modeling activities and modeling concepts, making the method step descriptions more precise.
- The method provides **guidance** for taking steps and making decisions. There are currently 125 guidelines defined, which are allocated to the method steps. The guidelines are written in terms of metamodel elements and, if necessary, complemented by commonly-used terminology from outside the metamodel. As such, we cannot ensure general understandability. As a mitigation, we use common terminology that 'anybody in the field' should be able to understand. To verify this, the method steps have been applied in several case studies, and the guidelines are reviewed by several practitioners.

Some guidelines could be defined as a separate step, as they define a separate modeling activity within a step. For example, the guideline *Define a function attribute for central objects* could be defined as a separate step *Define function attributes* with a guideline *Check for the central objects*. This refinement of the method steps could be useful, e.g., for building a modeling tool that actively leads the modeler through the method steps. The content of the method would still be the same, though. At this moment, we do not have hard criteria to decide whether something is a step or a guideline, because in the development of MuDForM we started identifying steps and gathering guidelines separately. Later, we assigned guidelines to steps and discovered that some guidelines could be seen as a step. Developing criteria for deciding whether some guidance is a step or a guideline is future work.

Chapter 7. Conclusion and Future Work

- **Formalness of specifications.** All modeling concepts have an unambiguous meaning. The semantics can be seen as formal, but there is no formal specification (yet) in terms of set theory, type theory, and process algebra.

Specification parts with informal semantics are also useful, *e.g.*, because it is not efficient to model everything formally. MuDForM allows hybrid specifications, *i.e.*, combining formal parts (interpretable by a machine) and informal parts (interpretable by a human), in several ways. First, some model elements have a name that has meaning outside the model. Second, it is possible to make incomplete specifications, *e.g.*, not specify the internal behavior of an activity or a function. Third, through the formalism pattern (Section 3.A.2), which can be used in context models, it is possible to connect model elements to specification elements that are not defined by the MuDForM metamodel. Those specification elements can be informal in meaning, because they only need to comply with the formalism pattern, *i.e.*, the external specification elements have a clear signature that can be invoked.

Besides the discussion of aspects from the SLR classification framework, we want to share some additional insights regarding the engineering of MuDForM.

At this moment, the method's ingredients (metamodel, steps, viewpoints, and guidelines) differ in maturity and completeness. The metamodel and viewpoints are stable, and the method steps change only sometimes. But guidelines are still frequently discovered, discussed, and specified more concisely.

Section 3.5.3 demonstrated how each MuDForM modeling concept is based on at least one cognitive aspect. This is not surprising, as none of the modeling concepts is completely new, and is borrowed from existing proven methods. What is new, is that we found literature that discusses the cognitive aspects (see Chapter 6), and that we explicitly embed them in the method definition.

We, the authors, are not experts in the cognitive sciences, but we think we made a solid start with covering the cognitive aspects that the literature has to offer. Further research is needed to identify more cognitive concepts that are relevant for specification methods, and to define more ways of using them in a method definition. We think that such research should involve cognitive scientists.

Although the evaluation design phase of the followed research method (see Section 3.2) did not include an explicit plan for reviewing the method definition, interested peers and people involved in case studies have reviewed the method definition.

They state it is difficult to understand how the modeling concepts must be used, and what they exactly mean from just the diagrams and their textual descriptions. The guidelines are easier to review, and reviewers have helped to improve their clarity. The most feedback came during face-to-face explanation of the method ingredients in modeling sessions, and not via reading the method documentation. There should be a more practical description of the method, including more example cases, to convey MuDForM to a wider audience.

7.3.3 MuDForM Specific

The MuDForM specific category of the SLR's classification framework covers the objectives that are specific to domain-oriented methods and MuDForM in particular. It considers the following aspects; each corresponds with a method objective:

- The method supports making specifications **in terms of domain specifications** (Objective O4). There are modeling concepts, method steps and guidelines for specifying feature models in terms of domain models, as is demonstrated in Chapter 5. Feature models are prescriptive, and domain models are descriptive. We have also experience with writing Gherkin scenarios [150] in terms of MuDForM models, to obtain uniform and unambiguous scenarios.
- The method supports the specification of **structural** (static) properties, **behavior** (dynamic) properties, and their relation (Objective O5). Concepts for state, *i.e.*, *Class*, *Domain Class*, and *Function* and concepts for change, *i.e.*, *Operation*, *Domain activity*, and *Function*, are all first class citizens, *i.e.*, model elements of these types can be declared independently of other elements. There are also concepts to specify the relations between classes and activities, and concepts to specify the details of those relations, *i.e.*, preconditions, postconditions, and invariants.

State composition is also possible, *i.e.*, context *Classes* may use other context *Classes*, and *Domain classes* use *Domain classes* and context *Classes*. Behavioral composition is also possible, *i.e.*, *Domain activities* use *Operations*, and *Functions* use other *Functions*, *Domain activities*, and *Operations*.

The metamodel class *Classifier*, and its specializations *Class* and *Activity* (see Table 3.3), enable an easy switch between the type of a model element, *i.e.*, between *Value type*, *Class*, *Domain class*, and *Function*, and between *Operation*, *Domain activity*, and *Function*. (*Function* is mentioned twice because it is a

Chapter 7. Conclusion and Future Work

combination of state and behavior.) The concepts for modeling the relations between *Activities* and *Classes* are not the same for the different *specification spaces*, because they require capturing different properties. There could be more uniformity than currently is the case, which would ease the transition of model elements to another specification space.

- Suitability for working with **multiple domains** (Objective O1). A MuDForM model consists of multiple specification spaces, which can be domains, feature, and contexts. Model elements in one space may use model elements from another space, which is a form of multiple domain support. We also foresee feature models and domain models that mainly consist of rules that specify how elements from different domains are related. The current metamodel does not contain modeling concepts for specifying such cross-domain rules. This requires more research involving case studies that require the integration of multiple domains.
- The support for using texts in **natural language as input** (Objective O6). The grammatical analysis step elicits and structures knowledge from the input text. We have extended the corresponding KISS method step with support for more grammatical constructs, and defined explicit guidelines. Chapter 4 explains this step in more detail.

It is possible to support more grammatical concepts, such as amounts, gerunds, comparatives, conjunctives, infinitives, imperatives, ordering in time, and modalities. Chapter 6 lists more suggestions for exploiting natural language input.

- The degree to which specifications are **translatable back into text in natural language**, and how well that text is still consistent with the original input text (also Objective O6). Although not fully demonstrated, every piece of model can be translated into natural language text. However, some viewpoints are more suitable for verbalization than others. For example, the interaction view of Figure 3.9 (cf. page 80) can easily be verbalized by making a sentence for each activity and its associated classes, as demonstrated in Section 3.6.2. That section also demonstrated that the verbalization of the object lifecycle from Figure 3.10 (cf. page 82) is less straightforward, and clearly requires more mental effort to understand.

7.4 Future Work

The results of our study unveil various paths for future work. We organized them in three categories: method ingredients, method engineering, and putting MuDForM into practice.

7.4.1 Method Ingredients

In general, more literature on domain analysis, ontologies, and different types of modeling can be studied to use method ingredients from existing methods.

More grammatical concepts or specific word categories found in linguistics literature can be supported with method ingredients. The same holds for the support of cognitive concepts from literature. We think it is best to define explicit criteria to decide which grammatical concepts or cognitive concepts should be supported, and which not.

We identified these topics for extra method ingredients:

- Specific modeling concepts to specify consistency rules across domains. Currently, only composition is provided as a mechanism to combine multiple domains in one model. We think that explicit modeling concepts for specifying how multiple independent domains fit together, without changing the specifications of those domains, would be beneficial for dealing with multiple domains. We want to apply MuDForM in case studies that require the integration of multiple domains, and quality domains in particular.
- Modeling concepts and steps for making analogies. It must be possible to take a model fragment and make an analogy with it, such that the result can be used in another model.
- As observed in cognition literature, people are mentally aware of the location and movements of things. These concepts should be supported with modeling concepts. For the specification of information systems, this might be less relevant. For the specification of software that controls and manages a physical domain, it makes sense.
- MuDForM supports the specification of the outcome of a reasoning process. It does not support the reasoning process itself. The support of reasoning would

Chapter 7. Conclusion and Future Work

include concepts like specification alternative, trade off between alternatives, and rationale behind decision.

- There should be explicit steps and guidelines for changing models, *e.g.*, merging two models. Furthermore, there could be default operations to specify what a model change entails for existing model instances.
- There should be modeling concepts to specify derived model elements.
- Support the specification of ‘how does it happen’. MuDForM supports the specification of what can happen, and what must happen. There are concepts to allocate actors to function steps, but these concepts are not mature, and we did not experiment with them in our case studies. To fully support the specification of a system design, it must be possible to state which activities are performed by which actors, and what the responsibilities of an actor are.

7.4.2 Method Engineering

We foresee the following topics for future improvements in the method engineering of MuDForM:

- Augment the metamodel with modeling activities, such that it is a MuDForM-compliant domain model itself, and the method steps can be fully expressed in terms of that domain model. In other words, the method flow will then be a feature model expressed in terms of the MuDForM domain. Namely, the metamodel specifies what can happen in the MuDForM domain, and the method flow specifies what should happen when making MuDForM models.
- Define criteria for deciding whether something should be a method step or a guideline. This could lead to refactoring MuDForM, as mentioned in Section 5.7.1.
- Consider the formal specification of the postconditions of the steps, and maybe the guidelines too, in OCL or another language.
- MuDForM does currently not prescribe a syntax. But just like morphology in natural language helps to understand the type of a word, it could be applied to better understand models. There could be guidance for defining a syntax that specifically increases the understanding of MuDForM models.

7.4.3 Putting MuDForM in Practice

Many hurdles have to be taken to let industry adopt a result from researchers from academia. There are many publications about its challenges and possible solutions, *e.g.*, [76, 66, 57, 167]. These can be used to define an approach for MuDForM.

For putting MuDForM into practice, we suggest at least the following topics to work on:

- Up till now, we have mostly used a UML modeling tool, *i.e.*, Enterprise Architect [152], for making MuDForM models. There should be a modeling tool that specifically supports the MuDForM metamodel and viewpoints. In particular, good support for the structure patterns and for consistency checking would improve the modeling experience. There should also be dedicated tool support for Grammatical analysis, to replace the set of MS Word templates that we used.
- There should be software that can execute models, or transform models into executable software. This is needed to fully realize the vision, because it would prevent that repetitive manual work is needed to make a system comply with the specification of knowledge and decisions of involved experts.
- We want to share all MuDForM related knowledge via an open platform, where suggestion for improvements and modeling issues can be discussed. The current GitHub project [33] is the starting point.
- There should be default domain models for quality attributes, *e.g.*, based on the text of the ISO/IEC25010 standard [86]. Standards for a specific quality attribute, like the for functional safety in the automotive domain [84], which we used in Chapter 5, can be used to create more elaborate domain models. After that, the domain models can be used to formalize requirements for those attributes, either from complying with such a standard, or for the development of a specific system.
- It would be useful for the software engineering community to have access to the various outputs of the application of MuDForM on an open-source system. In this way, interested researchers and practitioners can better understand the added value of the method, compare it with other methods, assess its applicability on their own systems, and verify the correctness of the application of the method, independently of the MuDForM developers.

Chapter 7. Conclusion and Future Work

- Finally, there should be practical support for practitioners, such as instruction videos on modeling topics, or a modeling course. The method definition from Chapter 3 is the starting point. We are currently working on training material for a domain modeling course.

7.5 Main Research Goal

After addressing RQ1-RQ5, the classification of MuDForM in the framework resulting from the SLR, and the summary of future work suggestions, we now turn to the main research question of Section 1.1.2:

What specification method enables all people involved in system development to unambiguously specify knowledge and decisions in a way that is close to how they think and communicate?

This thesis answers the question by explaining MuDForM, and reflecting on it. MuDForM is defined at a level of detail that goes further than those of other methods that we studied, as they do not guide the modeling process with so many well-defined steps, guidelines, and viewpoints. By being independent of any domain, by supporting the transformation of natural language into models, and by managing models in separate specification spaces, MuDForM supports capturing the knowledge of all involved people in a development process. The guidance does not end with the creation of domain models. Making system specifications based on domain models is also methodically supported.

We have seen that all modeling concepts and several guidelines are underpinned by cognitive concepts. As can be expected, not all concepts from cognition literature are covered by a MuDForM method ingredient. This requires more study of those concepts, and clear criteria about what cognitive concepts should be supported by the method.

When putting the method into practice, the used notation influences the ease of model creation and understanding. The current method definition does not prescribe a notation. More research into the use of different syntaxes for different modeling contexts is needed, to realize the vision that software development is connected to human thinking and communication as much as possible.

A Cognition-based Reflection Snippets

Contents

A.1 Research Background: Vision, Research Goals, and Method Objectives	248
A.1.1 Use unambiguous models to convey knowledge	248
A.1.2 People care about their domain. Computers do not know what they are doing	248
A.1.3 Specifications must mean something to people	249
A.1.4 Method steps and viewpoints focus the modeling process . .	250
A.1.5 Modeling is not an extra activity	251
A.1.6 Focus on domain-oriented methods	251
A.1.7 Exploit the lexical hypothesis	252
A.1.8 Language awareness is innate	252
A.1.9 Cognition is more than language	252
A.1.10 Modeling concepts should be based on Mentalese	253
A.1.11 Focus on mental concepts; not on the brain or reality . . .	254
A.1.12 On the evolution of thought	255
A.2 Separation of Syntax and Semantics In Mind and Method	257
A.2.1 Concepts have meaning	257
A.2.2 There are no colorless green ideas that sleep furiously . . .	257
A.2.3 Meaningless symbols refer to meaningful concepts	258
A.2.4 The mind has representations of concepts too	258
A.2.5 A concept can have multiple representations	259

Appendix A. Cognition-based Reflection Snippets

A.2.6 Eliminate homonyms and synonyms, but not always	259
A.2.7 Involved experts decide about model element names, not the modeler	261
A.2.8 Resembling word forms in models	261
A.3 Supporting Analysis, Design, and their Context	262
A.3.1 Support analysis and design for multiple domains	262
A.3.2 Analysis and design reflect lexical defining and real defining	263
A.3.3 Domain models give meaning to other specifications	264
A.3.4 Context models form a foundation for contracts with other parties	265
A.3.5 Context models support subjectivity	265
A.3.6 Separate what must happen from what makes it happen	266
A.3.7 Formal propositions vs. propositions about something	267
A.4 Model Engineering through Method Engineering	267
A.4.1 Support different types of defining a term	267
A.4.2 Support extensional and intensional defining	269
A.4.3 Support principles of vocabulary learning	270
A.4.4 Avoid different ways of modeling the same meaning	271
A.4.5 There should be guidelines for determining the right granu- larity of model elements	272
A.4.6 Guidelines for deciding which concepts become a model element	273
A.4.7 Guidelines for distinguishing model elements	273
A.4.8 Models represent situation-independent knowledge	274
A.4.9 Support knowledge elicitation from both semantic memory and episodic memory	275
A.4.10 Traceability helps to prevent falsities	276
A.4.11 Traceability and multiple viewpoints help to detect biases	276
A.5 Match Reality with Useful Categories	277
A.5.1 Categories enable inference of properties	277
A.5.2 Prevent mismatches between reality and model	278
A.5.3 Classifiers must represent functional categories	279
A.5.4 Support multiple functional categories for one instance	279
A.5.5 A model is a complete specification of the world it describes	280
A.5.6 It must be unambiguously determinable if an object belongs to a category	281
A.5.7 Different objects in the real world can correspond with one model instance	282

A.5.8 Objects differ if and only if they are distinguishable	282
A.5.9 Objects are not tied to one category	283
A.5.10 There are no exact category definitions in the real world . . .	285
A.6 Time is Perceived through Events	286
A.6.1 Events are ordered in time, and have a duration	286
A.6.2 The past is determined by the events that have happened . .	286
A.6.3 Lifecycles enable stepping through time	287
A.7 Building Modeling Concepts from Natural Language	288
A.7.1 Language is common, but there is no common language . .	288
A.7.2 Support for words that are common in all languages . . .	288
A.7.3 Support part of speech concepts	289
A.7.4 Support modeling verbs, and their temporal contour . . .	290
A.7.5 Support to be, to have, to do, and to go	291
A.7.6 Support different meanings of to be	292
A.7.7 Support specification of location, force, agency, and causation	293
A.7.8 Support different types of speech acts	294
A.7.9 Support Subject-Verb-Object sentences	295
A.7.10 Compound names correlate with hierarchy between concepts	295
A.7.11 Attributes and steps support indexicals	296
A.7.12 Support word definitions and word usage	297
A.8 More Cognitive Aspects and their Derived Modeling Concepts . .	297
A.8.1 Support possible connections between ideas	297
A.8.2 Support the specification of hierarchy between concepts .	299
A.8.3 Support making abstractions	300
A.8.4 Support the specification of analogies	300
A.8.5 Any model fragment can be seen as a pattern	301
A.8.6 Distinguish performative from constative speech acts . . .	302

This appendix contains the reflection snippets that are the foundation for Chapter 6. Each section here corresponds with a section in that chapter. Many of the snippets suggest more research or extensions for MuDForM, which are indicated with the term ‘future work’. When a bookmark and its findings led to multiple reflections, we did not copy the bookmark and its finding, but named the bookmark and referred to it.

Appendix A. Cognition-based Reflection Snippets

A.1 Research Background: Vision, Research Goals, and Method Objectives

A.1.1 Use unambiguous models to convey knowledge

Bookmark Hofstadter 1: In “Surfaces and Essences, Analogy as the Fuel and Fire of Thinking” [79], Hofstadter and Sander discuss that learning new concepts happens through words and through experiences. The first is seeing a word and then learning its meaning. People learn concepts mostly through experiences; some of those concepts get their own (compound) word.

Finding: Models are codifications of knowledge (from experiences), *i.e.*, they make tacit knowledge tangible. People might learn directly from a model by studying it. It is also possible to transform a model into, for example, software that can be executed. Such an execution can be experienced by someone, *i.e.*, an end user of the software. This means that people can learn from the model through experiencing it, which can also be used to validate a model. Namely, when the statements from an expert are codified in a model and that model is turned into software, then the expert can validate the codification of his knowledge by experiencing the execution of the software.

Impact: Models can be used to convey knowledge. just like text. Unambiguous models can be turned into software, and, when executed, used to convey knowledge through experience too. When transferring knowledge through unambiguous models, human learning and software development coincide.

A.1.2 People care about their domain. Computers do not know what they are doing

Bookmark: In “The Language Instinct: How the Mind Creates Language” [123], Steven Pinker discusses some consequences of the Turing machine. Such a machine can do things without understanding what it does. It can do any calculation without knowing what it stands for. For example, the syllogism ‘all men are mortal’, ‘Socrates is a man’, thus, ‘Socrates is mortal’, can be interpreted by a machine. The machine does not need to know what ‘Socrates’, ‘man’, and ‘mortal’ stand for. They can be replaced by anything when it learns how to interpret ‘all’, ‘are’. ‘is a’, and ‘is’.

A.1 Research Background: Vision, Research Goals, and Method Objectives

Bookmark: In “How the mind works” [125], Pinker reflects again on the computational theory of mind (from Alan Turing). The brain is storing and processing information. Information itself is emotionless, it is just symbols. Peoples’ minds give it meaning.

Bookmark: In “From Bacteria to Bach and Back: The Evolution of Minds” [42], Daniel C. Dennett sees the Turing machine as a mindless machine that can do computations without knowing that it is doing computations. A programmable digital computer is a universal Turing machine.

Finding: As mentioned in Section 1.1.1, source code often contains terms of its application domain, which makes that code less reusable than it potentially is. In order to put only necessary knowledge in specifications, they should be defined as abstract as possible, *i.e.*, they should not be defined in terms of their application domain, but in terms of their own abstract domain, which is often a design aspect or a quality attribute domain. After all, the computer, *i.e.*, a Turing machine, does not know the semantics of what it is doing.

Impact: Software, and formal specifications in general, are like the syllogism. The computer does the calculation, but has no clue what it means outside the computer. The Turing principle and the observation that source code often contains terms from the application domain of the software, are a reason to make specifications at an abstraction level that is not based on a computer-based programming language, but on cognition-based concepts. There is no reason why human knowledge must be captured in computer-based language.

A.1.3 Specifications must mean something to people

Bookmark Harari 1: In “Homo Deus: A Brief History of Tomorrow” [75], Yuval Noah Harari explains that Homo sapiens gives meaning to the world. Most animals live in a dual layered world. They perceive the physical world with trees, rivers, rocks, and other animals, and they also perceive an inner world with things like fears, beliefs, and desires. Humans live in a triple layered reality, because they also perceive made up things like money, gods, and corporations.

Finding: It should be possible to cover things in all three layers. But, only if people can communicate about it, which is also the limitation. The method only needs to cover things that people can communicate about, which is also the case with any other specification language or natural language. Specifications themselves can also

Appendix A. Cognition-based Reflection Snippets

be seen as elements at the third layer. They do not really exist in the same way as elements of the first layer. They only exist as a representation in the first layer; their meaning is in the third layer.

Impact: The method only needs to be based on concepts for the third layer. So, no concepts for representing trees (first layer), but only concepts for representing thoughts and communication about trees. And, no concepts for representing emotions (second layer), but only concepts for thinking and communicating about emotions. The essence of a specification is what it means to people, not how it is represented. This is complementary to the observation that they mean nothing to a Turing machine, as mentioned in *Reflection ‘People care about their domain. Computers do not know what they are doing’ (cf. page 248)*. This underpins the vision to work on specifications, *i.e.*, models.

A.1.4 Method steps and viewpoints focus the modeling process

Bookmark: In “The Organized Mind” [104], Daniel Levitin talks about active sorting, *i.e.*, separating things you need right now from things that you do not need right now.

Finding: When you are modeling, there is a lot of information. Which information is relevant at a certain moment, depends on the modeling step and viewpoint that you are working on. Some form of active sorting might offer support.

Impact: Through the steps defined by the method flow, you find which information you need to focus on at a certain moment during modeling. For example, when you model an object lifecycle, you consider different model aspects, leading to different questions to the involved domain expert, than when you are defining attributes. When you are modeling an object lifecycle of a particular domain class, you know you have to create a step for all the domain activities that the domain class has involvements in. A tool could help you to show you only those activities.

This kind of tool support can be generalized, because mostly when you are modeling, you are busy with one root element and already related model elements are possible candidates to create more detailed model content with. For example, when making a function lifecycle, you tend to only make function steps of domain activities related to the domain classes that are types of function attributes. This kind of active sorting can be automated, which improves the modeling experience.

A.1 Research Background: Vision, Research Goals, and Method Objectives

A.1.5 Modeling is not an extra activity

In “Everything All at Once” [116], Bill Nye states that if you want to be successful (with a design), get in the project in the beginning.

Finding: Modeling should start from the beginning.

Impact: It does not make sense to start a modeling activity when a project already made designs in another way, or when there is already source code developed before the specification is made that the source code should comply with. Modeling should start whenever relevant (domain) knowledge is entering the project, either through some document, or through some expert. When domain models already exist and specifications, *e.g.*, requirements, must be made, then the domain models should form the terminology for those specifications.

In other words, modeling is not something you do in parallel with things you already do. It should replace (some of) them, which is captured in the guideline ‘Modeling should replace or complement other activities’.

A.1.6 Focus on domain-oriented methods

Bookmark Dennett 1: The philosopher Daniel C. Dennett states in “From Bacteria to Bach and Back: The Evolution of Minds” [42] that programming is a creative activity. Software has bugs and debugging can not be automated. Only syntax errors can be prevented. Dennett compares it to how evolution gets rid of bugs. Reproduction creates variations, and natural selection will take care of the flawed variations.

Finding: This metaphor misses an important point: getting rid of bugs through (real-life) testing, *i.e.*, ‘natural selection’, is not the only possible mechanism, because programming, unlike biological reproduction, is not a mindless activity. Programming is a cognitive activity, which can be done systematically, *i.e.*, guided by a method, to prevent bugs from happening. Dennett does not consider that. Of course, bugs can still occur. But there is more possible than testing to eliminate them.

Impact: Modeling, like programming, does not need to be a purely creative activity, where the quality of a model depends on the involved experts’ capability to formulate their knowledge. It can be guided by a method that supports the extraction of knowledge and capturing it in a way that suits the involved expert.

Appendix A. Cognition-based Reflection Snippets

A program in a programming language is often the place where all decisions for a software system are encoded. But a programming language is not optimal for capturing most decisions in a development project. Different system aspects like security, logging, application initiation, and the application domain functionality, and development aspects like planability, time-to-market, and manageability, all have their own concepts, which could be supported by their own domain specific language (DSL). A DSL starts with a domain model. This underpins our choice to investigate a method for domain-oriented modeling.

A.1.7 Exploit the lexical hypothesis

Bookmark Levitin 1: In “The Organized Mind” [104], Daniel Levitin discusses the lexical hypothesis, *i.e.*, the most important things humans need to talk about eventually become encoded in language.

Finding: This does not only hold for peoples’ social life, but also for their work life.

Impact: It is clearly a motivation for using natural language as input for modeling, and to support the different ways of how those important things can be encoded.

A.1.8 Language awareness is innate

Bookmark: Mariano Sigman observes in the book “The Secret Life of the Mind: how our brain thinks, feels, and decides” [145] that a just born baby is already aware of spoken language, especially of their native language. *i.e.*, the language of their mother. He concludes that language awareness is innate.

Finding: Innate language awareness is a confirmation of its use in the method.

Impact: Natural language is taken into account when defining modeling concepts, which is reflected in Objective O6.

A.1.9 Cognition is more than language

Bookmark: In “The Language Instinct: How the Mind Creates Language” [123], Steven Pinker talks about universal concepts in language. An investigation about commonalities in languages did not yield generic mental concepts that are present

A.1 Research Background: Vision, Research Goals, and Method Objectives

in all languages, but identified words and grammatical constructions that occur in almost all languages. The only common thing in almost all languages is that they use sentences that have a subject, verb, and object. The mostly identified commonalities are about the presence of words.

Bookmark Pinker 1: Pinker also discusses Mentalese: the internal language of the mind. This is the notion that there is an internal language, which is free from syntax. For example, 'the man throws a ball', and 'the ball is thrown by the man', have a different syntax, but our mind knows they mean the same. Moreover, the mind can think without having words. People that did not learn language can still think and solve problems; just like some animals. It is a misconception that the native language of a person determines what that person can think. This misconception is called genuine linguistic determinism. Of course, language influences thoughts. Otherwise, language would be useless.

Bookmark Pinker 2: In "The Stuff of Thought: Language as a Window into Human Nature" [126], Pinker elaborates on genuine linguistic determinism, aka the Whorfian hypothesis, *i.e.*, that a person's language determines what the person can think. He states this is clearly wrong as mentioned in the previous bookmark.

Finding: We conclude that it is not trivial to find cognition-based or mental-based commonalities in language. But, it does not mean that common mental concepts do not exist in the human mind. They are simply not obvious.

Moreover, besides concepts from natural language, also concepts from other cognitive, not directly language-related mental processes, should be supported by our method. In perspective, if the Whorfian hypothesis would be right, then the meta-model of our modeling language could be limited to the grammar of natural language.

Impact: This is a reason not only to look at natural language for finding proper modeling concepts, but also at cognitive science beyond linguistics. A specification language should be based on how people think and not just on what they talk about. This underpins the research goal.

A.1.10 Modeling concepts should be based on Mentalese

Bookmark Pinker 3: In "How the mind works" [125] elaborates on Mentalese, *i.e.*, the language of the mind, and why English is not Mentalese. He gives the example of people knowing relations between family members, and that monkeys also know

Appendix A. Cognition-based Reflection Snippets

the family relations in their group. But monkeys do not talk about it. For humans, a family will all its relations can be verbalized in many ways. For example, John is Mary's father, means the same as Mary is John's daughter. Moreover, there are also derived concepts like being siblings, or more specific being a sister or brother. English is not Mentalese. English sentences are cluttered with articles, prepositions, suffixes, and other grammatical boilerplate. They are needed to help get information from the mouth to the ear. But they are not needed inside one's mind.

Bookmark Pinker 4: Pinker also discusses why thoughts cannot just be represented by pictures. Pictures can be ambiguous, but thoughts, by definition, cannot be ambiguous. If a mental picture is used to represent a thought, it needs to be accompanied by a caption, a set of instructions for how to interpret the picture; what to pay attention to and what to ignore. The captions cannot themselves be pictures, or we would be back where we started. Representations can be ambiguous; also textual representations.

Finding: The method's metamodel should be based on the language of thought, *i.e.*, Mentalese. Specifications should be unambiguous, just like thoughts.

Impact: This underpins looking for modeling concepts beyond natural language, *i.e.*, concepts based on cognition. We need to investigate Mentalese.

A.1.11 Focus on mental concepts; not on the brain or reality

Bookmark Marian 1: In “The Power of Language: How the Codes We Use to Think, Speak, and Live Transform Our Minds” [108], Viorica Marian states that the notion that precise categories exist outside our interpretation of reality, may be an illusion perpetuated by language. Regardless if there are real categories that exist in the world, the linguistic and mental categories that we create, matter. They have consequences for many areas. Marian also states that mental categories do not exist inside the brain as physically separate categories.

Finding: Mental categories are the ones that should be supported in the method, because they are the basis for communication and thinking. The method should not try to attempt to cover the modeling of reality, nor should it be based on the physiological structure of the human brain. A model should reflect how people see perceive reality. This is consistent with the Reflection A.1.3: ‘Specifications must mean something to people’.

A.1 Research Background: Vision, Research Goals, and Method Objectives

Impact: How the world actually is, or how the brain is constructed, is not important for a specification. Finding modeling concepts is, thus, not a matter of investigating how the brain or reality works, *i.e.*, it is not about neuroscience or physics. The method should support making models of how people perceive, think, and communicate about the world. This underpins the research goal.

A.1.12 On the evolution of thought

Bookmark Sloman 1: In “The Knowledge Illusion, Why We Never Think Alone” [149], Steven Sloman and Philip Fernbach explain that thought could have evolved for several functions:

1. To represent the world; to construct a model in our head that correspond in critical ways to the way the world is.
2. To make language possible
3. For problem-solving or decision-making.
4. For specific purposes, such as building tools or showing off to potential mates.

There is a thing in common for all these purposes:

- 5 Thought is for action; thinking evolved as an extension of the ability to act effectively. It evolved to make us better at doing what is necessary to achieve our goals. Thought is helpful for predicting the effect of actions, and to reason about actions that we did. Evolutionary seen, action came before thought.

Finding: From the perspective of creating a specification method, we conclude the following for each mentioned point above:

1. There should be a way to specify the world as it is.
2. There should be a way to translate thoughts into specifications via natural language.
3. There should be a way to specify decisions, problems, solutions, and their relation.

Appendix A. Cognition-based Reflection Snippets

4. We think that the method should not be tight to a specific purpose, but it should be possible to make a model that serves such purpose.
5. There should be a way to specify actions, and how reasoning leads to actions. It should be possible to observe the effect of actions, and use such observations in specifications.

Impact: Objective O6 reflects 2), as it underpins the link to natural language and the search for cognitive concepts, *i.e.*, concepts that make up thought. Objective O4 reflects 1) and 3). Objective O5, *i.e.*, support the specification of state, change, and their relation, addresses directly what is mentioned under 5). Regarding:

1. MuDForM offers the concepts of Context and Domain to capture ‘thoughts’ about how the world is.
2. MuDForM models are often transformations from natural language and can be translated back into natural language too.
3. MuDForM offers the concept of Features (with Functions) to reflect which actions should be taken under which circumstances. Multiple domains can be distinguished to separate problem from solution, and one can combine domains to specify how a solution solves a problem.
4. It does not make sense to put in modeling concepts for specific purposes. But of course, MuDForM can be used to model the domain of those specific purposes.
5. MuDForM offers autonomous concepts for specifying an action, *i.e.*, *Activity*, and for its *Behavioral composition* and *Lifecycle*. Each *Activity* can have pre-conditions, and each activity occurrence can have guards. *Flow structure types* (sequence, selection, parallel), and *multiplicities* (one, zero or more, at most one, at least one) offer a way to take different actions based on the constraints that are set on the flow structures.

For *Domain classes*, *Functions*, and *Domain activities*, it is possible to specify what the next actions can/must be in their *lifecycle*. By keeping track of where an instance is in its *lifecycle*, it is possible to decide what next action to perform.

A.2 Separation of Syntax and Semantics In Mind and Method

A.2.1 Concepts have meaning

Bookmark: John Searle discusses in his lectures on “Philosophy of Language” [140] the definition of concept according to German philosopher Gottlob Frege¹: a concept is a function with a truth value, and at least one argument. The statement ‘the king of France is bald’ is not a concept because it is neither true nor false. It simply does not mean anything. To be clear: baldness exists, but France does not have a king. That is why the statement is not a concept.

Finding: How does this relate to the MuDForM notion of concept?

Impact: In the definition of MuDForM, we state that a Concept is something that the human mind can give meaning to. This is a different formulation, but it covers the definition of Frege.

A.2.2 There are no colorless green ideas that sleep furiously

Bookmark: In “Syntactic Structures” [22], Noam Chomsky presents the -now famous- sentence ‘Colorless, green ideas sleep furiously’ as an example of a sentence that is grammatically well-formed, but semantically nonsensical. It is possible in natural language to write sentences that are grammatically correct, but that do not mean anything.

Impact: The metamodel enforces that feature models are expressed in terms of domain models and context models, and that domain models are expressed in terms of context models. For example, if a domain models declares that ‘someone can place an order for a book’, then a feature model has to abide this structure when using the activity ‘to place’ in a function, *i.e.*, a book and an order must be involved (and nothing else). The consequence is that feature and domain models can not be meaningless in themselves, which realizes Objective O3 (see Section 1.1.3). Natural language and MuDForM (just as many other specification and programming languages) differ at this point.

The intention is that context models declare concepts that have a meaning outside

¹https://en.wikipedia.org/wiki/Gottlob_Frege

Appendix A. Cognition-based Reflection Snippets

the model. Of course, when this is not the case, then the domain and feature models that use such concepts, can also be meaningless. However, this is different from the example of Chomsky, because there, despite the meaningful words, the English grammar allows the sentence to be meaningless. With MuDFoM the lack of meaning can only be caused by declaring meaningless context model elements. Regarding the example in *Reflection ‘Concepts have meaning’* (*cf. page 257*), with MuDFoM it is simply not allowed to use the phrase ‘the king of France’.

A.2.3 Meaningless symbols refer to meaningful concepts

See *Bookmark Hofstadter 1* (*cf. page 248*).

Impact: The cognitive aspect **Symbol** is the entry point for learning Concepts through words.

See *Bookmark Hofstadter 2* (*cf. page 300*).

Impact: From a semantics point of view, all the meaning of a model is in the meta-model, *i.e.*, in the meaning of the modeling concepts. Model elements are represented by a recognizable symbol, mostly a combination of an icon for the modeling concept and a name to point to a concept outside the model. Because the semantics of the model is defined in the metal, you can replace a name by any other name, and the model will mean the same.

See *Bookmark Levitin 1* (*cf. page 252*).

Impact: The cognitive aspect **Concept** reflects the notion of a concept and **Symbol** reflects the word that is used to represent its codification.

A.2.4 The mind has representations of concepts too

Bookmark: In “I Am a Strange Loop” [78], Douglas R. Hofstadter claims that people have a mental symbol for each concept that they know. Such a symbol is triggered whenever someone thinks about the concept. Concepts are anything that people can give meaning to; not only things represented with a noun or verb. For example, relations like ‘before’, ‘after’, and ‘but’ are also concepts, and also ‘the Eiffel Tower’, or ‘being late’ can be seen concepts.

A.2 Separation of Syntax and Semantics In Mind and Method

Finding: To be cognition-based, MuDForM should also support the representation of concepts with symbols. This is of course nothing new for a specification method; it is fundamental.

Impact: Concept is the root of all cognitive aspects, as represented in the diagram of Figure 3.2. Symbol1 is used in the MuDForM definition to refer to a concept. It could be a mental reference, but also a reference in other contexts, like an icon or word in a model, or another type of specification. The separation of cognitive aspects Concept and Symbol1 resembles the notion of a concept represented with a mental symbol as described by Hofstadter.

A.2.5 A concept can have multiple representations

See *Bookmark Pinker 1* (cf. page 253) about Mentalese not being the same as spoken or written language.

Finding: A specification requires a syntax. But the metamodel and its semantics should be independent of language syntax and grammar. Furthermore, it should be possible to define different syntaxes (notations and viewpoints) for specifications.

Impact: MuDForM is defined in a metamodel, which determines the semantics of MuDForM compliant models. The metamodel elements are based on cognitive aspects, which are intended to resemble the notion of Mentalese. It is possible to define multiple syntaxes and viewpoints. For example, a function lifecycle can be represented as a nested enumerated bullet list, or as a regular expression notation, or as a UML activity diagram. This is why the viewpoints are completely defined in terms of metamodel elements. They do not add extra semantics to a model.

A.2.6 Eliminate homonyms and synonyms, but not always

Bookmark Pinker 5: In “The Language Instinct: How the Mind Creates Language” [123], Steven Pinker states there are more homonyms than synonyms in natural languages. Homonyms are plentiful. Real Synonyms are rare.

Bookmark: In “Building a Better Vocabulary” [59] Kevin Flanigan states that a natural language does not have many real synonyms. Although two words might be seen as synonym, or a dictionary list them as synonyms, there is mostly a slight difference in the meaning, or in the context that a word is used.

Appendix A. Cognition-based Reflection Snippets

Finding: This might not apply to the terms in specifications, because there you are not always interested in all the differences between the meanings of two terms. Homonyms and real synonyms should be avoided in a specification, because they lead to unclear communication. Two terms might be synonyms, but used in different context by different people. Two terms might also refer to different properties of the same identity, which should be possible to model.

Different departments in an organization may use a different term for the same thing, because they consider different aspects of it. For example, employee and employment have the same identity, but refer to different properties. Vice versa, different departments might use the same word, but mean something different, for example the marketing department uses ‘book’ to refer to the book title, *i.e.*, the specification of the book, and the delivery department uses ‘book’ to refer to the instance that is shipped. Both must be supported. There should be guidelines for when to eliminate homonyms and synonyms and when to keep them.

Impact: To reduce ambiguity, eliminating homonyms and synonyms is an explicit step in the grammatical analysis. The step should be generalized to a step ‘Reduce ambiguity of terms’, because there are more issues of ambiguity that can be removed. There is already a guideline that says to unify terminology that expresses logical constructs, like ‘for all’, and ‘there is’. To investigate this is future work.

Via the pattern ‘Named elements’ (see Section 3.A.1), model elements can have a list of aliases, which are treated as real synonyms. Furthermore, separates classes and generalizations can be used to model different properties of the same instance. For example, ‘Employee’ and ‘Employment contract’ can be classes that each have a generalization to the class ‘Employment’.

Homonyms are allowed in the sense that a named element is declared in a Specification space, *i.e.*, a name space, and across several Specification spaces, the same name might be used for different model elements as in the ‘book’ example above.

The MuDForM metamodel itself is normalized and does not have ambiguities in its meaning. Hence, MuDForM models are unambiguous. To benefit from this property, also the used syntax must be unambiguous. Currently, MuDForM does not prescribe such a syntax.

A.2 Separation of Syntax and Semantics In Mind and Method

A.2.7 Involved experts decide about model element names, not the modeler

Bookmark: In “The Organized Mind” [104], Daniel Levitin observes that humans do not have a term (word) for every concept that they can have in their mind, *e.g.*, the category ‘people that you need to notify when you are going into the hospital for three weeks’ does not have a term for it in English. This does not mean that such a concept is not understood by people.

Finding: There must be guidance for naming model elements.

Impact: Sometimes a model element has to be introduced that does not have an existing term in the domain, *e.g.*, because normalization forces you to introduce a model element which experts do not perceive as a distinct autonomous identity (yet). In such a case, a name must be invented. Such a name should be supported, preferably chosen, by the involved (domain) experts. Also, sometimes a collection of objects must get a name, because they meet some constraint, as in the example ‘people that you need to notify when you are going into the hospital for three weeks’. Also then, the involved experts should support the chosen name.

The above is reflected in the guidelines ‘The term must be recognized by the expert’ in the step ‘Eliminate homonyms and synonyms’, and ‘Involved experts choose names’ in the step ‘Model engineering’. The name of a model element is often a composition of words, and not a single word, *e.g.*, the attribute ‘book title’, or the function ‘order books online’. Furthermore, there are the guidelines ‘Use nouns for class names’, ‘Use verbs for activity names’, and ‘Let function names reflect the duration of the function’. They are clearly less important than the acceptance of the involved experts, though.

A.2.8 Resembling word forms in models

Bookmark: In “The Language Instinct: How the Mind Creates Language” [123], Steven Pinker observes the words of a language are treated distinctly from their meaning. The sound of a word does not determine the meaning of the word, *e.g.*, a word pronounced somewhere between pet and bet does not mean something in between. But, via positioning words in a sentence, and via conjugations, meanings can be altered. For example, questions have a different word order, or the past tense has “-ed”.

Appendix A. Cognition-based Reflection Snippets

Finding: The MuDFoM definition separates meaning and syntax. The syntax should not mean anything. But, syntax could be used to make it easier to understand the meaning of a modeling concept.

Impact: The current MuDFoM definition does not prescribe any syntax. But, a specific implementation could benefit from the principles that natural language uses to alter the meaning of a stem term. For example, an activity ‘to order a product’, the class ‘order’, which is the result of the activity, a state ‘ordered’, which can be attribute value of an instance of the class ‘product’ or derived from a ‘order step’ in the object lifecycle of ‘product’, and the actor ‘orderer’, who is the one who placed the ‘order’. The present participle ‘ordering’ can be used to indicate that an order activity is active, as in ‘john is ordering Lord of the Rings’, or it could be a gerund of the order activity, as in ‘the ordering of Lord of the Rings by John’. There could be guidelines for resembling name correlation in the metamodel, which investigation is future work.

A.3 Supporting Analysis, Design, and their Context

A.3.1 Support analysis and design for multiple domains

Bookmark Goldberger 1: In “Why Architecture Matters” [68], Paul Goldberger states that it is the task of the architect to design the best building within the problem of the customer. Architecture must go beyond aesthetics, *i.e.*, beyond pure form. Architecture is not about itself, but about everything else. It is the job of the architect to design the best building. But society needs to provide the problems.

Bookmark: In “How to Think Like a Philosopher” [11], Julian Baggini compares thinking alone with thinking in groups. Thinking for ourselves is like turning your brain into a Swiss army knife. Your brain has to have all the required knowledge and thinking skills. Thinking together with others is like a Toolbox with dedicated tools. A Swiss army knife is compact and can do a lot. But the tool elements can not be used at the same time and are restricted by their integration in the knife. A toolbox is bigger ad possibly not aligned, but can potentially do more and do it better.

Bookmark: In “Everything All at Once” [116], Bill Nye talks about good design. If the design is no good, the product will never be good. The design, including requirements, is the bottom of the system pyramid. Everything that is created after that, builds upon it. It is difficult to correct design flaws, no matter how hard you try. However, if the design is great, it is still possible to screw up later on.

A.3 Supporting Analysis, Design, and their Context

Finding: Today's software development involves many people, who have their own distinct expertise and requirements. Developing a software system is mostly a team effort, and not the task of one or a few engineers. A development team must understand many aspects of the system and its environment. The head of an engineer is not the optimal tool to integrate knowledge from multiple stakeholders about diverse domains. Making one feature and domain model vs. making multiple feature and domain models is analogous to the army knife vs. the toolbox. There should be support for all involved (domain) experts, and their knowledge and decisions should be specified separately and integrated explicitly. MuDForM should support understanding domains, via domain models, and support creating good designs, via feature models. It should also minimize the chances that the design can be screwed up later in the development activity.

Impact: The specification of knowledge and decisions about many aspects of a system supports architecting, which is expressed by method objective O8 (see Section 1.1.3). Furthermore, it underpins that the method should be usable for any domain (method objective O2), and it should be usable to make specifications that involve multiple domains (method objective O1).

MuDForM supports capturing knowledge and decisions from different people, about multiple domains, and supports integrating them by using model elements from one specification space in the definition of model elements in another specification space. However, there are currently no explicit concepts in the metamodel to specify transformations and consistency rules between domains and features, as mentioned in Objective O1. This is future work.

Modeling domains is an analysis activity. Modeling features is a design activity. They can be applied to functionality of the system, but also to other, nonfunctional, domains, like security or reliability. MuDForM models are unambiguous, hence, can be transformed into a system in an automated manner, which reduces the chances that the system contains errors that were introduced after the design. This is in general the promise of model driven development. The MuDForM modeling concepts, method steps, viewpoints, and guidelines, contribute to get a good specification, *i.e.*, a good analysis and design, which contributes to Objective O8.

A.3.2 Analysis and design reflect lexical defining and real defining

Bookmark: In "Word by Word, The Secret Life of Dictionaries" [154], Kory Stamper discusses real defining vs. lexical defining. Lexical defining is the attempt to describe

Appendix A. Cognition-based Reflection Snippets

how a word is used and what it means in a particular setting. Real defining is the attempt to declare the nature of something via defining it. Lexicographers do not undertake real defining, as they are only concerned with words that have an existing meaning.

Finding: Modelers of existing knowledge do lexical defining, which is called analysis. Modelers of 'new' knowledge, *i.e.*, design, do real defining.

Impact: The method should support capturing the existing meaning of a term, leading to a descriptive statement, as well as declaring the meaning of a term, leading to a prescriptive statement. This is expressed through method objective O4.

A coarse grained transformation is that in a domain model we capture what something existing means, *i.e.*, a domain model is a lexical definition. However, if you are defining a new domain through a domain model, then it is a case of real defining.

In a feature model, we state what a term, *e.g.*, the name of a function, entails for the world that uses that term. As such, specifying a feature model is a case of real defining. This does not exclude that a feature is based on existing knowledge.

A context model is a form of lexical defining when you use it to refer to existing concepts, *e.g.*, physical quantities, or mathematical formalisms. When you use a context model to declare what you assume to exist outside the domains and features of interest, then it can be seen as real defining.

A.3.3 Domain models give meaning to other specifications

Bookmark: In the lectures on "Philosophy of Language" [140], John Searle disagrees with the indeterminacy of translation hypothesis of W.V. Quine [131]. Quine says that meaning depends on a reference 'coordinate system', *e.g.*, language, while Searle says that meaning exists without language. Searle says that meaning exists without the words for expressing that meaning. The argument of Quine is that you can only scientifically check the meaning through the use of words.

Finding: This discussion is related to the discussion about thoughts being independent of language, and to the notion of Mentalese (see *Bookmark Pinker 1* (*cf. page 253*) and *Bookmark Pinker 2* (*cf. page 253*)). Thoughts can exist without the explicit syntax that comes with a language, like in the family relations example. Primates that live in groups are aware of family relations, without having words to express those relations.

Impact: Specifications are not subject to the discussion about meaning and language. Namely, everything meaningful and relevant should be modeled, and thus fits Quine's perspective. Furthermore, MuDFoRM domain models describe what can happen and what can exist. If a created domain model meets this criterion, then the situation where you want to express something that cannot be expressed with the domain model, cannot occur. If it does not meet the criterion, then the domain model is incorrect and should be adapted. This mechanism also addresses *Bookmark Dennett 1* (*cf. page 251*), because the obligatory use of domain models in the specification of features reduces inconsistency and prevents that specifications are ambiguous.

An example: the domain model states that you can pour beer from a bottle into a glass, and that the amount of beer that goes out of the bottle is equals the amount that goes into the glass. This is fine. But if you also want to express (in a feature model) that much beer may be spilled during the pouring, and that the amount added to the glass can be less than the amount that goes out of the bottle, then you simply made the wrong domain model. In the words of Quine and Searle [140], the meaning of the concept was there, but you did not have the language to express it.

A.3.4 Context models form a foundation for contracts with other parties

See *Bookmark Goldberger 1* (*cf. page 262*).

Impact: An architect uses domain models for analyzing things that must be managed and controlled by the system they are designing, and feature models to define a design. Context models are used for things that already exist. Context model elements have a black box definition, *i.e.*, their externally observable signature, invariants, preconditions, and postconditions, are specified, but their implementation is not. Other parties may realize the implementation of the context model elements. A context model can serve as the foundation for a contract between the architect (and their organization) and the party that implements the context model elements.

A.3.5 Context models support subjectivity

Bookmark: In “Fake Physics: Spoofs, Hoaxes, and Fictitious Science” [110], Andrew May posits that right and wrong are relative concepts. For example, the earth is not flat, nor is it perfectly spherical. But to call it flat is more wrong than to call it round.

Appendix A. Cognition-based Reflection Snippets

Finding: Although MuDForM specifications are meant to be unambiguous, there should be a possibility to capture subjective, imprecise, and possibly ambiguous knowledge.

Impact: Capturing ambiguity and subjectivity is offered via context models. For example, if you have a criterion that you only want to buy 'beautiful' books, then you define a Boolean operation that determines if a book is beautiful. You can declare an Actor that has the capability to execute that Operation, which may lead to the situation the experienced meaning of the model is ambiguous. Namely, what books are bought depends on what the assigned actor finds beautiful. The model itself is still unambiguous, though.

A.3.6 Separate what must happen from what makes it happen

Bookmark: In "The Meme Machine" [16], Susan Blackmore discusses the notion of algorithms. Many things can be seen as an algorithm. Algorithms are mindless. Algorithms are substrate neutral, which means that they can run on many things, from a computer to a human brain.

Finding: The method should support the mindlessness of algorithms, *i.e.*, having a specification of some executable logic, without specifying how it is implemented, *i.e.*, without stating what actor runs it.

See *Bookmark Pinker 9* (*cf. page 295*).

Impact: Verbs that represent actions can have an Actor that performs the action. The Actor is often omitted in the domain model, because stating what (or who) performs an action is mostly a design choice and is not in the scope of a domain model. When you develop a system, you first model what can happen, then what should happen, and after that, you want to decide what makes it happen.

MuDForM has the modeling concept Behavior modality to express if an action must or can be executed, activated (called), or just observed (received). Context models contain the declarations of Actors, including the specification of their capabilities.

MuDForM supports making specifications without declaring the actor that does what the specification prescribes. There is an explicit transition in the modeling process, from specifying what must happen in a feature, to specifying how does it happen, *i.e.*, assigning actors to function steps. The actors are declared in a context model and

A.4 Model Engineering through Method Engineering

their capabilities must match with the function steps they are assigned to. There is currently a gap in the metamodel to fully capture the allocation of actors to function steps. Solving the gap requires investigation in future work.

A.3.7 Formal propositions vs. propositions about something

Bookmark: In “Language, Truth, and Logic” [10], A.J. Ayer builds upon the works of Bertrand Russell and Wittgenstein. Propositions are divided into two classes: those which concern relations of ideas, and those which concern matters of fact. The former class comprises the propositions of logic and mathematics, which are necessarily true. On the other hand, propositions about empirical matters of fact are hold to be hypotheses, which can be probable but never certain.

Finding: How is this division of propositions present in models?

Impact: Of course, a domain model is an example of the second class of propositions; you try to do your best to make a correct model, but you are never certain. It is different for a feature model. A feature model defines something that does not exist without the model. Namely, it defines what should be true in the modeled ‘reality’. A system can implement a feature incorrectly, and as such the model is an incorrect proposition of what really happens.

The first class of propositions is present in two different ways. First, formalisms, like logic and mathematics, can be embedded in context models via the formalisms pattern. This means that one can use the formalisms in models. Second, the metamodel of the MuDForM definition is itself also a formalism. For example, if you consider a step in an Object lifecycle, you always know that such a step refers to a role of a modeled Domain activity. This is not caused by the decisions of the modeler; the MuDForM metamodel forces it.

A.4 Model Engineering through Method Engineering

A.4.1 Support different types of defining a term

Bookmark Stamper 1: In “Word by Word, The Secret Life of Dictionaries” [154], Kory Stamper distinguishes different types of defining a term:

Appendix A. Cognition-based Reflection Snippets

- Ostensive definition: Point to the thing. Only usable when you can point to the real world is available in some form, *e.g.*, to a picture².
- Analytical definition (aka intensional definition): Start with the genus (aka headword), *i.e.*, the core of the word. Add descriptors for differentia that clarify how the term differs from the genus. The descriptors provide criteria for determining if something can be denoted with the term.
- Synonymous defining: Listing synonyms of the term. This happens a lot, but it is not a good way to give an exact definition. (See also *Bookmark Pinker 5 (cf. page 259)*.)
- Enumerative definition: special type of extensional definition that gives an explicit and exhaustive list of all the objects that fall under the concept or term in question. Enumerative definitions are only possible for finite sets and only practical for relatively small sets³.

Finding: We need to consider how the different types of defining are applicable in modeling.

Impact:

- Regarding ostensive definitions: the guidelines ‘A class description positions the class in its context’, ‘Describe a domain class’, and ‘Describe a domain activity’ state that an example instance should be given. This reflects ostensive defining.
- Regarding intensional defining: Model elements always have a modeling concept as a genus, *e.g.*, Domain class, or Attribute. The descriptors are the model element properties that are captured via the structures of the model element. For example, car is a Class with an Attribute ‘number-of-wheels’ and the ‘number-of-wheels’ must be higher than ‘two’. Furthermore, a Class can have a Generalization to a parent Class, in which the parent Classes has the role of genus, *e.g.*, a ‘car’ is a ‘vehicle’. A Class can have more than one Generalization, *e.g.*, a ‘car’ is a ‘vehicle’, a ‘status symbol’, and a ‘taxable object’. Depending on the domain, a different parent class can be seen as the genus of the car. Investigating if it makes sense to distinguish the genus property of a generalization is future work.

²https://en.wikipedia.org/wiki/Ostensive_definition

³https://en.wikipedia.org/wiki/Enumerative_definition

A.4 Model Engineering through Method Engineering

Semantics are fully defined through the metamodel. Hence, names of model elements do not express a meaning. Context model elements are an exception. They are the starting point for the definition of other elements. Their names are assumed to have meaning outside the model, *i.e.*, their internal properties are defined outside the model. Only the properties that domain model elements and feature model elements interface with, are defined in the context model.

Future work: It should be possible to change the model element of an instance, which is like changing the genus of a term. For example, if you decide to distinguish cars with four or more wheels from tricycles, and you have a car with three wheels, then you should be able to change the type of that car to tricycle.

Future work: It should be possible to specify derived elements by taking a model element as a genus and defining a constraint to determine which instances of the model element belong to the class of the derived element. For example, a ‘tricycle’ is a car”, which ‘number-of-wheels’ equals ‘three’. This is useful when tricycles do not have any other relevant distinguishing properties, but experts use the term in their communication.

- Regarding synonymous defining: each model element can have a set of aliases.
Future work: It should be possible to state that the instances of two model elements are the same, *i.e.*, to merge two model elements. Although real synonyms are scarce in a real-life-natural-language context, in a system development context, people from different organizational units might use different terms for the same thing. They often consider different properties of the same thing. In that case, the different terms are not exact synonyms, but it must be possible to state that (some of) their instances have the same identity. To support this is future work.
- Regarding enumerative defining: It is possible to specify classes that are enumerations of possible instances, *i.e.*, possible values.

A.4.2 Support extensional and intensional defining

Bookmark: Hofstadter and Sander discuss in “Surfaces and Essences, Analogy as the Fuel and Fire of Thinking” [79] the difference between the intensional and extensional defining of categories. Intensional defining is stating the properties an entity should have to belong to a category. Extensional defining is stating that an entity is a member of a category. (This is also observed in *Bookmark Stamper 1* (*cf. page 267*)).

Appendix A. Cognition-based Reflection Snippets

Finding: Both extensional and intensional defining should be possible.

Impact: Extension is done by simply creating a new instance of a class or by the instantiating domain activity of a domain class. Stating that a domain object belongs to a domain class is not directly possible. It always goes via an explicit domain action, in which objects from a context enter a domain. For example, in a domain activity ‘to register a person as a new customer’, person data is used to create a new customer object that has attribute values that correspond with the person data. In this case, the person properties already existed outside the domain, and the register action stands for the check whether the right properties are present to be categorized as a customer, and if so, assigning those properties to a new customer object.

A.4.3 Support principles of vocabulary learning

Bookmark: In “Building a Better Vocabulary” [59], Kevin Flanigan explains five principles of vocabulary learning:

1. Definition: a word should have a definition that explains what it means.
2. Context: a word is learned by seeing how it ‘behaves’ in a context, *i.e.*, how it is used in sentences, paragraphs, and complete texts.
3. Connections: a word is related to concepts that you already know.
4. Morphology: the form of a word, *i.e.*, its parts and the conjugations can help to learn what kind of word it is.
5. Semantic chunking: cut up a word into parts that have a distinct meaning.

Finding: The principles could help when a (domain) model should be understood by someone through studying it.

Impact: Regarding:

1. Definitions: There are guidelines for writing descriptions of model elements.
2. Context: A definition should also include an example instance of a model element, especially when it is a specification element, *i.e.*, an element that is directly contained in the specification space, such as a class, operation, actor,

A.4 Model Engineering through Method Engineering

class relation, domain class, domain activity, domain class relation, function, or an attribute of a feature.

3. Connections: All model elements are connected to other model elements. An exception can be context elements, because they are used as references to concepts that are not fully specified. But still, those elements are in the context model, because they are needed for the definition of feature elements or domain elements.
4. Morphology: This is currently not directly supported in the MuDForM definition, because it does not prescribe a syntax. Although, it could be useful for learning MuDForM. An example is the UML syntax convention to use the same symbol for an instance as for its type, and then underline the name of the instance so you can see it is an instance. Through the structure patterns in MuDForM, it can be seen if a model element is a container item, a structure, an element, a reference, or a referred item, which is helping to decide how it can be related to other model elements.
5. Semantic chunking: this is only supported implicitly, because it can be applied in the name of elements in structures. There are currently no guidelines for it. Although, the names in a domain model should come from the domain. It is not the modeler who should choose names of model elements, but the involved domain experts. For specifying functions it is different, because they often have an invented name.

In general, it is future work to investigate what guidelines can be made to reflect the principles of vocabulary learning.

A.4.4 Avoid different ways of modeling the same meaning

See *Bookmark Pinker 4* (cf. page 254) and *Bookmark Pinker 3* (cf. page 253).

Finding: The metamodel should be based on the concepts of Mentalese as close as possible. If some meaning can be expressed in several ways with such concepts, then there should be guidelines to choose the best way. For example, a lifecycle has two types of steps: A and B. Suppose that an A should always be followed by a B, then a guideline should lead you to model this in the lifecycle model with a sequence between A and B, instead of two parallel paths, one with step A and one with step B, and a precondition on path B that it should be preceded by path A.

Appendix A. Cognition-based Reflection Snippets

Impact: The MuDForM guidelines and criteria assure that models are normalized, *i.e.*, they do not have redundancies. For example, for defining (biological) family roles you only need the concepts of person, being a parent, and gender. You can specify the concepts father, mother, sibling, grandparent, cousin, etc., by expressing them in terms of parent relations and gender.

The guideline ‘Only use constraints when other modeling concepts do not suffice’ holds for the model engineering step. The special version of that guideline ‘Only use invariants if other concepts cannot cover it’ pertains to domain engineering. These are compliant with the example of paths A and B above. It is unclear how much of the modeling choices for each step are covered by the current set of guidelines. Finding out requires more investigation and is future work.

A.4.5 There should be guidelines for determining the right granularity of model elements

Bookmark Stamper 2: In “Word by Word, The Secret Life of Dictionaries” [154]”, Kory Stamper writes that lexicographers tend to fall into two categories: lumpers and splitters. Lumpers tend to give broad definitions that can cover several more minor variations on the meaning. Splitters tend to write a definition for each of those variations. A lumper’s definition might cause objects to fall under the definition while they shouldn’t. A splitter’s definition might become outdated, because the context of a term changes over time. For example, a police car was always blue, which could be in its definition. But this changed over time, and thus, the definition must be changed.

Finding: There should guidelines to determine the right granularity for the definition of a model element.

Impact: There are guidelines to determine if a model element should be split in two or more model elements, or vice versa, *i.e.*, the guidelines ‘Criterion for distinguishing a domain class’, ‘Criterion for generalizations’, ‘Criterion for compositions in domain models’, and ‘Use an abstract class for reoccurring involvements’.

However, the analogy with definitions in a vocabulary does not only pertain to the description of a model element. It is foremost the relations a model element has with other model elements. For example, if a domain class or function has two distinct lifecycle paths that are only united at the beginning and the end, then it is probably a case of lumping, *i.e.*, they should be distinct model elements. If two model elements

have the same model relations, *e.g.*, the same lifecycle and the same attribute, then it is probably the same model element.

A.4.6 Guidelines for deciding which concepts become a model element

Bookmark: In “Surfaces and Essences, Analogy as the Fuel and Fire of Thinking” [79] Hofstadter and Sander discuss that not all categories that a person can think of need to get their own word.

Finding: There should be guidelines determining which concepts are becoming a model element, and, consequently, get their own ‘word’, *i.e.*, name.

Impact: There are guidelines and criteria for distinguishing domain classes, generalizations, domain activities, functions, and attributes (see *Reflection ‘Guidelines for distinguishing model elements’ (cf. page 273)*).

Regarding analytical definitions, described under *Bookmark Stamper 1 (cf. page 267)*: typically an abstract subclass gets a name that ends with the name of its parent class, *e.g.*, firetruck is a truck. For non-abstract subclasses of an abstract parent class this is less common, *e.g.*, truck is a vehicle. This is just a heuristic. There can be professional domains where the naming evolved differently, and does not follow this heuristic.

MuDForM should offer derived concepts, through which one can define a category via a constraint that completely expressed in terms of other model elements. This is future work.

A.4.7 Guidelines for distinguishing model elements

See *Bookmark Pinker 7 (cf. page 283)* and *Bookmark Stamper 2 (cf. page 272)*.

Impact: There are currently a number of guidelines to determine if something should be a separate model element:

- Criterion for distinguishing a domain class
- Criterion for generalizations
- Criterion for compositions in domain models

Appendix A. Cognition-based Reflection Snippets

- A domain object is an instance of precisely one concrete domain class
- Only use invariants if other concepts cannot cover it
- Activity attributes refer to classes outside the domain
- Ensure the realizability of an activity
- Objects enter a role via an action
- Use an abstract class for reoccurring involvements
- Define a function for coherent behavior that will be assigned to an actor
- Define a separate function for function steps sequences that occur more than once
- An atomic object manipulation might indicate a domain activity
- Define only used dependencies
- Only use constraints when other modeling concepts do not suffice

The description of these guidelines can be found in the full method definition [33]. We have concluded in our SLR (see Chapter 2) that most methods do not have so many strict guidelines. We think that there can be more guidelines for this topic, e.g., for distinguishing attributes. Investigating this is future work.

A.4.8 Models represent situation-independent knowledge

Bookmark: In “Focus” [69], Daniel Goleman mentions that people can have situation-independent thought. They can reason and talk about situations that they are not in right now, or they can generalize knowledge from the situation that they are in.

Bookmark: In “How the mind works” [125], Steven Pinker reflects on quantification, and variable binding. Our compositional thoughts are often about individuals. The thought that a particular person eats an apple is different from the thought that people eat apples in general. We are capable of thinking things like ‘there is an X that Y’ without knowing such X, or things like ‘for all Xs: Y’ without knowing all Xs.

Finding: Capturing knowledge through both types of thoughts must be supported in the method. But, there is no need to make a MuDForM-specific metamodel for the concepts of predicate logic, and other formalisms like calculus or propositional logic.

A.4 Model Engineering through Method Engineering

Of course, making specifications is capturing knowledge and making decisions about situations that occur at a different moment than the moment of specifying.

Impact: Both cases are supported during knowledge elicitation. Experts can make statements that are generic, as in ‘People can order a book, or make instance-specific statements, as in ‘John ordered Lord of the Rings’. The first case can immediately be captured in a model. The second case needs to be generalized by asking what kind of thing ‘John’ is, what kind of thing ‘Lord of the Ring’ is, what kind of thing ‘orders’ is, and what other things can ‘order’ or be ‘ordered’.

It also works the other way around. What is captured by the specification of any model element pertains to all instances of that model element. Furthermore, constraints can use the quantifiers ‘for all’, and ‘there exists’, when extra properties need to be specified for the instances of some model element.

A.4.9 Support knowledge elicitation from both semantic memory and episodic memory

Bookmark: In “A Very Short Tour of the Mind” [26], Michael Corballis writes about the notions of episodic memory and semantic memory. Episodic memory is memory for events or episodes in our lives. For example, you remember what you ate this morning. Semantic memory is memory for statements about the world. For example, to remember that Amsterdam is the capital of the Netherlands, or to remember a word for some concept.

Finding: These notions might be relevant for modeling. But how?

Impact: The two notions of memory are relevant in the extraction of knowledge from involved (domain) experts. You can ask questions about events that happen(ed) in their domain (from their episodic memory). And, you can ask questions to let them share general knowledge about the targeted domain or system (from their semantic memory). The first type of questions lead to answers about specific instances, which can be generalized to model elements. The second type of questions will lead to answers that are already on model level.

A model can be seen as a representation of semantic memory, and when the model it is executed, it creates events, which can be logged and form a representation of episodic memory. The guideline ‘Ask for examples when generic knowledge is hard to phrase’ appeals to the episodic memory of the involved expert.

Appendix A. Cognition-based Reflection Snippets

A.4.10 Traceability helps to prevent falsities

Bookmark: John McWhorter discusses in his lectures “The Story of Human Language” [111] design features of human language, introduced by Charles Hockett [77], which distinguish it from animal communication.

Bookmark: In “Talking to Strangers” [67], Malcolm Gladwell writes about the time that Prime minister Chamberlain went to Germany to talk to Hitler. Hitler told him that he had no plans to start a war, and Chamberlain believed him. Chamberlain did the same as what we all do when we talk to strangers. We look into their eyes and decide then if we believe them or not. Apparently, Chamberlain’s decision was incorrect.

Finding: MuDForM, and many other specification languages, have the features described by Hockett, except for the ones that are directly about auditory aspects. There is one design feature that requires reflection: Prevarication, *i.e.*, the ability to lie or deceive. When using language, humans can make false or meaningless statements. Of course, lying and deceiving is not helpful in system development and dealing with it is out of scope. But not speaking the truth is something to reckon, because several domain experts might say contradicting things about one topic, which means that some of them are not speaking the truth. It should be prevented and detected.

Impact: Objective O7 (see Section 1.1.3) explicitly states that modeling actions must be traceable. When domain experts give input for a modeling activity, their input can be logged. Furthermore, all the decisions can be logged in terms of the modeling concepts and steps, and followed guidelines. This means that any inconsistency somewhere in the modeling process can be traced back to the input. Of course, it is still possible that falsities end up in the model. The many consistent model viewpoints that each cover a distinct but related model aspect, and that are defined on top of one metamodel, help to detect inconsistencies in the input. Moreover, having self-contained specifications, as stated in Objective O3, makes it harder to deceive, because it prevents models to become ambiguous through the use of undefined terms.

A.4.11 Traceability and multiple viewpoints help to detect biases

Bookmark: In “The Believing Brain” [142], Michael Shermer talks about cognitive biases. A cognitive bias is a belief that something is true based on some cognitive state. For example, the belief that more people die in plane crashes than in car crashes,

A.5 Match Reality with Useful Categories

right after a plane crash was in the news, is called the recency bias. The book names a few, but there are many more biases⁴.

Finding: The method should help to prevent that the biases influence a specification process. This might not be equally important for all biases.

Impact: There is currently nothing in the method that actively helps to prevent biases. But, through logging of modeling decisions, specifications are traceable to statements made by involved people, which helps to prevent that specifications become affected by the modelers beliefs and the involved experts' biases. Letting a specification be validated by others, either directly or indirectly via a translation into natural language or another representation, helps to prevent that a specification is based on the biases of one person. It is future work to investigate if it is possible to find analysis and modeling guidelines that explicitly target a specific bias. A bias in an input text that is transformed into a model, is probably also present in that model; bias in, bias out.

A.5 Match Reality with Useful Categories

A.5.1 Categories enable inference of properties

Bookmark: In "How the mind works" [125], Steven Pinker concludes that the mind has to get something out of forming categories. That something is inference. Obviously, we can not know everything about every object. But we can observe some of its properties, assign it to a category, and from the category predict properties that we did not observe. If Mopsy has long ears, he is a rabbit; if he is a rabbit, he should eat carrots, go hippity-hop, and breed like, well, a rabbit.

Finding: Categorizing a thing should enable you to infer properties of the thing that are captured in the definition of the category.

Impact: If you know an instance of a *Classifier*, e.g., *Domain class* or *Domain activity*, then it must have values for the specified properties of that *Classifier*. If it does not, then either the *Classifier's* definition is incorrect, the classification is incorrect, or your knowledge about the instance is incorrect (or incomplete). Correct the model or the instance such that the consistency between them is ensured. This reasoning is captured in the modeling guideline 'Ensure consistency between classifiers and known instances'.

⁴See for example the long list and references on https://en.wikipedia.org/wiki/List_of_cognitive_biases

Appendix A. Cognition-based Reflection Snippets

A.5.2 Prevent mismatches between reality and model

Bookmark: In “The Stuff of Thought: Language as a Window into Human Nature” [126], Steven Pinker stresses the importance of good definitions of concepts. He uses the insurance policy that covers the attack of 9-11-2001 as an example. There was a discussion whether the attack is one or two events? This is relevant because the insurance policy was up to \$3.5 billion per event. So, depending on the attack being one or two events, \$3.5 billion or \$7 billion are paid. Apparently, the definition of ‘event’ was too broad to get a statement whether it was one or two events.

Finding: In the example, the definition of event was apparently too broad to simply decide whether it was one or two events. The insurance domain model should help to clarify whether the twin towers are one or two insured objects, and what the time span of one event can be. Besides the definition of event, also definitions of concepts like causation and effect of an event could be relevant for the specification of the insurance policy.

It is impossible to specify a concept in the real world completely, as mentioned under *Bookmark Pinker 7* (*cf. page 283*). Concepts that are an instance of a specification do not have that problem. A specification of a concept can be seen as a complete specification of that concept. For example, if a car is defined as a thing with wheels, then a bicycle and a shopping cart are also cars. This is okay if only the wheel-having property is important. But if you want to know how many passengers it can transport, it is clearly an incomplete definition.

The method should help to minimize the mismatch between model and reality. It should ensure consistency across model elements. Relevant instances, including their properties, must unambiguously fit the model elements.

Impact: The cognitive aspect Entity stands for concepts that have an identity. MuD-ForM offers the possibility to relate a model element to other model elements: it can be a whole, and it can be part of other elements, it can be a generalization or a specialization of other elements, it can be the type of other elements, and it can be in a constraint specification together with other elements. All the possible relations are defined in the metamodel, derived from the cognitive aspects model (see Section 3.4), and listed in Table 3.3 in Chapter 3. There are guidelines for determining if something is a separate *Classifier*, e.g., *Domain class* or *Domain activity* (see *Reflection ‘Guidelines for distinguishing model elements’* (*cf. page 273*)).

Our experience is that having one metamodel, explicit steps and viewpoints, and clear guidelines, helps to achieve a coherent and consistent set of model elements. Especially the analysis coherency between state (objects) and change (actions), when analyzing the object lifecycle of a *Domain class*, minimizes mismatches between the real world and the model.

A.5.3 Classifiers must represent functional categories

Bookmark: In “The Organized Mind” [104], Daniel Levitin discusses the notion and importance of categorizing: perceptual and conceptual. We categorize things because they look, sound, smell alike. But also because they can fulfill some function in some context. The latter categories are called functional categories.

Finding: MuDForM should support the detection of functional categories. Objects do not belong to the same category because they look (or sound, or smell) the same, but they are the same because you do the same things with them. Vice versa, objects belong to a different category when you do something different with them, *i.e.*, they undergo different types of actions.

Impact: MuDForM has several guidelines driven by the notion of functional categorizing: ‘Criterion for distinguishing a domain class’, ‘Criterion for generalizations’, ‘Criterion for compositions in domain models’, ‘Activity attributes refer to classes outside the domain’, ‘Ensure the realizability of an activity’, ‘Use an abstract class for recurring involvements’, ‘Objects enter a role via an action’, ‘Define a function for behavior that will be assigned to an actor’, and ‘Define a separate function for function steps sequences that occur more than once’. These guidelines also take care of normalization of a model. (See also the guidelines in *Reflection ‘Guidelines for distinguishing model elements’ (cf. page 273)*.)

A.5.4 Support multiple functional categories for one instance

Bookmark: In “The Language Instinct: How the Mind Creates Language” [123], Steven Pinker observes that similarity is in the eye of the beholder. Similarity does not only depend on the properties that two things share, but also on who makes the comparison and when. Consider a piece of baggage at an airport. A spectator might see shape, color, and brand. The pilot is more concerned with the weight, and the passenger with destination and ownership. Which two pieces of baggage are more alike is different for each of those three people.

Appendix A. Cognition-based Reflection Snippets

Finding: This related to the notion of functional categories as mentioned in *Reflection ‘Classifiers must represent functional categories’* (cf. page 279). Objects do not belong to exactly one category. In the example, it must be possible to consider one piece of baggage from the perspective of three categories.

Impact: MuDForM allows classifying objects throughout their life with different classifiers. The notion of abstract and concrete subclass is derived from the cognitive aspect Identity. When an instance is created, it gets an identity from exactly one concrete classifier. But the instance can be categorized by multiple abstract classifiers. In the model, these classes must be related to the concrete class, either as subclass or as parent class. In the example, a concrete, *i.e.*, non-abstract Class ‘piece of baggage’ can have three abstract subclasses: ‘physical object’ for spectators (which does not make much sense), ‘piece of cargo’, and ‘owned baggage’. Each subclass considers different properties.

A.5.5 A model is a complete specification of the world it describes

Bookmark: in “How the mind works” [125], Steven Pinker says that the rules of common sense are very hard to write down, and to derive. He gives three examples:

1. If there is a gallon of milk in a bottle and the bottle is in the car, then there is a gallon of milk in the car. But when there is a gallon of blood in your body, and your body is in the car, it would be strange to claim there is a gallon of blood in the car.
2. A car seat is a chair, and a chair is furniture, but it would be strange to say that a car seat is furniture.
3. If you open a jar of peanut butter, your head will not evaporate.

An intelligent being has to deduce the implications of what it knows. But only the relevant implications. This is a challenge for robot design and for epistemology.

Finding: The examples address the discrepancy between concepts, *i.e.*, meaning, and their wording. Strictly seen, there is a gallon of blood in the car, and you can put a car seat in your house as furniture. You even don’t know for sure that your head will not evaporate (coincidentally) at the same moment you open a jar.

Impact: With MuDForM, just as with any other specification method, you specify what you find relevant. For example, if you do not model that people have a shoe size, then, in your perception of the world, they do not have it.

MuDForM is not a deduction engine, nor does it magically add “common sense” to specifications. But a model contains relations between concepts, which can be used in a deduction about instances that comply with the model. Regarding example 2, if you model that all car seats are chairs, and all chairs are furniture, then indeed a car seat is furniture. It is possible to specify constraints on generalizations, *e.g.*, ‘a chair is only furniture if it stands in a room and is available to sit on’. As a result, a car seat is not furniture when it is stored in a warehouse of car parts, but it is when it stands in your living room for people to sit on.

A.5.6 It must be unambiguously determinable if an object belongs to a category

Bookmark: in “How the mind works” [125], Steven Pinker says that humans put things in categories to apply the knowledge about similar objects to another object. But it is very hard to define a category precisely. The bachelor example is mentioned again (see *Bookmark Pinker 7 (cf. page 283)*).

Finding: If there is no clear definition of a category, then it is not possible to unambiguously deduct if an object belongs to the category. If the category bachelor is not precisely defined, which is the case, then knowing that someone is a bachelor can only be achieved by stating that the person is a bachelor.

Impact: Instances of model elements are created via Classifiers. Objects (or anything with an identity) can be allocated to multiple Classifiers. These classifiers must be related in the model. Otherwise, the semantics of saying that an object belongs to several categories is not clear.

The instances of MuDForM models, can only undergo a change of state via an action. So, making someone a bachelor has to happen in a specific action, *e.g.*, to divorce. Another option would be that the bachelor state can be derived from other properties, but this requires a precise specification of the bachelor category.

Appendix A. Cognition-based Reflection Snippets

A.5.7 Different objects in the real world can correspond with one model instance

Bookmark Searle 1: Searle discusses Leibniz's law in his lectures on "Philosophy of Language" [140]: two objects are identical if and only if they have the same properties.

Finding: How does Leibniz's law manifest itself for instances of a MuDForM model? Do they follow the same rule?

Impact: It depends. An object in Leibniz's law corresponds with an instance of the cognitive aspect **Entity**. If an entity is purely conceptual and only exist as instance of a model, then two entities with the same properties are indeed the same entity, because the entities only exist as instance of a model, and as such can only have properties that are defined by the model. If the entities are physical, and they have the same properties through the eyes of the model, then they might not be the same in the real world. Namely, you might not have modeled all the 'real-world' properties. See the example of the hunter's arrows in *Bookmark Pinker 6* (cf. page 282). A distinguishable arrow is an **Entity**. An indistinguishable arrow is just a **Concept**, which becomes an **Entity** when it is identifiable, e.g., by a distinguishable visual mark on the arrow.

A.5.8 Objects differ if and only if they are distinguishable

Bookmark Pinker 6: In "The Stuff of Thought: Language as a Window into Human Nature" [126], Steven Pinker discusses that some indigenous languages only have number words for one, two, and many, in which even two stands for 'a couple' and not exactly two. These people do not need to count more, because they keep track of individual items. For example, in a hunter-gatherer tribe that is investigated, hunters keep track of individual arrows that they have. So, they do not need to count them. They know which arrows they have fired, and which ones not.

Finding: The example illustrates that objects can be counted, and compared via their attributes. If arrows are indistinguishable, you cannot say which arrows are fired.

Impact: If you distinguish your arrows, then they are domain objects with their own identity and state, hence, 'Arrow' is a *domain class*. If you can not distinguish them, and can only count them, then 'Arrow amount' is a context *class*. In such case, 'Quiver' can be a *domain class* with an *attribute* 'Number of arrows inside' with the *class* 'Arrow amount' as type.

This is also related to Leibniz's law, as mentioned in *Bookmark Searle 1* (*cf. page 282*). When two arrows can not be distinguished, they are the same thing. However, in a collection they can be distinguished by their position and thus can be counted. If you shake the quiver, you can not determine with certainty which arrow was where before the shaking.

Impact: MuDFoRM offers a few default values for multiplicities: zero or more, one, at most one, or at least one. It could be possible to add something similar to 'a couple', like in the example of the hunter's arrows. For example, a car can have many wheels, but typically four or six is the maximum. So, being able to say that a car has 'a couple' of wheels could be useful. It is ambiguous what the formal semantic difference with 'at least one' is, but the differences can be meaningful to people. Typically, when there are only a couple of objects, they are distinguishable, *e.g.*, the left front wheel of the car, or my best long-distance arrow.

A.5.9 Objects are not tied to one category

Bookmark Spinoza 1: In "The Giants of Philosophy" [101], Charlton Heston narrates about Spinoza. Spinoza rejects the view of ancient philosophers that things have an essence, *i.e.*, each thing belongs to one category that defines its essential properties. He rejects it, because a thing can undergo many changes, resulting in different properties, while maintaining its identity. Second, imagination and perspective are very unreliable.

Bookmark Pinker 7: Steven Pinker observes in "Words and Rules" [124] that People think in categories. People make abstractions, *i.e.*, categories, and recognize an object as belonging to a category. Pinker even states that to understand mental categories is to understand much of human reasoning. We are not dumbfounded by every new turtle we see. We categorize it as a turtle and expect it to have certain traits, like being slower than a hare, or having a shield. Concepts in the mind pick out categories in the world. The simplest explanation of concepts is that they are conditions for membership in a category, a bit like definitions in a dictionary. These are called classical categories; a notion that was challenged by Ludwig Wittgenstein. (Remarkable is that Pinker assigns the statement that the notion of classical categories is wrong to Wittgenstein, but that Lang *et al.* [101] state that Spinoza already made that statement centuries earlier (see *Bookmark Spinoza 1* (*cf. page 283*)).

It is difficult and sometimes impossible to define categories in the classical way. The example of 'bachelor' is given. Is the pope a bachelor? Is a one-month-old boy? Is a

Appendix A. Cognition-based Reflection Snippets

man of 90 years old whose wife has just died? etc. Categories often have prototypes; a sparrow is a better example of a bird, than a penguin.

Bookmark: Hofstadter and Sander observe in “Surfaces and Essences, Analogy as the Fuel and Fire of Thinking” [79] that categories are very widespread. Not only physical things, or an abstract concept like love defines a category, but also the concept of ‘But’ can be seen as a category. It is repeatedly stated that categories as a clear box is very misleading, because things do not belong in precisely one box.

Finding: The notion of category must be supported. But objects should not be tied to one category. They can initially be created from a class, *i.e.*, a type. But it should also be possible to change the categorizations of an object.

The problem of deciding if a thing in the world belongs to a specific category is less present in a system/world that is governed via specifications made in a specification language. Namely, a thing is a member of a category if you state it is. The specification problem is defining as accurate as possible what the criteria for membership are. The modeling language and the guidelines should help in defining a concept, *i.e.*, a model element.

Impact: The cognitive aspect **Category** represents category as meant by Pinker. **Category** is a subclass of **Concept**. It is implemented via the modeling concept **Classifier**, which can have **Generalization** relations to other **Classifiers**. **Classifiers** can have instances, *i.e.*, values and objects, and events and actions.

In MuDForM, like in many other specification and programming languages, instances are created from a type, which gives them a set of predefined properties. It is possible that those properties change over time. MuDForM currently covers this by relating the types to each other. For example, a person can become a customer, which means that they can have a customer account, or a person can become an employee, which means they get a salary. In this case, the class **Person** is a generalization of both the class **Customer** and the class **Employee**.

All aspects in the cognitive aspects model (Figure 3.2) are a subclass of concept. This implies that an instance of any cognitive aspect can also be an instance of any other cognitive aspect. For example, a thing that is seen as an action, can also be seen as an object. In the MuDForM metamodel, this perspective is reflected via the class **Classifier**. Objects can become attributes of another object via domain actions. Or attributes can be transformed into objects. Note that typical (OO programming) languages do not allow objects to change their class.

The current method definition does not provide operations to change the type of an instance. Investigating what operations are needed is future work. Domain objects can be assigned to a different type, *i.e.*, be assigned to another domain class, while retaining meaning, as long as the new domain class has attributes that refer to the same attribute types (or subclasses thereof). For example, an instance of the domain class Employee with the name John can be retyped to domain class Person with the name John, as long as they refer to the same context class Name. Retyping objects requires modeling concepts that reflect what to do with instances when their type changes. Retyping is for example needed when domains are integrated, *e.g.*, when two domain class instances are identified as the same object when their domain classes are unified.

It should also be possible to state that an object is itself also a class. Then the step from metamodel to model becomes recursive, *i.e.*, there is no difference between model as a type and a model as an instance. There are only instances that comply to a model. Such a model could be called a metamodel, if its instances are models themselves. Enabling this is future work.

A.5.10 There are no exact category definitions in the real world

Bookmark: In “How the mind works” [125], Steven Pinker refers to the book “Women, Fire, and Dangerous Things” [99] by George Lakoff named after a fuzzy grammatical category in an Australian language. Lakoff argues that pristine categories are fictions. They are artifacts of the bad habit of seeking definitions, a habit that we inherited from Aristotle and now must shake off.

Finding: This is similar to the bookmarks where authors referred to Spinoza (*Bookmark Spinoza 1 (cf. page 283)*) and Wittgenstein (*Bookmark Pinker 7 (cf. page 283)*), who claimed that things do not belong to an exact category. But that does not mean that categories are not useful.

In the light of a model and its instances: you cannot talk about instance properties that you did not define in the model. So, it must be possible to define a model that can capture all relevant properties of all relevant instances. Which means, it should be possible to generalize instance properties and capture them in a category, and it should be possible to give instances extra properties that are captured in another category, where the latter category has as property that its instances also belong to the first category.

Appendix A. Cognition-based Reflection Snippets

Impact: A Classifier has a Generalization structure, which contains Generalizations. The modeling concept Generalization is based on the cognitive aspect characteristic of Concept, which is described in Table 3.3. For example, being a car is a characteristic of being a firetruck. This might seem counter-intuitive, but it is the same for the gray haired example (see *Bookmark Baggini 1* (*cf. page 292*)). Being a gray haired cat, *i.e.*, belonging to the category of gray haired cats, is the same a being a cat with gray hairs, *i.e.*, being a cat that has the characteristic of a gray hair.

A.6 Time is Perceived through Events

A.6.1 Events are ordered in time, and have a duration

Bookmark Pinker 8: In “The Stuff of Thought: Language as a Window into Human Nature” [126], Steven Pinker observes that time is encoded in grammar in two ways. First, in tense, *i.e.*, past, present, and future. The other is called aspect, which is about the shape of the event. For example, instantaneous, as in “swat a fly”, open-ended, as in “running around”, and marking completion, as in “draw a circle”. These are often confused in language, but are conceptually different. There is a third one is needed: reference time, *i.e.*, an event to which is referred. For example, I ate before I showered, in which, I showered is the reference time.

Finding: Regarding tenses: it must be possible to relate the execution time of one event relative to the execution time of any other event. The shape of events must also be covered in specifications.

Impact: The cognitive aspects model (see Section 3.4) has an order relation between Events. There is also the relation changes between Action and Object, which has a timing aspect, as one can speak of the moment before and after an action, and can consider how the object is changed. Regarding the event shape: an Activity can be atomic or non-atomic. Actions that are an instance of an atomic activity have a timestamp by definition. Instances of non-atomic activities always have a starting time, and if they are finished, they also have a stopping time.

A.6.2 The past is determined by the events that have happened

Bookmark Sigman 1: Mariano Sigman observes in “The Secret Life of the Mind: how our brain thinks, feels, and decides” [145] that in most languages, the future is in

front of us and the past is behind us, *i.e.*, we move forward and leave the past behind. But in the language Aymara⁵, it is the opposite. The past is in front of us, because you can see it. The future is behind us, because you cannot see it.

Finding: Seeing the future or past being in front or behind us does not impact on the MuDForM definition. Time is perceived through events. The notion of time needs to be present in the method. It should be possible to see what happened in the past and to see what will or can happen in the future.

Impact: The cognitive aspect Event is the anchor for time. There is no time without events. Events can be logged and have timestamp. There are three modeling concepts that cover Event and its derivative cognitive aspect Action: *Operation*, *Domain activity*, and *Function*.

A.6.3 Lifecycles enable stepping through time

See *Bookmark Pinker 8* (cf. page 286) and *Bookmark Sigman 1* (cf. page 286).

Impact: MuDForM does not provide in tense. But, object lifecycles and function lifecycles (flows), allow, as a matter of speaking, to virtually travel through time. An object and function execution are always at some point in their lifecycle. One can speak of the steps that happened, the steps that happen now, and the steps that might or must happen. The flow structure pattern, which is the foundation for lifecycles offers several possibilities to order events in time, *e.g.*, sequential or parallel, optional or obligatory, which allow you to consider what can happen next. It also allows simulating what to do next and what the impact of that action will be. Via logging of all the things that already happened, it is also possible to see the past.

It is also possible to let the start or end of an action depend on a constraint. Such a constraint can be expressed in terms of another event. To make a flow executable, constraints on the starting or stopping of an action, which involve other events, must be after those events. For example, you can not specify ‘start cooking two hours before dinner’. Simply because you do not know if and when the dinner will happen. Of course, saying ‘start cooking two hours before you planned to have dinner’ is possible.

⁵https://en.wikipedia.org/wiki/Aymara_language

Appendix A. Cognition-based Reflection Snippets

A.7 Building Modeling Concepts from Natural Language

A.7.1 Language is common, but there is no common language

Bookmark: John Searle expresses his doubts about Chomsky's universal grammar [23] in his "Philosophy of Mind lecture series" [139]. He mentions that Chomsky has weakened the claim that humans have innate principles for acquiring language, instead of hard innate rules for languages, which Searle finds more plausible. The debate about the nature of a language acquisition device in our brain is still going on. There are also no rules identified that all linguists agree on.

Finding: We conclude that the method should be based on principles that are common across most languages, and the mental principles that enable language acquisition. The fact that this is not 100% possible does not disqualify the idea.

Impact: Objective O6 states that the concepts on which the method should be based are not limited by natural language. Natural language is a starting point. Cognitive concepts, in this case the ones that are needed to acquire language, are more fundamental, which is reflected in Objective O6

A.7.2 Support for words that are common in all languages

Bookmark: In 'How the mind works' [125], Steven Pinker says that all human cultures ever documented have words for the elements of space, time, motion, speed, mental states, tools, flora, fauna, and weather, and logical connectives (not, and, same, opposite, part-whole, and general-particular).

Finding: Some of the mentioned word categories should be supported directly in the metamodel; some seem to be too domain-specific for a generic specification method.

Impact: All the mentioned words can be processed during Grammatical Analysis. But they are not supported equally.

'Not', 'and', and 'same' are directly supported, either via a structure or in a constraint. Part-whole is supported via Attributes. General and particular are supported via Generalization and instantiation. Opposite supported via ordered enumerations, like in 'small, medium, large' where small is the opposite of large. When there are only two possible values, then the opposite concept is the same as 'not'.

Space is currently not supported, but via Attributes and Activities roles, motion and location can be expressed. As mentioned in *Reflection ‘Support specification of location, force, agency, and causation’ (cf. page 293)*, it makes sense to support motion and location more directly.

Mental states are not supported, because that would make MuDForM tied to the topic of human conditions, which clearly goes too far. Tools, flora, fauna, and weather can be seen as specific domains, which could be modeled using the provided modeling concepts.

A.7.3 Support part of speech concepts

Bookmark: In “Word by Word, The Secret Life of Dictionaries” [154], Kory Stamper refers to the general concepts of part of speech. In English the main parts of speech are noun, pronoun, adjective, determiner, verb, adverb, preposition, conjunction, and interjection.

Finding: It makes sense that the method supports the part of speech concepts. Unless there is a reason not to.

Impact: During grammatical analysis, the different part of speech concepts become model candidates in the following way:

- Noun → Class (or one of its subclasses), or Attribute, or sometimes a feature.
- Pronoun → Function attribute, typically at feature level.
- Adjective → value of Attribute, or sometimes a subclass. See example about gray hair (see *Bookmark Baggini 1 (cf. page 292)*), which can be modeled as value of an Attribute hair color, or as a Class of gray haired cats.
- Determiner: This depends on the type of determiner:
 - Definite article → an Attribute, typically of a Function.
 - Indefinite article → an Involvement in an Activity, or a declaration of a Function attribute.
 - Quantifier → Either in the definition of a constraint, or the quantified property is in the definition of a classifier that pertains to the quantifier itself.

Appendix A. Cognition-based Reflection Snippets

- Demonstrative → an Attribute, typically of a Function or a Domain activity.
- Possessive → Attribute, or Life dependency.
- Verb → Activity, *i.e.*, Operation, Domain activity, or Function.
- Adverb → value of Attribute of an Activity, or sometimes as sub-Activity.
- Preposition → Activity role, or Attribute, or Life dependency.
- Conjunction: This depends on the meaning of the conjunction. ‘and’ and ‘or’ indicate a static structure of some Classifier. ‘but’, ‘while’, and ‘because’ indicate a Constraint.
- Interjection: It does not make sense to have a modeling concept for this. Interjections are hardly used in texts for system specification.

So all relevant part of speech concepts are supported. For some there are guidelines, but there could be better guidelines in case the transformation is not straightforward. For example, in the gray hair example (see *Bookmark Baggini 1 (cf. page 292)*), it is not clear from just the text which transformation should be chosen. But there could be guidelines for it, making the grammatical analysis process more systematical. Detecting and formulating them is future work.

A.7.4 Support modeling verbs, and their temporal contour

Bookmark: Hofstadter and Sander observe in “Surfaces and Essences, Analogy as the Fuel and Fire of Thinking” [79] that verbs are also labels of categories.

Bookmark: In “The Stuff of Thought: Language as a Window into Human Nature” [126], Steven Pinker writes that linguists sort verbs in classes based on their temporal contour. There are verbs for states, like ‘knowing an answer’ or ‘being in Amsterdam’, in which nothing changes. There are verbs for events, in which something happens. Events are divided into those that can go indefinitely, like ‘running around’ or those that have an end point, like ‘winning a race’. The end point indicates a change of state.

Verbs are also divided whether they describe an event that is durative or instantaneously. There are also iterative verbs, like ‘pound a nail’, or verbs for the inception of state, like ‘sit down’.

A.7 Building Modeling Concepts from Natural Language

Finding: It should be possible to see verbs as categories. The different verb classes must be supported, unless there is a reason not to.

Impact: Event and its subclass Action are cognitive aspects that reflect active verbs. MuDFoM has the metaclass Activity, which can have instances that express Events. Operation instances reflect Events. Instances Domain activities and Functions reflect Actions, because they also involve a state change to one or more Objects.

Verbs for states are captured in different ways: via Attributes (the car is blue, or the car has a blue color), via instances of Classes (John is a man), via Generalizations (cars are vehicles), via Class relations (the glass stands on the table, in which a stand relation exists between the glass and the table). Grammatical analysis has two Phrase types for state-oriented verbs: the ISA phrase, *i.e.*, state structure phrase state phrase, and the HAS a phrase, *i.e.*, static structure phrase. There are guidelines for deciding how to model verbs that express a state.

Verbs that indicate an event can be modeled as Activities, *i.e.*, as Operations, Domain activities, or Functions. All actions, *i.e.*, instances of Activities, are started explicitly. If the actual starting is not in scope, then observing that an action has started is the instantiation of the action. There are guidelines for deciding if an event verb is classified as an Operation, Domain activity, or Function.

Duration of actions is supported in two ways. Domain activities can be atomic or non-atomic. Non-atomic actions have an explicit starting moment, and, if they complete, have an explicit stopping moment. Atomic actions are either completed or not completed. Physically they take time, but conceptually not. For example, when you order a book online, it takes time to process the order in the system. But, the book is either ordered or not ordered, because the domain activity ‘to order’ is atomic. The whole online steps you go through of selecting a book title, is typically a Function and is not atomic. Operations, which are context model elements, are always atomic, because they are executed outside the scope of the domain and feature model.

A.7.5 Support to be, to have, to do, and to go

Bookmark: In “Words and Rules” [124], Steven Pinker observes that the verbs to be, to have, to do, and to go, are present in most languages. Many language scientists believe that the meanings of these verbs, existence, possession, action, motion, are at the core of the meaning of all verbs. For example, the mind treats telling him a story as causing the story to go to him, resulting in him having it.

Appendix A. Cognition-based Reflection Snippets

Finding: There should be a modeling concept for each of the mentioned verbs, unless there is a reason not to.

Impact:

- Existence (to be) is reflected in 1) instantiation, *e.g.*, the object ‘my car’ is an instance of the class ‘car’, 2) in Generalizations, *e.g.*, a ‘car’ is a ‘vehicle’, and 3) in Attributes, *e.g.*, ‘my car is blue’ can be modeled by giving the class ‘car’ an attribute ‘color’, and stating that the ‘color’ of ‘my car’ is ‘blue’. (See also *Reflection ‘Support different meanings of to be’ (cf. page 292)*.)
- Possession (to have) is reflected via 1) Attributes, *e.g.*, ‘my car’ has ‘four wheels’, and 2) via object life dependency, *e.g.*, an ‘order’ is dependent on a ‘customer’, which means that an order can not exist without a customer who placed the order.
- Action (to do) is reflected via the concept activity, *e.g.*, a Domain activity ‘to place an order for a customer’, or a Function ‘order books online’.
- Motion (to go) is currently not reflected directly. But, function flows can capture that an object that just participated in one step, can then participate in another step. Another kind of motion is the reaction to events, and the creation of events, which denote interactions with the context of a function execution. It is an option to support the notion of motion, and location, with specific modeling concepts. This requires more investigation in future work. (See also *Reflection ‘Support specification of location, force, agency, and causation’ (cf. page 293)*.)

A.7.6 Support different meanings of to be

Bookmark Baggini 1: In “How to Think Like a Philosopher: Essential Principles for Clearer Thinking” [11], Julian Baggini talks about the four different meanings of the English verb ‘to be’:

1. To be An instance of something; Felix is a cat.
2. To have a property; Felix is Furry.
3. To be Somewhere; Felix is in his basket.
4. To mean something; To be a cat is to be free.

A.7 Building Modeling Concepts from Natural Language

Some languages have different verbs for each of these meanings. There is a strong relation between being an instance and having a property. For example, Felix is a gray haired cat (meaning 1) vs. Felix has gray hair (meaning 2).

Finding: The different meanings of ‘to be’ should be supported in the grammatical analysis. There is no intrinsically correct model of ‘is a’ (meaning 1) and ‘has a’ (meaning 2). But there should be guidelines to help to decide how to model it.

Impact: There are guidelines during grammatical analysis for deciding how to represent the different meanings of ‘to be’.

The guideline ‘Criterion for distinguishing a domain class’ helps the modeler to make a decision if something should be modeled as a generalization to a separate class (meaning 1), or as an Attribute (meaning 2). In the example, should you define Domain classes ‘gray haired cat’ and its generalization ‘cat’, an Attribute ‘hair color’ with ‘gray’ as possible value, or even a boolean attribute ‘gray haired’? If gray haired cats are related to specific activities, which are not related to cats in general, then you should define separate classes for them. If there are many possible colors, and there are no different, related domain activities for each color, then model an Attribute ‘hair color’ of the class ‘cat’. If only the color gray is relevant for some activity that is related to the class ‘cat’, then model the boolean attribute.

Meaning 3 is not directly supported, because there are no concepts yet for modeling locations. It can be covered by either an Attribute ‘location’ with a possible value ‘his basket’ when basket is not a domain class, or by a relation between domain class ‘cat’ and ‘basket’ in case ‘basket’ is a domain class.

Meaning 4 is addressed via the same reasoning as for meaning 1 and 2. So, via a generalization of via an Attribute.

A.7.7 Support specification of location, force, agency, and causation

Bookmark: In “How the mind works” [125], Steven Pinker observes that location in space is one of the two fundamental metaphors in language. It is used for thousands of meanings. The other metaphor is force, agency, and causation.

Finding: As these metaphors are widespread, we see them as candidates for cognitive aspects.

Appendix A. Cognition-based Reflection Snippets

Impact: Agency is covered by the cognitive aspect Agent. An Agent may perform Events, which reflects force. Force is implicitly also present in the order relation, *i.e.*, if a specification states that one Event must be followed by another, then this can be seen as an expression of force. Also, the combination of an Agent perceiving an Event, followed by that Agent performing an Event can be seen as force. Finally, causation is simply covered by the causes relation.

Location in space is not covered yet. But, as mentioned in *Bookmark Pinker 10 (cf. page 297)*, it maybe should. Location can be a subclass of Concept with relation Concept is at Location. Then the cognitive aspect Movement can be a subclass of Event with relation Movement of Concept, and optional relations Movement from Location and Movement to Location. Extending the cognitive aspects, and the derived modeling concepts, with location and movement is future work.

A.7.8 Support different types of speech acts

Bookmark: In the lectures on “Philosophy of Language” [140], John Searle discusses if the different types of speech acts can also be done with pictures. The types are assertive, directive, commissive, expressive, and declaration.

Finding: Should the different types of speech acts be covered by MuDForM, and, if so, how?

Impact: MuDForM does not prescribe a textual or graphical notation. We think that that all metamodel concepts can have both. The relation between the modeling concepts and the different classes of speech acts is not straightforward, but we can observe the following:

- Assertives are expressed through a domain model, because a domain model describes what can happen and what can exist. Explicitly stating a collection of instances of domain activities and domain classes, can be seen as an assertive.
- Directives are present in a different way. Function flows express what must happen. So, given that a Function is activated, the Function’s specification states what must happen, which can be seen as a directive.
- Commissives, which state what a result will be, can be expressed as post-conditions of Activities.

A.7 Building Modeling Concepts from Natural Language

- Expressives, which state emotions, are not covered, because specifying emotions does not match with the purpose of a specification. Of course, it is possible to model the domain of emotions.
- Declarations, is the main type of speech act that is covered. Namely, a model declares what types of things can exist and happen.

The mentioned modeling possibilities do probably not cover the full spectrum of speech acts. More investigation is needed to identify new modeling concepts and guidelines to address the specification of speech acts. This is a suggestion for future work.

A.7.9 Support Subject-Verb-Object sentences

Bookmark Pinker 9: In “The Language Instinct: How the Mind Creates Language” [123], Steven Pinker discusses that almost all languages have subject-verb-object (SVO) as the basic structure of a sentence; not necessarily in that order.

Finding: The method should seamlessly deal with this.

Impact: MuDForM complies with the SVO structure. The phrase type ‘interaction structure’ expresses a change to one or more objects. The format is: *(subject) TO verb object (preposition object)*^{*}. The verb from this type of phrase typically becomes an Operation, a Domain activity, or a Function.

A.7.10 Compound names correlate with hierarchy between concepts

Bookmark: In “The Organized Mind” [104], Daniel Levitin observes that all languages and cultures independently came up with naming principles so similarly that they strongly suggest an innate predisposition towards classification. For example, every language contains primary and secondary plant and animal names, *e.g.*, there are apples, and there are granny smiths. This extends to man-made things, *e.g.*, there are knives, and there are hunting knives.

Finding: The classification principle must be supported.

Appendix A. Cognition-based Reflection Snippets

Impact: The generalization relation between Classifiers is used to indicate subcategories of categories. Furthermore, analysis of a compound names can be used to find subcategories, e.g., the name ‘motor vehicle’ indicates that it is a subclass of ‘vehicle’. The guideline ‘Criterion for generalizations’ helps to decide if a generalization is useful in the model. In the example, if there are also non-motorized vehicles (in the targeted domain), and they can participate in different activities or they have different attributes, then a generalization and a class of non-motorized vehicles are useful. If there are non-motorized vehicles, but you do the same with them as with the motor vehicles, then just the class vehicle suffices. (See also the naming guidelines mentioned in *Reflection Involved experts decide about model element names, not the modeler*’ (cf. page 261).)

A.7.11 Attributes and steps support indexicals

Bookmark: In the lecture on “Philosophy of Language” [140], John Searle discusses indexical expressions in speech acts. An indexical is an expression whose content varies from one context of use to another. The standard list of indexicals includes pronouns such as ‘I’, ‘you’, ‘he’, ‘she’, ‘it’, ‘this’, ‘that’, plus adverbs such as ‘now’, ‘then’, ‘today’, ‘yesterday’, ‘here’, and ‘actually’. The context determines where they point to exactly.

Finding: The method should support modeling indexicals.

Impact: The step ‘Classify candidates’ has a guideline ‘Definite articles and indexicals indicate an (activity) attribute’. The step ‘Specify Function lifecycle’ has a guideline ‘Check for temporal words in the input text’.

Indexicals appear as Attributes in the specification of Step participants (in an Object, Action, or Function lifecycle). Namely, Attributes can be used anywhere in the scope of its Attribute container. For example, when ordering books online, the shopping cart can be seen as an indexical. It can be referred to in the steps of the ordering process and in the constraints of the ordering process.

Indexicals that express a temporal relation appear as relations between steps in a lifecycle, e.g., select a book and then pay, or in explicit constraints between steps, e.g., paying must be done within 15 minutes after selecting a book.

A.7.12 Support word definitions and word usage

Bookmark: In “From Bacteria to Bach and Back: The Evolution of Minds” [42], Daniel C. Dennett discusses the notion of words and tokens of a word, *i.e.*, the word with the meaning it stands for, and the use of the word in a sentence. The word ‘word’ has four tokens in the previous sentence, and two in this one.

Finding: The method must support this notion too.

Impact: The notion is present at many places, *e.g.*, 1) an Attribute is a token of the type of the Attribute, 2) the use of an Attribute in an Operation call in an Activity view, is a token of the Attribute, 3) a Function step in a Function lifecycle, which is a reference to an Activity, is a token of that Activity. The structure pattern and its derivatives manifest the notion. Any *Reference* in a *Structure* is a token of the *Referred item*.

Besides the notion being present in a model, there is also the instantiation of the model. For example, the occurrence of a domain class instance, aka domain object, can be seen as a token of the domain class.

A.8 More Cognitive Aspects and their Derived Modeling Concepts

A.8.1 Support possible connections between ideas

Bookmark Pinker 10: In “The Sense of Style” [127], Steven Pinker refers to the 1748 book on human understanding of philosopher David Hume: ‘There appear to be only three principles of connections between ideas: similarity, contiguity in time or place, and cause or effect.’ Those connections are expressed as relations between sentences or partial sentences.

1. Similarity: a) similarity: too, similarly, also. b) contrast: but, however, c) exemplification and generalization: either one first, d) exception: generalization first or exception first.
2. Contiguity: a) sequence in time: before, after, b) sequence in place: before, after.
3. Cause effect: a) result: cause first, b) explanation: effect then cause c) violated explanation: preventer effect, d) failed prevention: effect preventer.

Appendix A. Cognition-based Reflection Snippets

4. Pinker observes that one other major coherence relation doesn't easily fit into Hume's trichotomy: Attribution, *i.e.*, so-and-so believes such and such. Typical connectives: according to, stated that.

Finding: It seems logical that these principles for connections are supported by the method. We think that the concept of idea from Hume corresponds with model fragment (or model element), because those express some unit of meaning just like idea does.

Impact:

1. Regarding Similarity: the cognitive aspect **Analogy** reflects the specification of similarity. Stating that something belongs to a category is also an expression of similarity. Regarding Similarity:
 - a) It is possible to create model fragments that are similar to other model fragments, as described in *Reflection 'Support the specification of analogies'* (*cf. page 300*) on analogy making. What the semantics would be of saying that two model fragments are similar is not clear. However, it could be useful to model, *e.g.*, to see if properties of one model element also hold for another model element. Example: a car is similar to a horse. You can carry stuff with a horse. Can you also carry stuff with a car? Supporting the specification of similarity, *i.e.*, analogy, is future work.
 - b) It is possible to specify contrast connections via guards in structures. For example, 'they choose the shortest way to work, but not on Saturdays' can be represented with a guard on a function step where the route to work is chosen.
 - c) Exemplification and generalization occur during the model discovery stage, *i.e.*, during grammatical analysis, and during model engineering. Also when a model is executed, in order to test it, examples are instantiated from the model.
 - d) It is possible to specify exceptions via constraints.
2. Regarding Contiguity: The cognitive aspect **Event** has an order relation to **Event**, which reflects the sequence in time. Contiguity of place is not supported at the moment. It is future work, as suggested in *Reflection 'Support specification of location, force, agency, and causation'* (*cf. page 293*).

A.8 More Cognitive Aspects and their Derived Modeling Concepts

- a) Sequences in time can be specified via the lifecycle viewpoints. It is also possible to define constraints on the time between two events, *i.e.*, between steps in a lifecycle.
 - b) It makes sense to have modeling concepts specifically for location related aspects, like positioning and movement. This is not supported yet. How to support it is future work.
3. Regarding Cause-effect: An Event may cause another Event. An Agent may perform an Event, which is also a kind of causation. The specification of function lifecycles is specifying what happens when. Via the behavior modality of function steps, the observation and generation of events can be specified, which is also a kind of cause-effect relation.
 - a) The result of an activity is defined in its internal specification. For operations, this is done via a formula. For domain activities, this is done via its postcondition or its activity view. For functions, this done via the function lifecycle.
 - b) There is currently no way to specify the deduction of a cause of a result. But it is possible to first specify a postcondition of an activity, and then reason how that postcondition can be guaranteed through steps in the internal specification of that activity.
 - c,d) These are handled the same as a) and b), because the effect is then a negation of a result specification.
 4. Regarding Attribution: The cognitive aspect Property can be used to attribute one concept to another. Moreover, the notion of 'according to' is also present between a model and its instances. Namely, the instances are in accordance with the model. For example, an attribute Person.name of type Name states that the attribute has the properties of the class Name. Or, the use of an operation in an activity means that the activity works in accordance with how that operation works. The notion of 'believes' does not make sense for specifications.

A.8.2 Support the specification of hierarchy between concepts

Bookmark: Hofstadter and Sander observe in “Surfaces and Essences, Analogy as the Fuel and Fire of Thinking” [79] that concepts are hierarchical. We are building concepts from other concepts throughout our whole life. But it does not lead to a hierarchy of concepts in our mind. It is not determined that a newer concept

Appendix A. Cognition-based Reflection Snippets

comprises the older concept. The older concept is also changed by creating a new concept with it.

Finding: We conclude that hierarchy should be seen as something that can be expressed in a specification.

Impact: The cognitive aspect *Property* reflects hierarchy. It is possible to define new concepts from other concepts via several relation types, such as behavior life dependency, generalization, attribute, involvement. The main mechanism in the metamodel is the structure pattern and its derivatives (see Section 3.A.3).

A.8.3 Support making abstractions

Bookmark Hofstadter 2: In “Surfaces and Essences, Analogy as the Fuel and Fire of Thinking” [79] Hofstadter and Sander observe that we (people) are constantly abstracting. ‘This is not stupidity, but intelligence.’

Finding: Support making abstractions. A link to the ‘real’ world helps people to create and use abstract knowledge. In other words, abstractions must still be recognizable by people.

Impact: To make abstractions, MuDForM offers the concept of *Generalizations* between *Classifiers*, which is based on the cognitive aspect *Property*. Using types, *i.e.*, *Classifiers*, is itself also a mechanism to create abstractions. Namely, the metamodel element *Classifier*, which is based on the cognitive aspect *Category*, is an abstraction of a set of instances. For example, the classifier *Apple* can be seen as an abstraction of one or more specific apples. Making domain models is making abstractions.

A.8.4 Support the specification of analogies

Bookmark Hofstadter 3: Hofstadter and Sander focus in “Surfaces and Essences, Analogy as the Fuel and Fire of Thinking” [79] on analogy making being in the core of our mental processes. At the end of the book, Hofstadter and Sander state that category making and analogy making are really the same mental process. You cannot do one without doing the other.

Finding: Analogy making should be possible. But, we need to decide if we should distinct categories and analogies. And, if so, how?

A.8 More Cognitive Aspects and their Derived Modeling Concepts

Impact: In the cognitive aspects model (see Section 3.4), we have kept the difference between **Analogy** and **Category**, because they are expressed in different ways. An analogy is expressed as a relation between two things. A **Category** is expressed as an abstraction of a set of things. Not all experiences, *i.e.*, analogy makings, are leading to a specified category. Moreover, when designing a system, one can define categories of things that should be created without those things being experienced yet. So, an **Analogy** does not necessarily lead to an explicit **Category**. Vice versa, a thing can be said to belong to a **Category** without making an **Analogy** to one or more other things in that **Category**.

Category and **Analogy** are both a subclass of **Concept**. So, it is possible to state that a specific analogy and category are the same thing. How to support analogy making with specific modeling concepts requires more investigation and is future work.

A.8.5 Any model fragment can be seen as a pattern

Bookmark: Michael Shermer talks in “The Believing Brain” [142] about the concept of patternicity. Seeing patterns has been evolved as a mechanism for survival and reproduction, especially to relate cause and effect. The fact that people see patterns even when they are not there is not necessarily harmful to them.

Finding: It should be possible to specify patterns. It should be possible to apply patterns in a specification. Any piece of model should be usable as a pattern, and as such serve as the foundation for an analogy in the way Hofstadter and Sander (see *Bookmark Hofstadter 3 (cf. page 300)*) mean it. A model can be seen as a directed acyclic graph, without any external semantics, *i.e.*, a model does not have semantics because of the names of model elements. It has semantics because the model is completely expressed in terms of the metamodel.

Impact: It is possible to take a piece of model, and rename the parts. This is how each model fragment can be used as a pattern. It is not directly supported with specific modeling concepts for defining patterns.

The metamodel does currently not provide in an explicit pattern mechanism. Treating a model fragment as a pattern can simply been done by copying the model fragment and replacing the names by the desired names.

Appendix A. Cognition-based Reflection Snippets

The method definition can be extended with modeling use cases or modeling functions. One of those use cases can be the making of an analogy. To realize this, the metamodel should be augmented with modeling activities. Activities are now only modeled for the grammatical analysis phase. If also the metamodel is expressed in terms of activities, then applying a pattern could be seen as a (modeling) function with a flow that contains steps typed by those activities. Applying a pattern is basically copying a set of related model elements, *i.e.*, selecting a model fragment, then renaming each of them or replacing them with an existing model element, and then integrating the created model fragment into the target model. The realization of this use case is future work.

A.8.6 Distinguish performative from constative speech acts

Bookmark: In the lectures on “Philosophy of Language” [140], John Searle writes about the difference between performative speech acts and constative speech acts. A performative speech act is performing an action by saying something, *e.g.*, ‘waiter, get me a beer please’ is the act of ordering a beer. The constative speech act would be ‘the waiter is getting me a beer’. Another aspect is that a performative one is always true, the constative one can be false. Speech acts can also be both, *e.g.*, when the waiter says ‘I promise that I will get you a beer’. The promise part is performative, but the actually getting part is constative.

Finding: Information (software) systems, and administrative domains in general, are all about automating performative speech acts, *e.g.*, ordering a beer via an app on your phone. It can be useful to express the difference between performative and constative speech acts in a model.

Impact: In function flows, you can specify the behavior modality of a function step; you can say 1) that you execute the step, *i.e.*, the actor that is executing the function will also execute the function step 2) that you want the step to be executed, *i.e.*, an event is sent to the function execution environment to execute the step or 3) that you observe the step being executed (and react to it), *i.e.*, an event is received from the function execution environment that the step is executed. A software system can execute a function step if the action that defines it is a performative act. Constative speech acts can only be controlled by a software system, *e.g.*, via a call to the context to let some actor (person or machine) perform the act, or observed by a software system, *e.g.*, via a physical sensor or when an end user indicates that the action happened.

A.8 More Cognitive Aspects and their Derived Modeling Concepts

It could be useful to have a distinction between physical and conceptual activities in the domain model. Namely, it is in the nature of the activity and not in the context in which the activity is used. In the example of ordering a book, the software execution is the implementation of the ordering of a book by a person. In case of a physical action, like delivering a book, the software execution may control the delivery, but it does not do the delivery. Investigating the support for and benefits from different types of speech acts –there are more types than discussed here– is future work.

B Lingo, Neuro, Psycho

I, Robert Deckers, joined the summer school on Human Language in July 2016 in the hope to find concepts that could be used to define MuDForM. That didn't really happen, but it became clear to me that the academic fields related to natural language are disconnected from sciences related to languages for software engineering, software architecture, and specifications in particular.

The lectures, ranging from genetics, to neurology, to linguistics, inspired me to write the lyrics below. It was used in a jam session that some students had during an evening at the location of the summer school. A recording was made, but, regrettfully, its quality is too low to publish.

Lingo, Neuro, Psycho

We fire neurons for attention
To get a grip on architecture
We see coherence, but feel tension
Suggested by iconic gesture

We go slowly through morphology
We try so hard, white matter tracts
If we violate phonology
We'll have to pay some sin tax

Chorus:

*Lingo, neuro, psycho
This is the hypothesis
Be careful for aphasia
If you miss your synapsis*

What's happening inside our brain
When we read, write, listen, speak?
What shows our MRI?
A language witch, genetic freak

We strive for complete comprehension
Search for meaning in the words
Relieve concepts from detention
Found in monkeys, wuggs, and birds

Bibliography

- [1] Gustav Aagesen and John Krogstie. Analysis and design of business processes using BPMN. In *Handbook on Business Process Management 1*, pages 213–235. Springer, 2010.
- [2] S. Abirami, G. Shankari, S. Akshaya, and M. Sithika. Conceptual modeling of non-functional requirements from natural language text. In *Computational Intelligence in Data Mining - Volume 3*, pages 1–11, New Delhi, 2015. Springer India.
- [3] Anas Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault. A practical approach to bridging domain specific languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA*, volume 5. Springer, 2005.
- [4] Anas Abouzahra, Ayoub Sabraoui, and Karim Afdel. A metamodel composition driven approach to design new domain specific modeling languages. In *2017 European Conference on Electrical Engineering and Computer Science (EECS)*, pages 112–118. IEEE, 2017.
- [5] John R Anderson. *The Architecture of Cognition*. Harvard university Press, 2013.
- [6] John Robert Anderson. *Learning and Memory: An Integrated Approach*. John Wiley & Sons Inc, 1995.
- [7] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Extracting domain models from natural-language requirements: Approach and industrial evaluation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 250–260. ACM, 2016.

Bibliography

- [8] Erika Asnina. The formal approach to problem domain modelling within model driven architecture. In *9th International Conference on Information Systems Implementation and Modelling*, pages 97–104. CEUR-WS.org, 2006.
- [9] TNO Automotive. Research on integrated vehicle safety. <https://www.tno.nl/en/focus-areas/traffic-transport/expertise-groups/research-on-integrated-vehicle-safety/>, 2021. Accessed: 23-8-2021.
- [10] Alfred Jules Ayer. *Language, Truth, and Logic*, volume 10. Courier Corporation, 1952.
- [11] Julian Baggini. *How to Think Like a Philosopher: Essential Principles for Clearer Thinking*. Granta Books, 2023.
- [12] Ankica Barišić, Vasco Amaral, and Miguel Goulão. Usability driven DSL development with USE-ME. *Computer Languages, Systems & Structures*, 51:118–157, 2018.
- [13] Ankica Barišić, Vasco Amaral, Miguel Goulao, and Bruno Barroca. Quality in use of DSLs: Current evaluation methods. In *Proceedings of the 3rd INForum-Simpósio de Informática (INForum2011)*. Universidade NOVA de Lisboa, 2011.
- [14] Ankica Barišić, Vasco Amaral, Miguel Goulão, and Bruno Barroca. Evaluating the usability of domain-specific languages. In *Software Design and Development: Concepts, Methodologies, Tools, and Applications*, pages 2120–2141. IGI Global, 2014.
- [15] Len Bass, Paul Clements, and Rick Kazamn. *Software Architecture in Practice, third edition*. Addison-Wesley, 2012.
- [16] Susan Blackmore and Susan J Blackmore. *The Meme Machine*. Oxford Paperbacks, 2000.
- [17] Dennis Van Den Brand. Formalization of the ISO 26262 standard. Master's thesis, Eindhoven University of Technology, 2018.
- [18] Sjaak Brinkkemper. Method engineering: Engineering of information systems development methods and tools. *Information and Software Technology*, 38(4):275–280, 1996.
- [19] Sjaak Brinkkemper, Motoshi Saeki, and Frank Harmsen. Assembly techniques for method engineering. In *Advanced Information Systems Engineering: 10th International Conference, CAiSE'98 Pisa, Italy, June 8–12, 1998 Proceedings 10*, pages 381–400. Springer, 1998.

Bibliography

- [20] Edmilson Campos, Uirá Kulesza, Marília Freire, and Eduardo Aranha. A generative development method with multiple domain-specific languages. In *Product-Focused Software Process Improvement*, pages 178–193, Cham, 2014. Springer International Publishing.
- [21] Roy Chaudhuri, S., S. Natarajan, A. Banerjee, and V. Choppella. Methodology to develop domain specific modeling languages. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, pages 1–10. ACM, 2019.
- [22] Noam Chomsky. *Syntactic Structures*. Mouton, The Hague/Paris, 1957.
- [23] Noam Chomsky. *On Language: Chomsky's Classic Works Language and Responsibility and Reflections on Language in One Volume*. New Press, 1998.
- [24] Tony Clark, Andy Evans, and Stuart Kent. Aspect-oriented metamodelling. *The Computer Journal*, 46:566–577, 2003.
- [25] Tony Clark, Paul Sammut, and James S. Willans. Applied metamodelling: A foundation for language driven development (second edition). *CoRR*, abs/1505.00149, 2015.
- [26] Michael C Corballis. *A Very Short Tour of the Mind: 21 Short Walks Around the Human Brain*. Abrams, 2017.
- [27] J. S. Cuadrado and J. G. Molina. A model-based approach to families of embedded domain-specific languages. *IEEE Transactions on Software Engineering*, 35:825–840, 2009.
- [28] Gerald Czech, Michael Moser, and Josef Pichler. A systematic mapping study on best practices for domain-specific modeling. *Software Quality Journal*, pages 1–30, 2019.
- [29] Remco C de Boer, Rik Farenhorst, Jan S van der Ven, Viktor Clerc, Robert Deckers, Patricia Lago, and Hans van Vliet. Structuring software architecture project memories. In *8th International Workshop on Learning Software Organizations (LSO), Rio de Janeiro, Brazil*, pages 39–47. Springer, 2006.
- [30] Gero Decker, Remco Dijkman, Marlon Dumas, and Luciano García-Bañuelos. The business process modeling notation. In *Modern Business Process Automation*, pages 347–368. Springer, 2010.

Bibliography

- [31] Robert Deckers. Towards a human-centric development method. <https://github.com/robertdeckers/MuDForM>, 2018.
- [32] Robert Deckers. From text to model for the SBD History. <https://github.com/robertdeckers/CaseStudySBDHistory>, 2022.
- [33] Robert Deckers. MuDForM method definition. <https://github.com/robertdeckers/MuDForM>, 2022.
- [34] Robert Deckers. Cognition literature analysis. <https://github.com/robertdeckers/MuDForM>, 2023.
- [35] Robert Deckers and Patricia Lago. [dataset] slr dost replication package. DOI: 10.5281/zenodo.6645856, <https://doi.org/10.5281/zenodo.6645856>, 2022.
- [36] Robert Deckers and Patricia Lago. Methodical conversion of text to models: MuDForM definition and case study. In *Proceedings of the Forum at Practice of Enterprise Modeling 2022 (PoEM-Forum 2022), November 23-25*, volume 3327 of *CEUR Workshop Proceedings*, pages 113–127. CEUR-WS.org, 2022.
- [37] Robert Deckers and Patricia Lago. Systematic literature review of domain-oriented specification techniques. *Journal of Systems and Software*, 192(C), October 2022.
- [38] Robert Deckers and Patricia Lago. Specifying features in terms of domain models: MuDForM method definition and case study. *Journal of Software: Evolution and Process*, 2023.
- [39] Robert Deckers and Patricia Lago. Engineering MuDForM: A cognition-based method definition. *Software and Systems Modeling*, 2024. Under Submission.
- [40] Robert Deckers and Ruud Steeghs. *DYA|Software: Architectuurpak voor Bedrijfskritische Applicaties*. Kleine Uijl, 2010.
- [41] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.M. Jézéquel. Melange: A meta-language for modular and reusable development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 25–36. ACM, 2015.
- [42] Daniel C Dennett. *From Bacteria to Bach and Back: The Evolution of Minds*. WW Norton & Company, 2017.

Bibliography

- [43] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35:26–36, 2000.
- [44] Leandro Marques do Nascimento, Daniel Leite Viana, PAS Neto, DA Martins, Vinicius Cardoso Garcia, and SR Meira. A systematic mapping study on domain-specific languages. In *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, pages 179–187. Curran Associates Inc., 2012.
- [45] Dov Dori. Object-process analysis: maintaining the balance between system structure and behaviour. *Journal of Logic and Computation*, 5(2):227–249, 1995.
- [46] Dov Dori and Moshe Goodman. From object-process analysis to object-process design. *Annals of Software Engineering*, 2(1):25–50, 1996.
- [47] Mosa Elbendak, Paul Vickers, and Nick Rossiter. Parsed use case descriptions as a basis for object-oriented class model generation. *Journal of Systems and Software*, 87:1209–1223, July 2011.
- [48] M. Emerson and J. Sztipanovits. Techniques for metamodel composition. In *OOPSLA, 6th workshop on domain specific modeling*, pages 122–139. ACM Press, October 2006.
- [49] Gregor Engels and Stefan Sauer. A meta-method for defining software engineering methods. *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, pages 411–440, 2010.
- [50] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, pages 1–8. ACM, 2012.
- [51] Andy Evans, Girish Maskeri, Paul Sammut, and James S Willans. Building families of languages for model-driven system development. In *Workshop in Software Model Engineering, San Francisco, CA*, pages 1–9. Springer, 2003.
- [52] Eric Evans. Deconstructing the domain: A pattern language for handling large object models, 1999.
- [53] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

Bibliography

- [54] Daniel Everett. *How Language Began: The Story of Humanity's Greatest Invention*. Profile Books, 2017.
- [55] Ricardo de Almeida Falbo, Giancarlo Guizzardi, and Katia Cristina Duarte. An ontological approach to domain engineering. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 351–358. ACM, 2002.
- [56] Rik Farenhorst, Remco C de Boer, Robert Deckers, Patricia Lago, and Hans van Vliet. What's in constructing a domain model for sharing architectural knowledge? In *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 108–113. Knowledge Systems Institute Graduate School, 2006.
- [57] Michael Felderer and Vahid Garousi. Together we are stronger: Evidence-based reflections on industry-academia collaboration in software testing. In *Software Quality: Quality Intelligence in Software and Systems Engineering - 12th International Conference, SWQD 2020, Vienna, Austria, January 14-17, 2020, Proceedings*, volume 371, pages 3–12. Springer, 2020.
- [58] Donald Firesmith. Specifying reusable security requirements. *Journal of Object Technology*, 3:61–75, 2004.
- [59] Kevin Flanigan. *Building a Better Vocabulary*. The Great Courses, 2015.
- [60] Wan Fokkink. *Introduction to Process Algebra*. Springer Science & Business Media, 2013.
- [61] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.
- [62] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing support for model composition in metamodels. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 253–253. IEEE, 2007.
- [63] U. Frank. Outline of a method for designing domain-specific modelling languages. Technical report, Universität Duisburg-Essen, Institut für Informatik und Wirtschaftsinformatik (ICB), 2010.
- [64] Ulrich Frank. The memo meta modelling language (mml) and language architecture. Technical report, Universität Duisburg-Essen, Institut für Informatik und Wirtschaftsinformatik (ICB), 2011.

Bibliography

- [65] Pedro Gabriel, Miguel Goulão, and Vasco Amaral. Do software languages engineers evaluate their languages? In *Proceedings of the 13th Iberoamerican Conference on Software Engineering, CIBSE 2010, Cuenca, Ecuador, April 12-16, 2010*, pages 149–162. CIBSE - IberoAmerican Conference on Software Engineering Steering Committee, 2010.
- [66] Vahid Garousi, Dietmar Pfahl, João M. Fernandes, Michael Felderer, Mika V. Mäntylä, David C. Shepherd, Andrea Arcuri, Ahmet Coskunçay, and Bedir Tekinerdogan. Characterizing industry-academia collaborations in software engineering: evidence from 101 projects. *Empirical Software Engineering*, 24(4):2540–2602, 2019.
- [67] Malcolm Gladwell. *Talking to Strangers: What We Should Know About the People We Don't Know*. Little, Brown, 2019.
- [68] Paul Goldberger. *Why Architecture Matters*. Yale University Press, 2023.
- [69] Daniel Goleman. *Focus: The Hidden Driver of Excellence*. Bloomsbury Publishing, 2014.
- [70] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, and C. Guychard. Addressing modularity for heterogeneous multi-model systems using model federation. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 206–211. ACM, 2016.
- [71] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Using free modeling as an agile method for developing domain specific modeling languages. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 24–34. IEEE, 2016.
- [72] E Grant, Krish Narayanan, and Hassan Reza. Rigorously defined domain modeling languages. In *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling*, pages 1–8. ACM, 2004.
- [73] Emanuel S Grant. A meta-model approach to defining UML-based domain-specific modeling language. In *Proceedings of the International MultiConference of Engineers and Computer Scientists 2012 (IMECS 2012)*, volume 1, pages 780–785. Newswood Limited, 2012.
- [74] Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha-Pekka Tolvanen. DSLs: The good, the bad, and the ugly. In *Companion to*

Bibliography

- the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, page 791–794, New York, NY, USA, 2008. ACM.
- [75] Yuval Noah Harari. *Homo Deus: A Brief History of Tomorrow*. random house, 2016.
 - [76] Tanja Elina Havstorm and Fredrik Karlsson. Software developers reasoning behind adoption and use of software development methods – a systematic literature review. *International Journal of Information Systems and Project Management*, 11(2), 2023.
 - [77] Charles F Hockett. The problem of universals in language. *Universals of Language*, 2:1–29, 1963.
 - [78] Douglas R Hofstadter. *I Am a Strange Loop*. Basic books, 2007.
 - [79] Douglas R Hofstadter and Emmanuel Sander. *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking*. Basic Books, 2013.
 - [80] S. Hoppenbrouwers, A. Bleeker, and H. Proper. Modeling linguistically complex business domains. Technical report, Nijmegen Institute for Information and Computing Sciences, University of Nijmegen, 2004.
 - [81] A. Huberman et al. *Qualitative Data Analysis: A Methods Sourcebook*. Thousand Oaks, California SAGE Publications, Inc., 2014.
 - [82] M. Ibrahim and R. Ahmad. Class diagram extraction from textual requirements using natural language processing techniques. In *2nd International Conference on Computer Research and Development (ICCRD'10)*, pages 200–204. IEEE, 2010.
 - [83] Juhani Iivari and John R Venable. Action research and design science research—seemingly similar but decisively dissimilar. In *ECIS 2009 Proceedings*. AIS, 2009.
 - [84] ISO26262 ISO. 26262: Road vehicles-functional safety. *International Standard ISO/FDIS*, 26262-2, 2018.
 - [85] ISO/IEC. Systems and software engineering – recommended practice for architectural descriptions of software intensive systems, iso/iec 42010:2007. Technical report, ISO/IEC/IEEE, 2007.

Bibliography

- [86] ISO/IEC. Iso/iec 25010:2011: Systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models, 2011.
- [87] Anibal Iung, Joao Carbonell, Luciano Marchezan, Elder Rodrigues, Maicon Bernardino, Fabio Paulo Basso, and Bruno Medeiros. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering*, 25(5):4205–4249, 2020.
- [88] Samireh Jalali and Claes Wohlin. Systematic literature studies: Database searches vs. backward snowballing. In *Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement*, pages 29–38. IEEE, 2012.
- [89] JetBrains. MPS, meta programming system. <https://www.jetbrains.com/mps/>, 2021. Accessed: 2021-08-19.
- [90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.
- [91] R. Kazman, M. Klein, and P. Clements. *ATAM: Method for Architecture Evaluation*. Carnegie Mellon University, 2000.
- [92] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.
- [93] A Khabbaz Saberi. *Functional Safety: A New Architectural Perspective: Model-Based Safety Engineering for Automated Driving Systems*. PhD thesis, Eindhoven University of Technology, 2020.
- [94] Barbara Kitchenham. Procedures for performing systematic reviews. Technical Report 2004, Keele, UK, Keele University, 2004.
- [95] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.
- [96] Gerald Kristen. *Object-Orientation: The KISS Method : From Information Architecture to Information System*. Addison Wesley, 1994.
- [97] K. Kronlöf. *Method Integration: Concepts and Case Studies*. John Wiley and Sons, 1993.

Bibliography

- [98] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In *International Conference on Model Driven Engineering Languages and Systems*, pages 240–255. Springer, 2009.
- [99] George Lakoff. *Women, Fire and Dangerous Things: What Categories Reveal About the Mind*. University of Chicago press, 2008.
- [100] George Lakoff and Mark Johnson. *Metaphors we live by*. University of Chicago press, 2008.
- [101] Berel Lang et al. *The Giants of Philosophy*. Blackstone PUB, 2007.
- [102] K. Lano and S. Kolahdouz-Rahimi. Model-transformation design patterns. *IEEE Transactions on Software Engineering*, 40:1224–1259, 2014.
- [103] Kwanwoo Lee, Kyo C Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *International Conference on Software Reuse*, pages 62–77. Springer, 2002.
- [104] Daniel J Levitin. *The Organized Mind: Thinking Straight in the Age of Information Overload*. Penguin, 2014.
- [105] H. Lochmann and A. Hessellund. An integrated view on modeling with multiple domain-specific languages. In *Proceedings of the IASTED International Conference Software Engineering SE 2009*, pages 1–10. ACTA Press, 2009.
- [106] Yaping Luo, Mark van den Brand, and Alexandre Kiburse. Safety case development with sbvr-based controlled language. In Philippe Desfray, Joaquim Filipe, Slimane Hammoudi, and Luís Ferreira Pires, editors, *Model-Driven Engineering and Software Development*, pages 3–17, Cham, 2015. Springer International Publishing.
- [107] H. Mannaerts and Jan Verelst. *Normalized Systems*. Koppa BvBa, 2009.
- [108] Viorica Marian. *The power of language: How the codes we use to think, speak, and live transform our minds*. Penguin, 2023.
- [109] R. Marvie. A transformation composition framework for model driven engineering. Technical report, Laboratoire d’Informatique Fondamentale de Lille, 2004.
- [110] Andrew May. *Fake Physics*. Springer, 2019.

Bibliography

- [111] John H McWhorter, Jon Leven, and James Blandford. *The Story of Human Language*. Teaching Company, 2004.
- [112] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
- [113] MetaCase. Metaedit+ workbench user's guide version 4.5. Technical report, MetaCase, 2022.
- [114] Fahimeh Alizadeh Moghaddam, Robert Deckers, Giuseppe Procaccianti, Paola Grosso, and Patricia Lago. A domain model for self-adaptive software systems. In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, pages 16–22. Springer, 2017.
- [115] S Nayanamana and S Somé. Generating a domain model from a use case model. In *14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2005)*, volume 278. ISCA, 2005.
- [116] Bill Nye. *Everything All at Once: How to Unleash Your Inner Nerd, Tap Into Radical Curiosity, and Solve Any Problem*. Rodale, 2017.
- [117] OMG. MDA guide version 1.0. Technical report, OMG, May 2003.
- [118] OMG. Object constraint language version 2.4. Technical report, OMG, February 2014.
- [119] OMG. Meta object facility 2.5.1. Technical report, OMG, October 2016.
- [120] OMG. Unified modeling language version 2.5.1. Technical report, OMG, December 2017.
- [121] Kai Petersen, Cigdem Gencel, Negin Asghari, Dejan Baca, and Stefanie Betz. Action research as a model for industry-academia collaboration in the software engineering context. In *Proceedings of the 2014 international workshop on Long-term industrial collaboration on software engineering*, pages 55–62. ACM, 2014.
- [122] Kai Petersen, Cigdem Gencel, Negin Asghari, and Stefanie Betz. An elicitation instrument for operationalising gqm+ strategies (gqm+ s-ei). *Empirical Software Engineering*, 20(4):968–1005, 2015.

Bibliography

- [123] Steven Pinker. *The Language Instinct: How the Mind Creates Language*. William Morrow and Company, 1994.
- [124] Steven Pinker. *Words and Rules: The Ingredients of Language*. Basic Books, 1999.
- [125] Steven Pinker. *How the Mind Works*. Penguin UK, 2003.
- [126] Steven Pinker. *The Stuff of Thought: Language as a Window into Human Nature*. Penguin, 2007.
- [127] Steven Pinker. *The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century*. Penguin Books, 2015.
- [128] Rubén Prieto-Díaz. Domain analysis: An introduction. *SIGSOFT Software Engineering Notes*, 15(2):47–54, April 1990.
- [129] H.A. Proper, A.I. Bleeker, and S.J.B.A. Hoppenbrouwers. Object–role modelling as a domain modelling approach. In *Proceedings of the Workshop on Evaluating Modeling Methods for Systems Analysis and Design (EMMSAD'04)*, pages 317–328. CEUR-WS.org, 2004.
- [130] S. Purao, V. Storey, A. Sengupta, and M. Moore. Reconciling and cleansing: an approach to inducing domain models. In *International Workshop on Information Systems and Technologies (WITS)*, pages 61–66. Paul Bowen and Vijay Mookerjee, 2000.
- [131] Willard V Quine. On the reasons for indeterminacy of translation. *The Journal of Philosophy*, 67(6):178–183, 1970.
- [132] Paul Ralph. Possible core theories for software engineering. In *2nd SEMAT Workshop on a General Theory of Software Engineering (GTSE)*, pages 35–38. IEEE, May 2013.
- [133] Iris Reinhartz-Berger, Pnina Soffer, and Arnon Sturm. A domain engineering approach to specifying and applying reference models. *Enterprise Modelling and Information Systems Architectures*, 2005.
- [134] Iris Reinhartz-Berger and Arnon Sturm. Behavioral domain analysis — the application-based domain modeling approach. In *7th International Conference on The Unified Modeling Language. Modeling Languages and Applications. UML 2004. Lecture Notes in Computer Science, vol 3273*, pages 410–424. Springer-Verlag, 2004.

Bibliography

- [135] Sylvain Robert, Sébastien Gérard, François Terrier, and François Lagarde. A lightweight approach for domain-specific modeling languages design. In *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 155–161. IEEE, 2009.
- [136] Ildevana Poltronieri Rodrigues, Márcia de Borba Campos, and Avelino F Zorzo. Usability evaluation of domain-specific languages: a systematic literature review. In *International Conference on Human-Computer Interaction*, pages 522–534. Springer, 2017.
- [137] José Raúl Romero, Juan Ignacio Jaén, and Antonio Valleccillo. Realizing correspondences in multi-viewpoint specifications. In *IEEE International Enterprise Distributed Object Computing Conference*, pages 163–172. IEEE, 2009.
- [138] V. B. Vidya Sagar and S. Abirami. Conceptual modeling of natural language functional requirements. *Journal of Systems and Software*, 88, 2014.
- [139] John R Searle. Philosophy of mind, lecture 1-28. <https://www.youtube.com/-playlist?list=PL039MUyjHR1wfjpULVP1a1ZeCBmIHmhxt>, 2011. Accessed: 2021-10-20.
- [140] John R Searle. Philosophy of language, lecture 1-28. <https://www.youtube.com/playlist?list=PL8C19A595E537E3C9>, 2012. Accessed: 2020-08-19.
- [141] B. Selic. A systematic approach to domain-specific language design using UML. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 2–9, 2007.
- [142] Michael Shermer. *The Believing Brain: From Ghosts and Gods to Politics and Conspiracies – How We Construct Beliefs and Reinforce Them as Truths*. Macmillan, 2011.
- [143] S. Shlaer and S. J. Mellor. An object-oriented approach to domain analysis. *SIGSOFT Software Engineering Notes*, 14(5):66–77, July 1989.
- [144] Keng Siau. Information modeling and method engineering: A psychological perspective. In *Successful Software Reengineering*, pages 193–208. IGI Global, 2002.
- [145] Mariano Sigman. *The Secret Life of the Mind: How Your Brain Thinks, Feels, and Decides*. Little, Brown Spark, 2017.

Bibliography

- [146] M. Simos and J. Anthony. Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In *Proceedings Fifth International Conference on Software Reuse*, pages 94–102. IEEE, 1998.
- [147] Mark Simos, R Creps, Carol Klingler, and L Lavine. Software technology for adaptable reliable systems (stars). organization domain modeling (odm) guide-book, version 1.0. Technical report, Defense Technical Information Center (DTIC), 1995.
- [148] Mark A. Simos. Organization domain modeling (odm): Formalizing the core domain modeling life cycle. In *SIGSOFT Software Engineering Notes, Special Issue on the 1995 Symposium on Software Reusability*, pages 196–205. ACM, Aug 1995.
- [149] Steven Sloman and Philip Fernbach. *The Knowledge Illusion: Why We Never Think Alone*. Penguin, 2018.
- [150] John Smart. *BDD in Action: Behavior-Driven Development for the whole software lifecycle*. Simon and Schuster, 2014.
- [151] Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, Albert Ambroziewicz, and Tomasz Straszak. Complementary use case scenario representations based on domain vocabularies. In *International Conference on Model Driven Engineering Languages and Systems*, pages 544–558. Springer, 2007.
- [152] Sparx Systems. Enterprise architect version 15.2. <https://sparxsystems.com/products/ea/>, 2021. Accessed: 2021-08-19.
- [153] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.
- [154] Kory Stamper. *Word by Word: The Secret Life of Dictionaries*. Vintage, 2018.
- [155] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.
- [156] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.
- [157] A. Sturm, I, and Reinhartz-Berger. Applying the application-based domain modeling approach to UML structural views. In *the 23rd International Conference on Conceptual Modeling (ER'2004), Lecture Notes in Computer Science 3288*, pages 766–799. Springer, 2004.

Bibliography

- [158] R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures. *ACM SIGSOFT Software Engineering Notes*, 20:27–37, 1995.
- [159] Wesley Torres, Mark GJ Van den Brand, and Alexander Serebrenik. A systematic literature review of cross-domain model consistency checking by model management tools. *Software and Systems Modeling*, pages 1–20, 2020.
- [160] Antonio Vallecillo. On the combination of domain specific modeling languages. In *6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, volume 61381 of *Lecture Notes in Computer Science (LNCS)*, pages 305–320. Springer, June 2010.
- [161] B. van der Vos, J. Hoppenbrouwers., and S. Hoppenbrouwers. NL structures and conceptual modelling: The KISS case. In *Applications of Natural Language to Information Systems: Proceedings of the Second International Workshop*, page 197. IOS Press, 1996.
- [162] Roberto Verdecchia, Emelie Engström, Patricia Lago, Per Runeson, and Qunying Song. Threats to validity in software engineering research: A critical reflection. *Information and Software Technology*, 164:107329, December 2023.
- [163] N. Visic, H. Fill, R. A. Buchmann, and D. Karagiannis. A domain-specific language for modeling method definition: From requirements to grammar. In *IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, pages 286–297. IEEE, 2015.
- [164] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. Createspace Independent Publishing Platform, 2013.
- [165] Markus Völter. Best practices for DSLs and model-driven development. *Journal of Object Technology*, 8(6):79–102, 2009.
- [166] Yair Wand. Ontology as a foundation for meta-modelling and method engineering. *Information and Software Technology*, 38(4):281–287, 1996.
- [167] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16*, pages 1–17. Springer, 2013.

Bibliography

- [168] RJ Wieringa. Object-oriented analysis, structured analysis, and jackson system development. In *Object Oriented Approach in Information Systems*, pages 1–21. Elsevier Science Amsterdam, 1991.
- [169] Claes Wohlin. Second-generation systematic literature studies using snowballing. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–6. ACM, 2016.
- [170] Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.
- [171] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2017.
- [172] Tao Yue, Lionel C. Briand, and Yvan Labiche. A systematic review of transformation approaches between user requirements and analysis models. *Requirements Engineering*, 16(2):75–99, 2011.
- [173] Y. Zhang, X. Liu, Z. Wang, and L. Chen. A model-driven method for service-oriented modeling and design based on domain ontology. In *Computer, Informatics, Cybernetics and Applications. Lecture Notes in Electrical Engineering, vol 107.*, pages 991–998. Springer, 2012.

C SIKS Dissertation Series

- 2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VUA), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UvA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VUA), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VUA), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UvA), Search Engines that Learn from Their Users
12 Max Knobbiout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VUA), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization

Appendix C. SIKS Dissertation Series

- 15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
- 16 Guangliang Li (UvA), Socially Intelligent Autonomous Agents that Learn from Human Reward
- 17 Berend Weel (VUA), Towards Embodied Evolution of Robot Organisms
- 18 Albert Meroño Peñuela (VUA), Refining Statistical Data on the Web
- 19 Julia Efremova (TU/e), Mining Social Structures from Genealogical Data
- 20 Daan Odijk (UvA), Context & Semantics in News & Web Search
- 21 Alejandro Moreno Cálleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
- 22 Grace Lewis (VUA), Software Architecture Strategies for Cyber-Foraging Systems
- 23 Fei Cai (UvA), Query Auto Completion in Information Retrieval
- 24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
- 25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
- 26 Dilhan Thilakarathne (VUA), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
- 27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
- 28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
- 29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
- 30 Ruud Mattheij (TiU), The Eyes Have It
- 31 Mohammad Khelghati (UT), Deep web content monitoring
- 32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
- 33 Peter Bloem (UvA), Single Sample Statistics, exercises in learning from just one example
- 34 Dennis Schunselaar (TU/e), Configurable Process Trees: Elicitation, Analysis, and Enactment
- 35 Zhaochun Ren (UvA), Monitoring Social Media: Summarization, Classification and Recommendation
- 36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
- 37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry

-
- 38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
 - 39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
 - 40 Christian Detweiler (TUD), Accounting for Values in Design
 - 41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
 - 42 Spyros Martzoukos (UvA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
 - 43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
 - 44 Thibault Sellam (UvA), Automatic Assistants for Database Exploration
 - 45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
 - 46 Jorge Gallego Perez (UT), Robots to Make you Happy
 - 47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
 - 48 Tanja Buttler (TUD), Collecting Lessons Learned
 - 49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
 - 50 Yan Wang (TiU), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains
-
- 2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
 - 02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
 - 03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
 - 04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
 - 05 Mahdieh Shadi (UvA), Collaboration Behavior
 - 06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
 - 07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
 - 08 Rob Konijn (VUA), Detecting Interesting Differences: Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
 - 09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
 - 10 Robby van Delden (UT), (Steering) Interactive Play Behavior
 - 11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
 - 12 Sander Leemans (TU/e), Robust Process Mining with Guarantees

Appendix C. SIKS Dissertation Series

- 13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
- 14 Shoshannah Tekofsky (TiU), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
- 15 Peter Berck (RUN), Memory-Based Text Correction
- 16 Aleksandr Chuklin (UvA), Understanding and Modeling Users of Modern Search Engines
- 17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
- 18 Ridho Reinanda (UvA), Entity Associations for Search
- 19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
- 20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
- 21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
- 22 Sara Magliacane (VUA), Logics for causal inference under uncertainty
- 23 David Graus (UvA), Entities of Interest — Discovery in Digital Traces
- 24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
- 25 Veruska Zamborlini (VUA), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
- 26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
- 27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
- 28 John Klein (VUA), Architecture Practices for Complex Contexts
- 29 Adel Alhuraibi (TiU), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
- 30 Wilma Latuny (TiU), The Power of Facial Expressions
- 31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
- 32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
- 33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
- 34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
- 35 Martine de Vos (VUA), Interpreting natural science spreadsheets
- 36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
- 37 Alejandro Montes Garcia (TU/e), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
- 38 Alex Kayal (TUD), Normative Social Applications

-
- 39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
 - 40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
 - 41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
 - 42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
 - 43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
 - 44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
 - 45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
 - 46 Jan Schneider (OU), Sensor-based Learning Support
 - 47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
 - 48 Angel Suarez (OU), Collaborative inquiry-based learning
-
- 2018 01 Han van der Aa (VUA), Comparing and Aligning Process Representations
 - 02 Felix Mannhardt (TU/e), Multi-perspective Process Mining
 - 03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
 - 04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
 - 05 Hugo Huurdeman (UvA), Supporting the Complex Dynamics of the Information Seeking Process
 - 06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
 - 07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
 - 08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems
 - 09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
 - 10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
 - 11 Mahdi Sargolzaei (UvA), Enabling Framework for Service-oriented Collaborative Networks
 - 12 Xixi Lu (TU/e), Using behavioral context in process mining
 - 13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
 - 14 Bart Joosten (TiU), Detecting Social Signals with Spatiotemporal Gabor Filters
 - 15 Naser Davarzani (UM), Biomarker discovery in heart failure

Appendix C. SIKS Dissertation Series

- 16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
 - 17 Jianpeng Zhang (TU/e), On Graph Sample Clustering
 - 18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
 - 19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
 - 20 Manxia Liu (RUN), Time and Bayesian Networks
 - 21 Aad Slootmaker (OU), EMERGO: a generic platform for authoring and playing scenario-based serious games
 - 22 Eric Fernandes de Mello Araújo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
 - 23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
 - 24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
 - 25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
 - 26 Roelof Anne Jelle de Vries (UT), Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
 - 27 Maikel Leemans (TU/e), Hierarchical Process Mining for Scalable Software Analysis
 - 28 Christian Willemsen (UT), Social Touch Technologies: How they feel and how they make you feel
 - 29 Yu Gu (TiU), Emotion Recognition from Mandarin Speech
 - 30 Wouter Beek (VUA), The "K" in "semantic web" stands for "knowledge": scaling semantics to the web
-
- 2019 01 Rob van Eijk (UL), Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
 - 02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
 - 03 Eduardo Gonzalez Lopez de Murillas (TU/e), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
 - 04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
 - 05 Sebastiaan van Zelst (TU/e), Process Mining with Streaming Data
 - 06 Chris Dijkshoorn (VUA), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
 - 07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
 - 08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
 - 09 Fahimeh Alizadeh Moghaddam (UvA), Self-adaptation for energy efficiency in software systems
 - 10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction

-
- 11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
 - 12 Jacqueline Heinerman (VUA), Better Together
 - 13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
 - 14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
 - 15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
 - 16 Guangming Li (TU/e), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
 - 17 Ali Hurriyetoglu (RUN), Extracting actionable information from microtexts
 - 18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
 - 19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
 - 20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
 - 21 Cong Liu (TU/e), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
 - 22 Martin van den Berg (VUA), Improving IT Decisions with Enterprise Architecture
 - 23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
 - 24 Anca Dumitrasche (VUA), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
 - 25 Emiel van Miltenburg (VUA), Pragmatic factors in (automatic) image description
 - 26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
 - 27 Alessandra Antonaci (OU), The Gamification Design Process applied to (Massive) Open Online Courses
 - 28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
 - 29 Daniel Formolo (VUA), Using virtual agents for simulation and training of social skills in safety-critical circumstances
 - 30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
 - 31 Milan Jelisavcic (VUA), Alive and Kicking: Baby Steps in Robotics
 - 32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
 - 33 Anil Yaman (TU/e), Evolution of Biologically Inspired Learning in Artificial Neural Networks
 - 34 Negar Ahmadi (TU/e), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
 - 35 Lisa Facey-Shaw (OU), Gamification with digital badges in learning programming

Appendix C. SIKS Dissertation Series

- 36 Kevin Ackermans (OU), Designing Video-Enhanced Rubrics to Master Complex Skills
 - 37 Jian Fang (TUD), Database Acceleration on FPGAs
 - 38 Akos Kadar (OU), Learning visually grounded and multilingual representations
-
- 2020 01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
 - 02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
 - 03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
 - 04 Maarten van Gompel (RUN), Context as Linguistic Bridges
 - 05 Yulong Pei (TU/e), On local and global structure mining
 - 06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
 - 07 Wim van der Vegt (OU), Towards a software architecture for reusable game components
 - 08 Ali Mirsoleimani (UL), Structured Parallel Programming for Monte Carlo Tree Search
 - 09 Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
 - 10 Alifah Syamsiyah (TU/e), In-database Preprocessing for Process Mining
 - 11 Sepideh Mesbah (TUD), Semantic-Enhanced Training Data Augmentation Methods for Long-Tail Entity Recognition Models
 - 12 Ward van Breda (VUA), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
 - 13 Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
 - 14 Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
 - 15 Konstantinos Georgiadis (OU), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
 - 16 Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling
 - 17 Daniele Di Mitri (OU), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
 - 18 Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
 - 19 Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
 - 20 Albert Hankel (VUA), Embedding Green ICT Maturity in Organisations
 - 21 Karine da Silva Miras de Araujo (VUA), Where is the robot?: Life as it could be
 - 22 Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar

-
- 23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
 - 24 Lenin da Nóbrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
 - 25 Xin Du (TU/e), The Uncertainty in Exceptional Model Mining
 - 26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
 - 27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
 - 28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
 - 29 Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
 - 30 Bob Zadok Blok (UL), Creatief, Creatiever, Creatiefst
 - 31 Gongjin Lan (VUA), Learning better – From Baby to Better
 - 32 Jason Rhuggenaath (TU/e), Revenue management in online markets: pricing and online advertising
 - 33 Rick Gilsing (TU/e), Supporting service-dominant business model evaluation in the context of business model innovation
 - 34 Anna Bon (UM), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
 - 35 Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production
-
- 2021 01 Francisco Xavier Dos Santos Fonseca (TUD), Location-based Games for Social Interaction in Public Space
 - 02 Rijk Mercuri (TUD), Simulating Human Routines: Integrating Social Practice Theory in Agent-Based Models
 - 03 Seyyed Hadi Hashemi (UvA), Modeling Users Interacting with Smart Devices
 - 04 Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
 - 05 Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems
 - 06 Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
 - 07 Armel Lefebvre (UU), Research data management for open science
 - 08 Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
 - 09 Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Children's Collaboration Through Play
 - 10 Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
 - 11 Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision

Appendix C. SIKS Dissertation Series

- 12 Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
 - 13 Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
 - 14 Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
 - 15 Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
 - 16 Esam A. H. Ghaleb (UM), Bimodal emotion recognition from audio-visual cues
 - 17 Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks
 - 18 Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
 - 19 Roberto Verdecchia (VUA), Architectural Technical Debt: Identification and Management
 - 20 Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems
 - 21 Pedro Thiago Timbó Holanda (CWI), Progressive Indexes
 - 22 Sihang Qiu (TUD), Conversational Crowdsourcing
 - 23 Hugo Manuel Proença (UL), Robust rules for prediction and description
 - 24 Kaijie Zhu (TU/e), On Efficient Temporal Subgraph Query Processing
 - 25 Eoin Martino Grua (VUA), The Future of E-Health is Mobile: Combining AI and Self-Adaptation to Create Adaptive E-Health Mobile Applications
 - 26 Benno Kruit (CWI/VUA), Reading the Grid: Extending Knowledge Bases from Human-readable Tables
 - 27 Jelte van Waterschoot (UT), Personalized and Personal Conversations: Designing Agents Who Want to Connect With You
 - 28 Christoph Selig (UL), Understanding the Heterogeneity of Corporate Entrepreneurship Programs
-
- 2022 01 Judith van Stegeren (UT), Flavor text generation for role-playing video games
 - 02 Paulo da Costa (TU/e), Data-driven Prognostics and Logistics Optimisation: A Deep Learning Journey
 - 03 Ali el Hassouni (VUA), A Model A Day Keeps The Doctor Away: Reinforcement Learning For Personalized Healthcare
 - 04 Ünal Aksu (UU), A Cross-Organizational Process Mining Framework
 - 05 Shiwei Liu (TU/e), Sparse Neural Network Training with In-Time Over-Parameterization
 - 06 Reza Refaei Afshar (TU/e), Machine Learning for Ad Publishers in Real Time Bidding

-
- 07 Sambit Praharaj (OU), Measuring the Unmeasurable? Towards Automatic Co-located Collaboration Analytics
 - 08 Maikel L. van Eck (TU/e), Process Mining for Smart Product Design
 - 09 Oana Andreea Inel (VUA), Understanding Events: A Diversity-driven Human-Machine Approach
 - 10 Felipe Moraes Gomes (TUD), Examining the Effectiveness of Collaborative Search Engines
 - 11 Mirjam de Haas (UT), Staying engaged in child-robot interaction, a quantitative approach to studying preschoolers' engagement with robots and tasks during second-language tutoring
 - 12 Guanyi Chen (UU), Computational Generation of Chinese Noun Phrases
 - 13 Xander Wilcke (VUA), Machine Learning on Multimodal Knowledge Graphs: Opportunities, Challenges, and Methods for Learning on Real-World Heterogeneous and Spatially-Oriented Knowledge
 - 14 Michiel Overeem (UU), Evolution of Low-Code Platforms
 - 15 Jelmer Jan Koorn (UU), Work in Process: Unearthing Meaning using Process Mining
 - 16 Pieter Gijsbers (TU/e), Systems for AutoML Research
 - 17 Laura van der Lubbe (VUA), Empowering vulnerable people with serious games and gamification
 - 18 Paris Mavromoustakos Blom (TiU), Player Affect Modelling and Video Game Personalisation
 - 19 Bilge Yigit Ozkan (UU), Cybersecurity Maturity Assessment and Standardisation
 - 20 Fakhra Jabeen (VUA), Dark Side of the Digital Media - Computational Analysis of Negative Human Behaviors on Social Media
 - 21 Seethu Mariyam Christopher (UM), Intelligent Toys for Physical and Cognitive Assessments
 - 22 Alexandra Sierra Rativa (TiU), Virtual Character Design and its potential to foster Empathy, Immersion, and Collaboration Skills in Video Games and Virtual Reality Simulations
 - 23 Ilir Kola (TUD), Enabling Social Situation Awareness in Support Agents
 - 24 Samaneh Heidari (UU), Agents with Social Norms and Values - A framework for agent based social simulations with social norms and personal values
 - 25 Anna L.D. Latour (UL), Optimal decision-making under constraints and uncertainty
 - 26 Anne Dirkson (UL), Knowledge Discovery from Patient Forums: Gaining novel medical insights from patient experiences
 - 27 Christos Athanasiadis (UM), Emotion-aware cross-modal domain adaptation in video sequences
 - 28 Onuralp Ulusoy (UU), Privacy in Collaborative Systems

Appendix C. SIKS Dissertation Series

- 29 Jan Kolkmeier (UT), From Head Transform to Mind Transplant: Social Interactions in Mixed Reality
 - 30 Dean De Leo (CWI), Analysis of Dynamic Graphs on Sparse Arrays
 - 31 Konstantinos Traganas (TU/e), Tackling Complexity in Smart Manufacturing with Advanced Manufacturing Process Management
 - 32 Cezara Pastrav (UU), Social simulation for socio-ecological systems
 - 33 Brinn Hekkelman (CWI/TUD), Fair Mechanisms for Smart Grid Congestion Management
 - 34 Nimat Ullah (VUA), Mind Your Behaviour: Computational Modelling of Emotion & Desire Regulation for Behaviour Change
 - 35 Mike E.U. Ligthart (VUA), Shaping the Child-Robot Relationship: Interaction Design Patterns for a Sustainable Interaction
-
- 2023 01 Bojan Simoski (VUA), Untangling the Puzzle of Digital Health Interventions
 - 02 Mariana Rachel Dias da Silva (TiU), Grounded or in flight? What our bodies can tell us about the whereabouts of our thoughts
 - 03 Shabnam Najafian (TUD), User Modeling for Privacy-preserving Explanations in Group Recommendations
 - 04 Gineke Wiggers (UL), The Relevance of Impact: bibliometric-enhanced legal information retrieval
 - 05 Anton Bouter (CWI), Optimal Mixing Evolutionary Algorithms for Large-Scale Real-Valued Optimization, Including Real-World Medical Applications
 - 06 António Pereira Barata (UL), Reliable and Fair Machine Learning for Risk Assessment
 - 07 Tianjin Huang (TU/e), The Roles of Adversarial Examples on Trustworthiness of Deep Learning
 - 08 Lu Yin (TU/e), Knowledge Elicitation using Psychometric Learning
 - 09 Xu Wang (VUA), Scientific Dataset Recommendation with Semantic Techniques
 - 10 Dennis J.N.J. Soemers (UM), Learning State-Action Features for General Game Playing
 - 11 Fawad Taj (VUA), Towards Motivating Machines: Computational Modeling of the Mechanism of Actions for Effective Digital Health Behavior Change Applications
 - 12 Tessel Bogaard (VUA), Using Metadata to Understand Search Behavior in Digital Libraries
 - 13 Injy Sarhan (UU), Open Information Extraction for Knowledge Representation
 - 14 Selma Čaušević (TUD), Energy resilience through self-organization
 - 15 Alvaro Henrique Chaim Correia (TU/e), Insights on Learning Tractable Probabilistic Graphical Models

-
- 16 Peter Blomsma (TiU), Building Embodied Conversational Agents: Observations on human nonverbal behaviour as a resource for the development of artificial characters
 - 17 Meike Nauta (UT), Explainable AI and Interpretable Computer Vision – From Oversight to Insight
 - 18 Gustavo Penha (TUD), Designing and Diagnosing Models for Conversational Search and Recommendation
 - 19 George Aalbers (TiU), Digital Traces of the Mind: Using Smartphones to Capture Signals of Well-Being in Individuals
 - 20 Arkadiy Dushatskiy (TUD), Expensive Optimization with Model-Based Evolutionary Algorithms applied to Medical Image Segmentation using Deep Learning
 - 21 Gerrit Jan de Bruin (UL), Network Analysis Methods for Smart Inspection in the Transport Domain
 - 22 Alireza Shojaifar (UU), Volitional Cybersecurity
 - 23 Theo Theunissen (UU), Documentation in Continuous Software Development
 - 24 Agathe Balayn (TUD), Practices Towards Hazardous Failure Diagnosis in Machine Learning
 - 25 Jurian Baas (UU), Entity Resolution on Historical Knowledge Graphs
 - 26 Loek Tonnaer (TU/e), Linearly Symmetry-Based Disentangled Representations and their Out-of-Distribution Behaviour
 - 27 Ghada Sokar (TU/e), Learning Continually Under Changing Data Distributions
 - 28 Floris den Hengst (VUA), Learning to Behave: Reinforcement Learning in Human Contexts
 - 29 Tim Draws (TUD), Understanding Viewpoint Biases in Web Search Results
-
- 2024 01 Daphne Miedema (TU/e), On Learning SQL: Disentangling concepts in data systems education
 - 02 Emile van Krieken (VUA), Optimisation in Neurosymbolic Learning Systems
 - 03 Feri Wijayanto (RUN), Automated Model Selection for Rasch and Mediation Analysis
 - 04 Mike Huisman (UL), Understanding Deep Meta-Learning
 - 05 Yiyong Gou (UM), Aerial Robotic Operations: Multi-environment Cooperative Inspection & Construction Crack Autonomous Repair
 - 06 Azqa Nadeem (TUD), Understanding Adversary Behavior via XAI: Leveraging Sequence Clustering to Extract Threat Intelligence
 - 07 Parisa Shayan (TiU), Modeling User Behavior in Learning Management Systems
 - 08 Xin Zhou (UvA), From Empowering to Motivating: Enhancing Policy Enforcement through Process Design and Incentive Implementation
 - 09 Giso Dal (UT), Probabilistic Inference Using Partitioned Bayesian Networks

Appendix C. SIKS Dissertation Series

- 10 Cristina-Iulia Bucur (VUA), Linkflows: Towards Genuine Semantic Publishing in Science
- 11 withdrawn
- 12 Peide Zhu (TUD), Towards Robust Automatic Question Generation For Learning
- 13 Enrico Liscio (TUD), Context-Specific Value Inference via Hybrid Intelligence
- 14 Larissa Capobianco Shimomura (TU/e), On Graph Generating Dependencies and their Applications in Data Profiling
- 15 Ting Liu (VUA), A Gut Feeling: Biomedical Knowledge Graphs for Interrelating the Gut Microbiome and Mental Health
- 16 Arthur Barbosa Câmara (TUD), Designing Search-as-Learning Systems
- 17 Razieh Alidoosti (VUA), Ethics-aware Software Architecture Design
- 18 Laurens Stoop (UU), Data Driven Understanding of Energy-Meteorological Variability and its Impact on Energy System Operations
- 19 Azadeh Mozafari Mehr (TU/e), Multi-perspective Conformance Checking: Identifying and Understanding Patterns of Anomalous Behavior
- 20 Ritsart Anne Plantenga (UL), Omgang met Regels
- 21 Federica Vinella (UU), Crowdsourcing User-Centered Teams
- 22 Zeynep Ozturk Yurt (TU/e), Beyond Routine: Extending BPM for Knowledge-Intensive Processes with Controllable Dynamic Contexts
- 23 Jie Luo (VUA), Lamarck's Revenge: Inheritance of Learned Traits Improves Robot Evolution
- 24 Nirmal Roy (TUD), Exploring the effects of interactive interfaces on user search behaviour
- 25 Alisa Rieger (TUD), Striving for Responsible Opinion Formation in Web Search on Debated Topics
- 26 Tim Gubner (CWI), Adaptively Generating Heterogeneous Execution Strategies using the VOILA Framework
- 27 Lincen Yang (UL), Information-theoretic Partition-based Models for Interpretable Machine Learning
- 28 Leon Helwerda (UL), Grip on Software: Understanding development progress of Scrum sprints and backlogs
- 29 David Wilson Romero Guzman (VUA), The Good, the Efficient and the Inductive Biases: Exploring Efficiency in Deep Learning Through the Use of Inductive Biases
- 30 Vijanti Ramautar (UU), Model-Driven Sustainability Accounting
- 31 Ziyu Li (TUD), On the Utility of Metadata to Optimize Machine Learning Workflows
- 32 Vinicius Stein Dani (UU), The Alpha and Omega of Process Mining
- 33 Siddharth Mehrotra (TUD), Designing for Appropriate Trust in Human-AI interaction
- 34 Robert Deckers (VUA), From Smallest Software Particle to System Specification