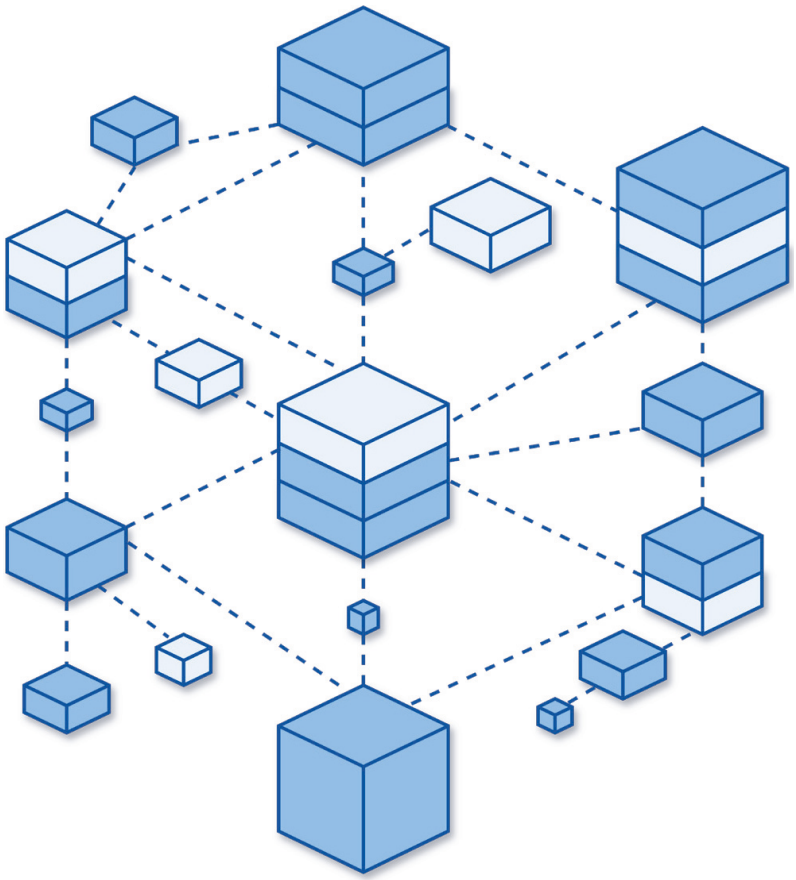
SE | RWTH AACHEN UNIVERSITY

Rohit Gupta

A Systematic Approach to Fostering Engineering of Industrial Domain-Specific Modelling Languages



Aachener Informatik-Berichte,
Software Engineering

Band 57

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

A Systematic Approach to Fostering Engineering of Industrial Domain-Specific Modelling Languages

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

M.Sc. TUM
Rohit Gupta
aus Kolkata, Indien

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. Matthias Tichy

Tag der mündlichen Prüfung: 10.04.2025

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

[Gup25] R. Gupta:
A Systematic Approach to Fostering Engineering of Industrial Domain-Specific Modelling Languages.
Aachener Informatik-Berichte, Software Engineering, Band 57,
ISBN 978-3-8440-9993-5, Shaker Verlag, Jun. 2025.

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 57

Rohit Gupta
RWTH Aachen University

A Systematic Approach to Fostering Engineering of Industrial Domain-Specific Modelling Languages

Shaker Verlag
Düren 2025

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Zugl.: D 82 (Diss. RWTH Aachen University, 2025)

Printed in Germany.

Print-ISBN	978-3-8440-9993-5
PDF-ISBN	978-3-8191-0076-5
ISSN	1869-9170
eISSN	2944-6910

Shaker Verlag GmbH • Am Langen Graben 15a • 52353 Düren
Phone: 0049/2421/99011-0 • Telefax: 0049/2421/99011-9
Internet: www.shaker.de • e-mail: info@shaker.de

Dedicated to my parents

Eidesstattliche Erklärung
Declaration of Authorship

I, Rohit Gupta

declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

Hiermit erkläre ich an Eides statt / I do solemnely swear that:

1. This work was done wholly or mainly while in candidature for the doctoral degree at this faculty and university;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others or myself, this is always clearly attributed;
4. Where I have quoted from the work of others or myself, the source is always given. This thesis is entirely my own work, with the exception of such quotations;
5. I have acknowledged all major sources of assistance;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published before as: [GKR$^+$21], [GJRR22a], [GJRR22b], [BGJ$^+$23], [GJRR23], [GBJ$^+$24].

Aachen, 10.04.2025

Rohit Gupta

Abstract

Domain-Specific Modelling Languages (DSMLs) help modellers and domain experts in various domains such as healthcare, energy, information technology, and so on, in reducing the gap between the problem space and the solution space by placing models at the centre of development activities. This shift towards model-driven development (MDD), where models are introduced at early stages in any software or systems engineering process and are the primary software engineering artefacts, allows modellers to design complex, heterogeneous real-world abstractions of their systems. In such complex systems, concepts from individual domains are often integrated as part of the bigger language infrastructure. Specifically in an industrial setting, the methodologies to describe a systematic engineering process for developing such complex yet modular and reusable DSMLs that provides a seamless modelling experience to modellers in both the large scale organisations as well as in small and medium enterprises is still largely neglected.

Accordingly, this thesis is aimed at providing the means to engineer graphical DSMLs that are specifically focussed on industrial contexts. Based on existing approaches, this work presents a systematic approach to fostering the engineering of industrial DSMLs by composing reusable language infrastructure parts without the need for creating completely new language infrastructure for similar domains every time. These reusable units of a DSML, termed *DSML building blocks*, consist of reusable language components that, entirely or in part, contributes to the technical definition of the language itself. The language components compose through different forms of language composition to form heterogeneous, integrated DSMLs. To foster the interoperability of such common language infrastructure parts between modelling environments, a bidirectional exchange mechanism is detailed in this work. This work further provides guidelines and design decisions that language engineers should consider for their language infrastructure in order to elevate the overall experience of modellers. An approach for integrating methods, techniques, and concepts in terms of guidance and recommendations for modellers is detailed in this thesis that aims to move away from plain technical views of models to instead model-aware and dynamic views that are focussed on the current modelling situation of such modellers.

Overall, this thesis presents approaches to modularly build reusable units of DSMLs that compose together. The approaches presented in this thesis allows language engineers to provide a more complete and integrated language infrastructure that is ultimately aimed at improving the modelling experience of practitioners in the industry.

Kurzfassung

Domänenspezifische Modellierungssprachen (DSMLs) hilft Modellierern und Domänenexperten in verschiedenen Bereichen wie Gesundheitswesen, Energie, Informationstechnologie usw. dabei, die Lücke zwischen dem Problemraum und dem Lösungsraum zu verringern, indem Modelle in den Mittelpunkt der Entwicklungsaktivitäten gestellt werden. Diese Verlagerung hin zur modellgesteuerten Entwicklung (MDD), bei der Modelle bereits in frühen Phasen eines Software- oder Systementwicklungsprozesses eingeführt werden und die primären Artefakte der Softwareentwicklung darstellen, ermöglicht es den Modellierern, komplexe, heterogene reale Abstraktionen solcher Systeme zu entwerfen. In solchen komplexen Systemen werden häufig Konzepte aus einzelnen Bereichen als Teil einer größeren Sprachinfrastruktur integriert. Speziell im industriellen Umfeld ist die Methodik zur Beschreibung eines systematischen Entwicklungsprozesses für solche komplexen und dennoch modularen und wiederverwendbaren DSMLs, die Modellierern sowohl in großen Organisationen als auch in kleinen und mittleren Unternehmen eine nahtlose Modellierungserfahrung bietet, noch weitgehend vernachlässigt.

Dementsprechend zielt diese Arbeit darauf ab, die Mittel zur Entwicklung grafischer DSMLs bereitzustellen, die speziell auf industrielle Kontexte ausgerichtet sind. Basierend auf bestehenden Ansätzen wird in dieser Arbeit ein systematischer Ansatz vorgestellt, der die Entwicklung industrieller DSMLs fördert, indem wiederverwendbare Sprachinfrastrukturteile zusammengestellt werden, ohne dass jedes Mal eine komplett neue Sprachinfrastruktur für ähnliche Domänen erstellt werden muss. Diese wiederverwendbaren Einheiten einer DSML, die als *DSML-Bausteine* bezeichnet werden, bestehen aus wiederverwendbaren Sprachkomponenten, die ganz oder teilweise zur technischen Definition der Sprache selbst beitragen. Die Sprachkomponenten lassen sich durch verschiedene Formen der Sprachkomposition zu einer heterogenen, integrierten DSMLs zusammensetzen. Um die Interoperabilität solcher gemeinsamen Sprachinfrastrukturteile zwischen Modellierungsumgebungen zu fördern, wird in dieser Arbeit ein bidirektionaler Austauschmechanismus beschrieben. Diese Arbeit bietet außerdem Richtlinien und Designentscheidungen, die Sprachingenieure für ihre Sprachinfrastruktur in Betracht ziehen sollten, um die Gesamterfahrung von Modellierern zu verbessern. In dieser Arbeit wird ein Ansatz zur Integration von Methoden, Techniken, und Konzepten in Form von Leitlinien und Empfehlungen für Modellierer vorgestellt, der darauf abzielt, von einer rein technischen Sichtweise auf Modelle zu einer modellbewussten und dynamischen Sichtweise überzugehen, die die aktuelle Modellierungssituation der Modellierer berücksichtigt.

Insgesamt werden in dieser Arbeit Ansätze zum modularen Aufbau wiederverwendbarer Einheiten von DSMLs vorgestellt, die sich zusammensetzen lassen. Die in dieser Arbeit vorgestellten Ansätze ermöglichen es Sprachingenieuren, eine vollständigere und integrierte Sprachinfrastruktur bereitzustellen, die letztlich darauf abzielt, die Modellierungserfahrung von Praktikern in der Industrie zu verbessern.

Acknowledgements

I would like to extend my heartfelt appreciation to my thesis advisor, Prof. Dr. Bernhard Rumpe, for his invaluable guidance, support, and motivation throughout the entire research journey. His expertise, insightful feedback, and unwavering commitment has played a critical role in shaping this thesis and enhancing my academic growth. I feel incredibly fortunate to have had the opportunity to work under his mentorship.

I would also like to express my gratitude to Prof. Dr. Matthias Tichy for taking on the role of the second reviewer, Prof. Dr. Stefan Decker for willing to take my theoretical oral examination, and Prof. Dr. Sander Leemans for completing the thesis committee.

To my colleagues at the Software Engineering chair, I am grateful to your guidance and inspiration throughout this academic journey. Thank you Dr. Arvid Butting, Nico Jansen, Dr. Judith Michael, Dr. Lukas Netz, Jérôme Pfeiffer, David Schmalzing, and Prof. Dr. rer. nat. Andreas Wortmann for your continuous support. I would also like to thank Florian Drux, Alexander Hellwig, Nico, Florian Rademacher, David Schmalzing, Marc Schmidt, Max Stachon, and Sebastian Stüber for reading and reviewing various parts of this work to help improve my thesis. A special thanks to Sonja Müßigbrodt and Sylvia Gunder for the administrative assistance during the globally challenging times.

This work would not have been possible without the support of my industrial advisor Nikolaus Regnat at Siemens Technology. Your encouragement, motivation, and invaluable discussions has supported me to grow immensely both professionally and personally, and encouraged me to think of solutions (and problems) that I would not have come up with. To Sieglinde Kranz, Prof. Dr. Andreas Biesdorf, Dr. Ambra Cala, Nicole Wengatz, and Carolin Rubner, thank you for providing me with the platform to grow within Siemens. A warm thanks to my colleagues at Siemens especially Christoph, Daniel, Deepak, Gowri, Manoj, Michael, Mirjam, Nicole, Preeti, Shady, and many others for making everyday a great place to work. Thank you also to my Siemens Healthineers colleagues for the industrial use cases that helped shape some examples in this thesis.

Lastly, I would like to express my deepest gratitude to my friends and family for their unshakeable love, support, and understanding. Their belief in my abilities, encouragement, and sacrifices have been the foundation of my success. I am forever grateful for their presence in my life. To my parents, Shobha and Ramesh, you have been the pillars of support in my life and always pushed me to chase my ambitions. To my brother Saurabh and my sister-in-law Vishakha, your unwavering support and wisdom has always been a source of inspiration for me. Congratulations on becoming proud parents of twins. To my adorable toddler nephews, Samrat and Virat, your pure presence has been a reminder of the beauty and preciousness of life. Thank you all for being a part of this incredible journey.

Aachen, April 2025
Rohit Gupta

Contents

Chapter 1

Introduction

As systems grow more complex, heterogeneous, and interlinked in nature, so do challenges in the efficient engineering of such systems. Consequently, there exists a substantial conceptual gap [FR07] in the systems engineering domains between the expertise of domain experts, such as biologists, chemists, mechanical engineers, medical assistants, and so on, and the challenges posed by systems engineering. To address these challenges, advancements have been made in the systems engineering domains to move from traditional documents such as Microsoft Word or Excel to models, by introducing modelling at the early, but crucial, stages in the systems engineering processes [Fow10, HMR+19]. Often, software engineers are skilled and knowledgeable in ubiquitous General Purpose Languages (GPLs) [BGM10], such as Java or C++, which are primarily used for software development. On the other hand, modellers or domain experts are rarely software engineers and often do not model on a daily basis. This presents particular challenges in modelling systems and their domains [PB20] through such GPLs. As such, GPLs focus strongly on the technical implementation details, and rarely consider solutions that directly impact only a particular domain. To this end, model-driven development (MDD) techniques [Sel03] are deployed to foster the comprehensibility of systems using models that help practitioners and researchers alike understand complex problems and their solutions through real-world abstractions.

Domain-Specific Languages (DSLs) aim to reduce the gap in a particular domain by supporting domain-specific abstractions, that are better used to understand, analyse, and synthesize systems and their parts [Fow10]. Domain-Specific Modelling Languages (DSMLs), on the other hand, supports both modellers and domain experts in solving similar challenges in MDD by providing the necessary language infrastructure needed to solve a problem in a particular domain through models in the textual, graphical, or projectional technological spaces. Models, when solely represented as documentation, are of little value as there should be meaning attached to them and the domains they represent. Naturally, as DSLs are rather restricted in their syntax, their complexity is also relatively lower than that of GPLs. Therefore, DSMLs provide the relevant concepts for realising a technical language targeted for a specific domain and modellers do not need to worry about building their own technical terms. Although DSMLs in research and industry are based on similar engineering foundations, the focus of this thesis is

primarily in the industrial engineering of DSMLs [FRR09], specifically in the graphical space. In a comprehensive book detailing domain-specific modelling, [KT08] describe the overall development and design of models using DSMLs that focus mainly on the technical aspects such as code generation and meta-modelling. However, development methodologies in industrial projects must provide a flexible engineering process that caters to both the large scale organisations as well as small and medium enterprises without a lot of investment and with support for a better modelling experience for DSMLs practitioners. This thesis therefore describes a systematic approach to the engineering of industrial DSMLs by combining approaches from software language engineering (SLE) and developing reusable and modular language infrastructure that eliminates the need to build DSMLs with similar domain concepts every time from scratch.

1.1 Motivation

DSLs and DSMLs, just like any other software languages, are also regarded as software [FGLP10] and therefore subject to the usual challenges of SLE [HRW18]. Specialised tools such as language workbenches or modelling tools have been created over the years to advance support in language engineering for both the academia as well as the industry [EvdSV+13]. Until now, systematic SLE methodologies are rare, tied to individual departments, and building DSMLs is resource-intensive, often with lack of good usability, proper guidance, and support for DSML users [MGD+16]. Further, it is also assumed that language engineering should focus solely on providing the technical language. In spite of these previous SLE efforts [KKP+09, CMP20], the engineering of industrial DSMLs requires a more complete language infrastructure, that is focussed on combining the technical aspects of a language, with good usability, methods, and concepts to help modellers reach their goals.

Building graphical DSMLs is challenging as it requires the integration of a complete graphical syntax and often parts of textual syntax directly within a graphical modelling tool. The graphical concrete syntax is meant to visually represent the constructs of a domain in a way its abstractions appear in the real world. Therefore, any kind of drawing or sketches that are made by hand, should be resembled evenly in a DSML, such as with images or boxes [LJJ07]. Although previous literature exists on various graphical modelling tools [Ent23, LNH06, Met, HHFN13] and their applications, the remainder of this thesis explores the concepts for systematically engineering graphical industrial DSMLs in MagicDraw [Mag20] and also briefly considers language development aspects in the textual space, with the MontiCore ecosystem [HKR21].

DSMLs are modularised [DCB+15, CKM+18] into reusable and composable units, referred to as *language components*, that defines the (incomplete) language. These reusable units allow a language definition to be independent of a modelling environment. The composition of DSMLs are primarily achieved using grammars in the textual space, or

using language configurations in the graphical space. Technically, language components are comprised of all the software artefacts needed for describing the language, including the syntax, the well-formedness rules, and any other code generation techniques. To foster reusability of DSMLs and its parts, language components are composed to build novel languages or versions of languages, without a strict coupling between the composed language and the base languages. Furthermore, this thesis also details mechanisms of exchanging language components and DSMLs constructs between multiple modelling environments as a step towards fostering interoperability of domain-specific constructs across different technological spaces.

A key benefit in equipping users with the necessary language infrastructure required for them to achieve their modelling goals improves the modelling experience of such users. In this regard, defining a good user experience (UX) for a DSML is crucial in positively impacting the feelings and impressions of a modeller. While definitions of UX have been proposed in the literature [Has08], they are rather too generic and apply commonly across the software and systems development. A good UX must consider both the DSML and its accompanying modelling environment, i.e., the language workbench or the modelling tool. Design decisions including those that help improve the general usability of a modelling environment must be actively considered during the development of a language infrastructure, which is explored in this thesis.

The integration of solid methods, techniques, and concepts within a modelling environment that elevates the UX of modellers is still missing, partly due to project and resource constraints. To address this challenge, active and synchronised information pertaining to the current models must be made available to the users. A way to achieve this is using method engineering [Bri96] or a kind of recommender system [AT05] that analyses the designed models and their properties, and retrieve a list of recommended methodical steps or information from an external source, such as a database, to provide dynamically changing suggestions to users. Techniques described in this thesis aim to move away from providing a plain technical view or a rather static, and often outdated, source of information, and instead provide guidance and support to users by describing a set of recommendations, tasks, activities, or processes during modelling that are rarely considered during the development of a language infrastructure.

It is therefore essential to integrate these aspects of industrial DSML engineering into reusable units that encapsulate the complete language infrastructure, i.e., the language definition, methods, and concepts related to a domain or a set of domains. These reusable units, termed *DSML building blocks* in this thesis, are aimed at providing effective modelling techniques to DSML practitioners. This thesis therefore describes, with industrial examples and case studies, a systematic approach to engineering industrial DSMLs in the large by composing reusable language infrastructure parts with the modelling tool MagicDraw [Mag20]. Different forms of language composition are described that reuse language components and compose novel languages without the need of generating a completely new language infrastructure for similar domains every time. Moreover, this

thesis explores the interoperability of such language infrastructure across different modelling environments, provides UX and usability guidelines for improving the modelling experience of DSML users, as well as provides a framework for integrating active model-aware recommendations that is directly accessible to practitioners during modelling.

1.2 Research Questions and Objectives

Overall, this thesis contributes to answering questions on engineering DSMLs in the industrial context. As discussed above, related work does not sufficiently address aspects of engineering such industrial DSMLs in terms of reusing parts of a language or providing a more integrated and holistic modelling experience for practitioners. While this thesis cannot possibly answer all unanswered questions on the engineering of industrial DSMLs, it does provide a systematic approach in fostering the engineering of such DSMLs that leads to a wider adoption in industrial contexts. To this end, the following is the main research question of this thesis:

> *How can domain-specific modelling languages be composed using reusable language units and how can language engineers foster the efficient engineering of such languages in industrial contexts?*

To answer this main research question, this thesis explores possible answers to the following six partial research questions:

RQ1: *How can language engineers systematically develop reusable building blocks for a language?*
To effectively engineer industrial DSMLs, parts of the language infrastructure or the language itself must be reusable and modular in nature. Such a reusable unit that potentially provides the complete language infrastructure of a particular aspect of a domain, i.e., the language components, methods, and concepts, is termed a *DSML building block*. In practice, industrial DSMLs constitute the representation of notions from multiple domains, often with intersecting concepts, raising the need for multiple DSML building blocks. For example, defining requirements for a project is a common concept that is reused almost identically in a variety of domains such as medical, energy, information technology, and so on. An answer to this research question should propose a systematic approach to developing such reusable units that are modular in nature and are eventually composed into larger languages without the need for developing similar notions of domains from scratch every time.

RQ2: *What constitutes a reusable language component?*
Software languages are typically realised through software, meaning the artefacts of a

language encapsulates the (incomplete) definition of the language itself. To foster the reusability of such artefacts, a software should be decomposed into smaller language components [EGR12]. This research question on language components has been explored previously in the textual space [But23], particularly in the MontiCore language workbench [HKR21]. This thesis primarily explores the concept of language components in the graphical space, using the MagicDraw modelling tool [Mag20]. An answer to this research question should identify the requirements and properties of a language component, and provide the definitions and concepts needed to realise language components in the graphical space. Further, unified and mutual notions of language components, valid across different technological spaces, must also be answered by this research question.

RQ3: *What different forms of language composition can be applied to foster the better development of heterogeneous DSMLs?*
A DSML can be modularised into composable units called language components as mentioned in the previous research question **RQ2**. However, in order to compose complete and heterogeneous DSMLs, suitable forms of language composition must exist. An answer to this research question should provide the notions for different forms of language composition, such as language inheritance, extension, embedding, and aggregation primarily in the graphical technological space.

RQ4: *How can we achieve the interoperability of DSML constructs between multiple modelling environments?*
In order to foster the interoperability of domain-specific constructs across modelling tools or environments, it is important to define language components, consisting of software artefacts, in a way that it promotes a bidirectional exchange mechanism between such modelling environments. An answer to this research question should provide a method in which language components, their characteristics, and properties are extracted using custom GPL code, such as in Java, and reused as-is or with minimal adjustments in another modelling environment.

RQ5: *How can language engineers develop better DSMLs that improves the overall modelling experience for users?*
A DSML should elevate the UX for practitioners by providing domain abstracts that are as close as possible to its real world abstractions. In an industrial setting, practitioners often face challenges in terms of usability of DSMLs. This is partly due to the inconsideration of language engineers for providing good design aspects within the DSMLs that effectively represents the domain concepts, and partly because of the limitations in the functionalities offered by modelling tools. An answer to this research question should provide effective design guidelines and design decisions that language engineers must consider that complements the technical definition of a language and its components to help improve the overall modelling experience for DSML practitioners.

RQ6: *How can we establish a modelling methodology for providing integrated recommendations and guidance to modellers that considers their active modelling situation?*
To further enhance the UX of practitioners, methods that actively assist practitioners in their modelling must be more tightly integrated as part of defining a more complete language infrastructure. Such methods must move away from providing only static sources of information, and rather provide training materials, active recommendations based on ongoing modelling work, and other processes that eliminate the need to tediously search through endless pages of DSML documentation, with the guidance provided directly on the modelling environment itself. An answer to this research question should detail an approach for developing a more tightly integrated guidance infrastructure with a DSML and the accompanying modelling environment, which actively provides recommendations and methods for practitioners that are synchronised with their currently designed models.

1.3 Main Contributions and Thesis Organisation

The main contributions of this thesis are:

- a systematic approach to engineering reusable DSMLs by defining *DSML building blocks* for industrial contexts that help language engineers develop modular DSMLs and its parts;
- describing *language components* and its constituents in the graphical modelling space and defining mutual notions of language components that are valid across the textual and graphical technological spaces of MontiCore [HKR21] and MagicDraw [Mag20];
- describing different forms of language composition for composing heterogeneous DSMLs in the graphical modelling space;
- a tool mechanism to exchange DSML constructs across the modelling tools of MagicDraw and Enterprise Architect [Ent23] for fostering interoperability and exchange of DSMLs and its parts;
- defining a (non-exhaustive) set of design decisions and design guidelines for language engineers to better engineer industrial DSMLs and their constructs in terms of their real world abstractions, thereby improving a practitioner's modelling experience;
- a customisable recommendation tool in MagicDraw that provides active, synchronised, and model-aware recommendations to users for a more holistic and complete language infrastructure; and

- multiple case studies that evaluate the systematic approach to engineering industrial DSMLs using such DSML building blocks.

The thesis is structured in a way that it follows a top-down approach: it first explains a systematic engineering process for developing reusable industrial DSMLs and its parts, and then takes a look at the various research questions starting from the technical definition of language components, moving towards the interoperability of DSML parts, and finally integrating aspects of UX for a more complete language infrastructure. Each chapter contributes to the main research question and discusses topics that are considered related work. The chapters are structured as follows:

Chapter 2 presents the foundations and the background knowledge necessary to understand the concepts described in this thesis. The foundations comprise topics such as an overview of SLE, DSML engineering, introduction of the MagicDraw and MontiCore ecosystems, and different domain models that are frequented throughout the examples presented in individual chapters.

Chapter 3 presents an overall developmental approach to the systematic engineering of industrial DSMLs using the concept of reusable DSML building blocks in MagicDraw. This chapter also discusses the roles in industrial DSML engineering and combines the individual approaches presented later in the thesis.

Chapter 4 describes the concepts of language components in the graphical technological space, discusses how different forms of language composition are achieved in both MagicDraw and MontiCore, and provides mutual notions of composing languages in both the textual and graphical technological spaces.

Chapter 5 details a mechanism for exchanging DSML constructs between different modelling environments to foster the interoperability of common language components.

Chapter 6 describes a set of UX guidelines and design decisions that language engineers must consider while defining DSML building blocks to ultimately improve the modelling experience of industrial DSML practitioners.

Chapter 7 explains a methodology for generating model-aware recommendations and guidance to DSML practitioners by using the UX guidelines described in Chapter 6 for developing a tightly integrated language infrastructure.

Chapter 8 takes a look at three individual case studies in different domains based on the concepts described in the previous chapters.

Chapter 9 concludes this thesis by providing a summary of the results to the research questions as well as an outlook for further work.

1.4 List of Publications

Some of the answers to the research questions stated in this thesis have already been published in various forms before or are currently in press or preparation. Accordingly, some of the results of this thesis including figures, listing, data, and other relevant content are part of those publications. The following provides an overview of the publications and the contribution of the respective authors. If not stated otherwise, the author of this thesis is the main author of the following mentioned papers:

- [GKR+21] presents a systematic approach towards engineering industrial DSLs using modular, reusable DSL building blocks. The research problem and the concept was developed by the author of this thesis in detailed discussion with Nikolaus Regnat and Sieglinde Kranz. The background and motivation for the research problem was detailed by Andreas Wortmann and the author of this thesis. The conceptualisation and implementation of the DSL building blocks was defined in discussions between Bernhard Rumpe and the author of this thesis. The evaluation of this study was conducted with a focus group of modelling practitioners and researchers both from the industry and academia. Ambra Calá and Jérôme Pfeiffer provided additional feedback during the course of this study.
- [GJRR22b] showcases the construction of a comprehensive methodical language workbench by integrating key aspects of a modelling language, the Software Platform Embedded Systems (SPES) [BBK+21] methodology, and the modelling tool MagicDraw. The research problem and the concepts presented in this paper were developed by the author of this thesis along with Nikolaus Regnat and Bernhard Rumpe. The background, motivation, and the overview of the methodology was detailed by the author of this thesis. The details of the implementation of the SpesML workbench were illustrated by Nikolaus Regnat and the author of this thesis, while Nico Jansen provided the implementation details for embedding textual languages into MagicDraw.
- [GJRR22a] describes the categorisation of key UX aspects, including guidelines and design decisions, that language engineers must consider during industrial DSML development. The research problem and concepts presented in this paper were developed by the author of this thesis along with Nikolaus Regnat. The motivation, background, and the methodology presented in this paper was developed by the author of this thesis and Nico Jansen. The definitions of UX and design aspects as well as the case study were developed in detailed discussions by the the author of this thesis, Nikolaus Regnat, and Bernhard Rumpe.

- [BGJ+23] presents individual definitions of language components and the different forms of language composition in MontiCore and MagicDraw, and describes mutual notions for developing versions or families of DSMLs valid in both the textual and the graphical technological spaces. The research problem and concepts presented in this paper were developed by Arvid Butting, the author of this thesis, and Nico Jansen. The parts of the paper that detailed the engineering of software languages in MontiCore, describing example DSMLs in the MontiCore language workbench, and detailing the method and implementation of language components and their composition in MontiCore were developed by Arvid Butting and Nico Jansen. Parts of this paper also refer to the work of Arvid Butting in his thesis [But23]. The other parts of the paper that detail a similar engineering process in MagicDraw were developed by the author of this thesis and Nikolaus Regnat. The unified concepts of language components were mainly developed by the author of this thesis, Nico Jansen, and Bernhard Rumpe.

- [GBJ+24] presents an exchange mechanism between the Enterprise Architect and MagicDraw modelling tools, that ultimately allows exchanging individually created DSMLs and their constructs for promoting DSML interoperability. The research problem, background, methodology, and case study presented in this paper was developed by the author of this thesis, Christoph Binder, Nico Jansen, and David Schmalzing. The specification of engineering DSMLs in Enterprise Architect was provided by Christoph Binder, whereas the specification of engineering DSMLs in MagicDraw was provided by the author of this thesis and Nikolaus Regnat. In general, constant feedback on the concepts and implementation of the exchange mechanism were provided by the other authors.

- [GJRR23] presents a guidance infrastructure that provides methods, techniques, and recommendations to DSML practitioners that is dependent on their current modelling situation. The research problem and the concepts presented in this paper were developed by all the authors. The methodology and its implementation was carried out by the author of this thesis with detailed feedback from Nikolaus Regnat and Bernhard Rumpe. The case study and evaluation presented in this paper is a result of the collaborative efforts of the author of this thesis, Nikolaus Regnat, and domain experts from Siemens Healthineers. Certain parts of the overall architecture and threats to this study were discussed in detail by the author of this thesis and Nico Jansen.

Chapter 2

Foundations

This chapter describes the terms, concepts, tools, and language definitions which form the basis of this thesis. Concepts that describe SLE and DSMLs are described in Section 2.1. The foundations for graphical DSMLs, the choice of tool primarily used in this thesis, language composition basics, interoperability concepts, improving general usability, and active support methods for users are laid out in Section 2.2. Notions of feature models and function models that provide the basis of certain examples and case studies described in this thesis are introduced in Section 2.3.

2.1 Software Language Engineering

Software languages descriptions are also considered pieces of software, and therefore, must be treated in the same way as software [FGLP10]. They are subjected to the usual challenges of software engineering along with defining a language's constituents [GKR+21]. Therefore, SLE is considered a discipline within the world of software engineering that describes the systematic design, realisation, deployment, and evolution of software languages [Kle08]. To this end, software languages are considered languages that are readable both by engineers as well as those that are easily processed by machines [But23]. While GPLs are used to solve problems in any domain and are commonly based on programming languages such as Java or C, DSLs [Fow10] solve problems particular to a domain. Parts of the Unified Modelling Language (UML) are also used for programming [Rum16], but since it focusses on defining a modelling language, it is rather considered a general-purpose modelling language.

A DSL is a language that is capable of describing a particular system and its parts either entirely or from a particular viewpoint [Kle08]. A DSL, therefore, aims to reduce the gaps in a particular domain by supporting abstractions at a domain-specific level [CBCR15, GKR+21], meaning DSLs and their parts are better analysed and synthesised. The gap from the problem space to the implementation space is thus addressed by a DSL [FR07]. As they do not consider other aspects from other viewpoints or domains, DSLs are often rather restricted in their syntax and their complexity is relatively lower than that of GPLs. Some common examples of DSLs found in everyday software use are hypertext markup language (HTML), cascading style sheets (CSS), structured query

language (SQL), and so on, which aim to solve only a specific problem. For example, an SQL query cannot be used to design the user interface (UI) design of a webpage, as it is used only for querying database management systems.

DSMLs, on the other hand, are a specialised form of DSLs, which are important in the field of MDD. Here, each model is designed in a specific language with the possibility of transforming this model into another model that could be valid in another modelling language. Models of a DSML live only within the same domain environment, which means the models are generally tied to a specific implementation of a domain. As such, DSMLs are also subject to maintenance and evolution since they are primarily software languages. The technological spaces for DSMLs are almost always heterogeneous and can be categorised as either textual, graphical, or projectional [DCB+15, Bet16, Tol06]. This thesis describes concepts pertaining to DSMLs for improving the overall DSML engineering process in the graphical technological space. Although references to textual languages are made in this thesis, the use of SLE in the context of graphical DSMLs means the term language is synonymously used with a graphical DSML unless otherwise stated.

Definition of a Software Language.

Homo sapiens have designed elaborate and intricate languages for communication. While such a "set of all linguistic utterances" is described for defining a language [Kle08], MDD considers the "utterances" of models for describing a language or its parts. To this end, several definitions of software languages have been proposed in the literature [HR04, CBCR15], that concern with defining the syntax and semantics of a language. In the remainder of this thesis, we use the following definition of a software language [HR04, CGR09, CBCR15] that also applies to DSMLs:

Definition 1 (Software language)**.** *A software language definition consists of:*

- *(1) an abstract syntax that contains the essential information and describes the structure of a model, e.g., in the form of context-free grammars or class diagrams [HKR21];*
- *(2) a concrete syntax that is used to describe the concrete representation of the models, e.g., graphical [DCB+15], textual [Bet16], or projectional [Cam14];*
- *(3) semantics, in the sense of meaning [HR04]; and*
- *(4) context conditions to check the well-formedness of the language.*

In the remainder of the thesis, we refer to the definition of DSMLs in the graphical technological space unless explicitly stated. This means that the concrete syntax of a DSML is realised as a graphical concrete syntax, such as in a tabular, box-and-line,

or tree-based format [CFJ+16]. To this end, graphical modelling editors or tools are required to design graphical DSMLs and it is natural that such DSMLs have found their way into industrial applications.

Applying systematic SLE in a variety of domains is still challenging. While studies describe the implementation of industrial DSLs [MAGD+16, TK05], often these methodologies are restricted to specific departments within organisations, and require rebuilding languages from scratch which is rather time-consuming. A way to solve this challenge is to introduce a central research unit within an organisation that trains language engineers to develop domain-specific concepts with sufficient skills in SLE. The language engineering and reuse methods have been described to technically improve the languages with the introduction of explicit language interfaces [BPRW20], language merging techniques [DCB+15], or types of languages [SJ07]. However, considerations to improve the usability of graphical industrial DSMLs in terms of composition, interoperability, user experience, and guidance methods for both language engineers and users are still missing and are the key topics addressed in the remainder of this thesis.

2.2 Domain-Specific Modelling Languages

This section details the foundations of graphical industrial DSMLs that are necessary for understanding the concepts presented in this thesis (Section 2.2.1). In particular, the basic features of the modelling ecosystems of MagicDraw (Section 2.2.2) and MontiCore (Section 2.2.3) are introduced before describing the properties and forms of language composition in the textual and graphical technological spaces (Section 2.2.4). This section also explains the foundations of DSML interoperability (Section 2.2.5) between modelling tools and introduces various methods (Section 2.2.7) used to improve the modelling experience of graphical DSML users (Section 2.2.6).

2.2.1 Graphical Industrial DSMLs

The concrete syntax of graphical DSMLs is graphical, meaning certain kinds of graphical notations are used to visualise the various constructs of a DSML. This concrete syntax is defined in a way that it uses symbols to represent human-centric notions of a domain and are abstractions of the symbols commonly found in the real world. Graphical DSMLs can be thought of in a way that they resemble drawings, sketches, or annotations that are built by hand for a specific domain-specific scenario. Various types of icons, colours, customised views, and additional graphical user interfaces (GUIs) are considered part of representing a graphical concrete syntax. Often graphical notations are combined with textual notations to offer a more detailed outlook on the DSML constructs [LFGGdL16], meaning possibilities to transform textual metamodels to a suitable graphical format is also possible.

MDD is increasingly being used in industrial projects [FRR09, WG09, WBCW20] and is continuously evolving in research [FR07]. At the same time, introducing newer development paradigms and technologies for MDD or model-based systems engineering (MBSE) is difficult to adopt for stakeholders in an industrial domain [Sta09]. This is because either large-scale organisations need stable and incremental development methodologies, or small and medium enterprises cannot spend huge amounts of money to realise a rather simpler domain concept using commercial modelling tools. To achieve a measurable improvement in a DSML's productivity and quality, industrial projects must be flexible for adapting the processes of an organisation. This means that as more concepts are added to a DSML, language engineers must ensure that such concepts are not only captured in the definition of a DSML but also the DSMLs and their infrastructures evolve to support a good modelling experience.

There exists a number of graphical modelling tools such as Enterprise Architect [Ent23], Rational Rhapsody [IBM23], MetaEdit [Met], MagicDraw [Mag20], Modelio [Mod23c], and so on. Most tools support building languages that are based on UML, support model-driven architecture (MDA) generation templates, and produce similar templates for fostering the exchange of UML constructs. This thesis explores MagicDraw as the choice of modelling tool for answering the research questions, as it provides possibilities to enhance its existing functionalities, thereby allowing comprehensive extension opportunities for bringing the various parts of a language, its methods, and the tool together in order to elevate a user's experience of designing models using a DSML.

2.2.2 The MagicDraw Ecosystem

A significant portion of the research presented in this thesis has been implemented in the MagicDraw modelling tool. MagicDraw is a proprietary visual modelling tool based on UML and Systems Modelling Language (SysML) belonging to the mid-sized software company CATIA NoMagic, Inc. which was acquired by Dassault Systemes in 2018 [Mag20]. MagicDraw helps business and software analysts, programmers, modellers, and quality assurance engineers to easily deploy a software development life cycle (SDLC), and the in-built Open API support allows greater extensibility and flexibility in MDD solutions.

MagicDraw provides a broad range of customisation possibilities that capture most, if not all, challenges related to DSML engineering. These customisations allow the creation of language profiles based on UML that detail different language components and their artefacts. One such example of an artefact is a language element, which is also referred to as a *stereotype* in MagicDraw. Each of these stereotypes are configured with additional properties in the form of a *customisation* element that allows defining specific properties and rules for a given stereotype. MagicDraw also provides options to design GUIs using a Java-based plugin mechanism that eventually assists users in the integration of automation and the creation of enhanced functionalities that are not offered by default.

Further, various kinds of model templates and additional configurations are made to the DSMLs to ensure that users of such DSMLs easily adopt and work with the modelling language constructs.

DSMLs have been built in MagicDraw for a wide variety of domains. As part of this thesis, a large number of projects were either built or studied for research with MagicDraw. Projects in the healthcare domain enable the modelling of medical devices, radiation therapy imaging, and X-ray equipment. In the information technology domain, MagicDraw is used to develop a DSML that models IT workflows and inventory management processes. Large industrial appliances such as turbines can be modelled with DSMLs in the energy sector. An integration of the hardware and software aspects of frequency converters and electric motors can be achieved through DSMLs for digital industries. The application of MagicDraw is used to show the implementation of domain-specific aspects in important domains for supporting DSML practitioners. Therefore, the modelling tool of MagicDraw is used as the foundation of realising the presented research questions in this thesis.

2.2.3 The MontiCore Ecosystem

MontiCore [HR17, HKR21] is an open-source[1] language workbench [Fow10] used to engineer textual DSMLs and generates their corresponding language infrastructures to deal with models of the language. In principle, a MontiCore DSML uses MontiCore grammars (MCGs) that describe the DSML's concrete and abstract syntax. MontiCore grammars are described using the custom extended Backus-Naur form (EBNF)-based [ALSU06, Fey16] notation for context-free grammars (CFGs). Using these grammars, MontiCore is able to generate the corresponding language infrastructure such as parsers, abstract syntax trees (ASTs) data structures, a visitor infrastructure for traversing the specified abstract syntax data structure, and the harness for context conditions (well-formedness rules). The language infrastructure generated from a MontiCore grammar is depicted in Figure 2.1.

The grammars defined with MontiCore generally consists of terminals and non-terminals. Terminals are composed of lexical elements such as fixed character strings, while non-terminals are composed of either terminals or further non-terminals. MontiCore also provides interface non-terminals that can extend other non-terminals and, therefore, can be implemented by multiple non-terminals, similar to the concepts from object-oriented programming (OOP). The body of a MontiCore grammar consists of a left-hand side existing of a single non-terminal and a right-hand side containing both terminals and non-terminals, separated with an "=" sign. MontiCore generates AST classes for each non-terminal, and symbol tables are used to lookup AST objects.

MontiCore generates a large amount of language infrastructure. A symbol table infrastructure enables the symbolic links between name usages and definitions [But23]. Java

[1] MontiCore is available via: `https://monticore.github.io/monticore/`

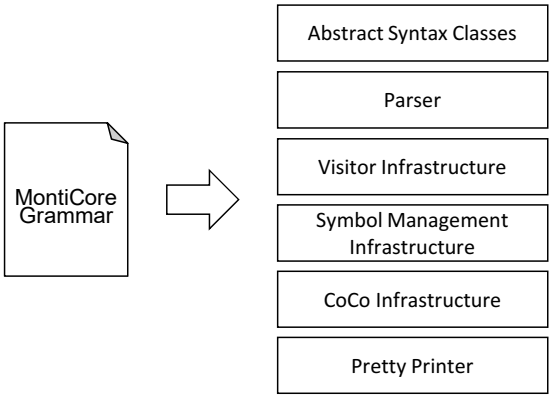


Figure 2.1: A conceptual overview of the language infrastructure that MontiCore provides for an input MontiCore grammar. Figure taken from [BGJ+23].

classes are used to describe the context conditions (CoCo) of the language that check the well-formedness of models by checking against the abstract syntax data structure. Analyses and model transformations is achieved with the visitor infrastructure while giving meaning to a model is realised using pretty printers that translates an AST into a source code conforming to another language. MontiCore also supports code generation realisation using template-based code generation techniques using FreeMarker [For13] templates. Further interesting work on MontiCore, its associated concepts, and its realisation in academic and industrial projects can be found at the end of this thesis under related interesting work from the SE Group, RWTH Aachen.

2.2.4 Language Composition in Textual and Graphical DSMLs

To describe concepts of reusability in software engineering, it is important to decompose a software into smaller components [EGR12]. The individual technological spaces present their notions of language composition, which means the concepts described in composing a language are either vendor-locked or tied to a specific language workbench or graphical modelling tool. For example, textual languages primarily explore the composition of languages using either a shared grammar, a unifying grammar, resolvers, or strongly kind-typed symbol table files [But23]. On the other hand, graphical DSMLs explore language composition through the definition of languages described as part of the configurations that are allowed by the graphical modelling tools.

In MontiCore, a language component comprises of all the software artefacts that are necessary for describing the syntax, context conditions for validating the well-formedness rules, symbol management, and any relevant code generation techniques. A compositional language design for MontiCore is proposed in [DJR22]. MagicDraw, on the other hand, enables graphical DSML engineering [GKR+21]. Here too, software artefacts are

bundled together to describe a DSML although the domain-specific aspects are described using stereotypes and additional custom properties that help realise a language. The software artefacts that are generated and managed for a DSML are a result of the compiled source code for both MontiCore and MagicDraw.

In this thesis, we use the following definition of a language component that is independent of a technological space [BGJ+23]:

Definition 2 (Language component)**.** *A language component is a reusable unit encapsulating a potentially incomplete language definition valid for its respective technological space, including but not limited to the textual or graphical representation. A language definition consists of reusable software artifacts comprising the realisation of the syntax and semantics of a software language.*

Both of the previously mentioned tools, MontiCore and MagicDraw, support four forms of language composition: language inheritance, language extension, language embedding, and language aggregation. All the forms of language composition, except language aggregation, produce a composed language that comprises an integrated syntax of the individual languages. Only in language aggregation do the models remain in their individual artefacts and is therefore considered loosely coupled. Each of the forms of language composition reuses at least one language in some way or the other and the composition aspects described in this thesis are based on the composition of the individual artefacts of the language components. Reusing a language generally requires more effort than completely building a new one from scratch [MGD+16]. Therefore efforts must be undertaken to describe common notions of language components and language composition that serve as the foundation for language reuse across any technological space.

Language Inheritance.

In general, the concept of language inheritance relationships is transferred to textual and graphical DSMLs [Sny86]. Language inheritance in MontiCore is achieved by inheriting grammars. This means that a language is said to inherit from another language if the grammar inherits from the other language's grammar. MontiCore grammars that inherit from other MontiCore grammars reuse, extend, and even overwrite all its nonterminals. Further, a novel grammar rule is specified in the inheriting language to compose a language. Multiple inheritance between languages is possible in MontiCore through the use of interfaces, as the underlying MontiCore infrastructure is realised in Java, which does not support multiple inheritance. All the context condition classes are reused in an inherited language because the context condition checker is based on the visitor infrastructure that traverses the AST and is compositional in nature.

In MagicDraw, language inheritance is achieved at the level of the individual classes that use the graphical notations of generalisations and specialisations on the UML and

SysML stereotypes. The inheritance defined in MagicDraw allows an inheriting subclass to reuse, extend, and override the attributes and methods of one or more superclasses in the parent language. A language, is therefore, said to inherit from another language if the language elements are inherited from the other language. MagicDraw also naturally composes all the context conditions, as the Java validation classes of the composed language are also checked against the inheriting language components. Similar to MontiCore, multiple inheritance in MagicDraw is achieved through the implementation of interfaces.

Language Extension.

Language extension is a particular form of language inheritance where a language simply extends another language. Adding novel parts to a reused language results in language extension. This means that the extension through novel parts is not meaningful without the original language.

MontiCore regards the addition of novel parts to a reused language as *conservative*, meaning the models of the original language continue to remain valid in the extended language. Any language inheritance where the inheriting language reuses the start rule of the inherited language but adds novel parts to the inheriting language is considered a language extension. This means the start nonterminal of the extended language is reused and can add any novel parts to any nonterminal of the extended language.

Similar to MontiCore, language extension in MagicDraw requires that novel parts be added to a reused language. Such extensions in MagicDraw are termed *safe*, since the models of the base language continue to remain valid in the extended composed DSML. Language extension is considered a rather stricter form of coupling as it depends on the reuse of the base language. In all other parts of the language infrastructure, no distinction is made for the concepts from language inheritance. However, if the base language contains restrictions on any particular language element, these restrictions also exist on the composed language, and such restrictions on DSML constructs is called *language restriction*.

Language Embedding.

Language embedding, in contrast to language inheritance and language extension, reuses at least two languages: a *host* language H and an *embedded* language E, without the languages being aware of each other. The main idea of language embedding is to describe how a language is embedded fully into the host language.

MontiCore realises language embedding using multiple language inheritance that is controlled with a novel grammar. The novel grammar inherits from H and E and uses the start grammar rule of H. At some point, a nonterminal of H is overridden, extended, or implemented such that it adds new language syntax from H. This integration of the

two languages is done at a single point. The sets of context conditions for both languages are unified in the composed language.

In MagicDraw, language embedding is realised using an *integration glue*, which is a novel syntax in the composed language that describes the embedding of E into H. This kind of composition allows the host and embedded languages to be reused completely and is set via a tooling property in MagicDraw. Any underlying relations to the language elements in E can be interfaced with H meaning the novel syntax's metaclass is one or more of the metaclasses inherited from H and E. The other parts of the language infrastructure such as the context conditions are naturally composed in the composed language, and newer context conditions are defined at the level of a composed language that checks against both the H and E languages.

Language Aggregation.

Language aggregation also reuses at least two languages. Here, the composed models remain entirely in their own artefacts. However, model elements of these artefacts refer to each other through their instances. In practice, language aggregation requires knowledge of the individual languages meaning they are coupled to a certain extent.

MontiCore achieves language aggregation in a number of ways. Language aggregation in MontiCore is achieved using symbol tables, where symbols are unique elements that are resolved using their names. A shared grammar that specifies the symbols that all models adhere to is one way of language aggregation. Using a unified grammar, the unique constructs of the existing language definitions are accessed, meaning all symbols in both languages are integrated. Languages are aggregated using resolvers that derive symbols from their qualified names by looking up a symbol table. Finally, symbol tables are persisted in artefacts with a symbol table file that allows a complete decoupling of the tools of the individual languages.

In MagicDraw, language aggregation occurs at an *extension point*, where associations to either language are established. By using such relations, models refer to other models of either language. However, the extension point ensures that only at this point the associations refer to the other language's model elements and are otherwise unaware of the existence of any other references or interfaces to the other language. A key benefit of language aggregation to compose languages is that no completely new infrastructure needs to be generated, rather only configuring the associations is required. Other parts of the language infrastructure also naturally compose, similar to the other forms of composition, meaning parts such as context conditions are completely valid in the respective languages.

2.2.5 Interoperability of DSMLs

Managing and evolving complex systems in various business lines is a challenging task. Organisations often have a wide variety of domain-specific concepts that need translations towards a model-driven approach. Larger organisations, or groups within these organisations, use different modelling tools that help realise and solve these domain-specific challenges. However, stakeholders in small and medium enterprises are limited to a single modelling environment and often work with a single modelling tool. A modelling environment is a software environment that potentially consists of language workbenches, graphical modelling tools, and a set of languages combined with such tools. There is little consideration for constructs of a DSML, such as language components, to be easily reusable across a multitude of such modelling environments, meaning these concepts of DSMLs are not often easily interchangeable [AHRW17]. Therefore, there is a need to build modular and reusable components of a DSML that are easily interchanged [HRW16] between different modelling tools and different modelling environments. Conceptually building a common infrastructure that allows modelling similar aspects of a domain in different modelling tools requires extensive engineering efforts. However, this is an effort that must be carried out right from the start of the project. Globalising DSLs using MBSE approaches has been proposed by [CCF+15], although engineering such DSMLs in organisations requires constant support during the entire development lifecycle of a project.

The MDA was proposed by the Object Management Group (OMG) [Spe06] to define an approach that helps separate the specification of a system functionality from the implementation functionality. By elevating the abstraction levels towards more automation, MDA helps achieve portability, interoperability, and reusability of software [AR08] and therefore of DSMLs. As newer systems are developed, newer functionalities are introduced for using newer technologies that need to communicate with each other. This is especially true in web-based systems, as DSLs such as HTML, CSS, and so on, runs on a web browser and often need an exchange of information from other DSLs such as back-end data stores, e.g., SQL. Old and new technologies overlap to create and evolve new systems. Therefore, a large focus of software and systems development is to focus on building modular units instead of bulky, monolithic systems which makes it ideal for communicating with other parts of the system, and for updating parts of the system without significantly modifying other parts of the system. Truly modular units of a language must be seamlessly interoperable between different systems to avoid re-engineering similar language concepts.

Exchanging models and language constructs between various model-driven software and systems development frameworks still requires significant effort [Rum16]. This becomes more challenging as both language engineers and modellers learn to live in their own modelling environments and are not concerned with other aspects of reusability or modularity of DSMLs and their constructs in their projects. The OMG has tried to ad-

dress this challenge of exchanging models in a much more standard way, by using XML Metadata Interchange (XMI) file formats [XMI23] that are based on Extensible Markup Language (XML) [XML23]. This is particularly beneficial while using plain UML as they are much more standard and applicable to almost every domain. However, the concepts do not still translate to exchanging models between DSML environments and are therefore a currently unsolved problem. Further, there must be a distinction between exchanging model information between language workbenches in the same technological space, or across technological spaces, e.g., textual to graphical DSML exchange or integration [Sch08]. Modelling tools are not particular in the way they generate the export and import of domain-specific constructs for similar concepts, which means specifications of semantics of such exchange formats is not easily possible. This means that if UML stereotypes need to be modelled with MagicDraw and later exported to an XMI format, it would generate certain elements that are configured with a metaclass *uml:Stereotype*, while this same export generates a *uml:Class* metaclass when it is exported from Enterprise Architect (EA) [GBJ+24].

Because of the inconsistencies in storing various parts of a DSML, there is a restriction in reusing constructs between modelling environments which often leads to delays in deploying similar domain constructs across organisations. In this thesis, interoperability is considered as the ability of two or more graphical modelling tools to exchange DSML constructs so that these constructs are used effectively without having to rebuild the language definition from scratch. To enable interoperability of DSMLs and their constructs the exchange of metamodels and their properties between the various modelling tools that develop such languages must be possible. To this end, it is important to align such metamodels so that the mapping of DSMLs in an isomorphic form is beneficial in enabling a seamless interchange of domain concepts across various modelling environments. Bidirectional transformations of DSMLs and their constructs allow the domain-concepts to be used in both directions, meaning there must be a kind of transformation definition that handles both the primary transform as well as the inverse of the transform. However, this is difficult in a domain-specific environment because even if the models are semantically equivalent, the abstractions from a business model are often lost in this transformation [KWB03]. Therefore, there needs to be investigations around defining an exchange mechanism to ensure tool interoperability between DSMLs and their constructs that help assist both language engineers and modellers in a multitude of modelling environments using different modelling tools or language development frameworks.

2.2.6 Usability and User Experience in DSMLs

Adapting technology to suit human needs is a key topic addressed by human-computer interactions (HCIs) and usability engineering. These needs are often perpetual and aim to introduce cognitive processes and abilities that allow users to operate a program with

a greater level of efficiency. Just accomplishing modelling tasks with a DSML restricts users from truly experiencing the pleasurable stimulation, feelings, and experiences that are gained from integrating the ever-adapting technology. While a lot of these notions are considered UX, there still exists a lack of guidelines in the technological space of graphical DSMLs. This is because there is a vast difference in opinions and perception as to what a good UX must be for both practitioners and researchers. UX can be defined as a person's instantaneous feeling, good or bad, while interacting with the DSMLs and their constructs. A basic question that leads to a better understanding of UX in DSMLs is "Am I happy at this moment with my models?". The modelling situation can, here, refer to the layout and positioning of DSML constructs, the overall model of a system, or the current relations between different models, among others. A good or bad feeling, therefore, dictates the outcome of the resulting models. It is also rather important that notions of good UX are imbibed into the language development process and that considers all stakeholders relevant to that project.

Modelling tools such as MagicDraw contribute to a great extent in order to improve a user's experience of working with a DSML. This is achieved in MagicDraw using a variety of customisation capabilities during graphical DSML development. However, this also means that language engineers who have experience in language development must also be trained to effectively represent the domains in consideration. They must ideally consider all functional aspects of the language but also focus on the non-functional aspects of the language that improve the general efficiency and modelling experience for users. Combining design guidelines within the modelling language and the modelling tool such as MagicDraw really elevates the user's experience and lead them to modelling with more confidence. This thesis will therefore look at providing guidelines and various design decision aspects that language engineers must consider in developing graphical DSMLs that eventually improve the overall UX for both novice and expert modellers.

The literature has proposed several definitions of UX [Has08] and best practices for DSLs, especially in MDD [Voe09]. As a consequence, several notions of feelings, experiences, insights, and pleasurable stimulation are subsumed under the various definitions of UX. This is because UX is a ceaseless topic with a wide variety of opinions on what a good UX is in each specific domain. Therefore, a single definition cannot possibly provide a solution to all kinds of UX problems. A definition of UX as mentioned in the ISO 9241-210 standard [ISO10] on the ergonomics of human-system interaction is "a person's perceptions and responses resulting from the use and/or anticipated use of a product, system or service". The definition describes UX in a rather generic form and while it can be reused across various facets of software and systems development, it would need clarity for a better UX in graphical modelling that captures domain-specific notions. This means domain experts from different fields of research such as biologists, chemists, mechanical engineers, and so on, cannot all be sufficed with only a restricted view of a good UX. Various studies have observed that there is a general lack of consensus as to which UX and usability definitions such as ISO 9241-11 [BCH15], ISO 13407 [JIMK], or

Nielsen's [Nie00] must be considered for graphical DSMLs.

There exists other aspects of HCI that detail the way people interact and interface with computers, such as with the ISO 9241-161 standard [ISO16], and how it can be used to gather practical results that change the way a user interacts with such interfaces [PRBCZ17]. Absolute consideration is given for developing the syntax of a language, which means the proposed aspects of UX, usability, and HCI are ignored by language engineers [ABC+17]. This is partly also because of the fact that typically such engineers are not experts in designing good UX and are seldom domain experts in the corresponding DSML. Therefore there is a growing need to focus on developing DSMLs that not only focusses on the language definition but also considers UX aspects, because a one-size-fits-all approach is generally unsuitable in domain-specific scenarios. Design aspects that are considered during the development of DSMLs are termed as *user experience design (UXD)* [GJRR22a]. These design aspects compliment a good UX, meaning UXD is defined broadly as any design decision that is undertaken by a language engineer during DSML development that aids in enhancing a user's experience during their modelling. Later this thesis will provide concrete definitions of UX and UXD specific to the graphical modelling space, in particular, defined in the MagicDraw ecosystem.

A key benefit of integrating UX aspects in graphical DSMLs allows practitioners to reach their modelling goals more efficiently. Here, multiple aspects of UX must be considered by language engineers that aim to target a wider variety of users. The challenges of using the constructs of a DSML are greatly improved as a good UX aims to relieve users of the unnecessary burdens of both the DSML and the modelling tool. This is achieved through effectiveness, efficiency, and satisfaction of using DSMLs [ISO10]. It also serves as a starting point in providing effective guidance and suggestions to users in achieving their modelling goals. In the remainder of this thesis, aspects of delivering a good UX to users are discussed by proposing design decisions that must be considered by language engineers for developing graphical DSMLs. A standard set of UX and usability guidelines, that is independent of a specific implementation or a modelling tool, not only elevates the resulting DSML but also provides both novice and advanced users with confidence in their modelling.

2.2.7 Methods in DSMLs

As DSLs and DSMLs are also considered software [FGLP10], they are therefore also subjected to the usual challenges in maintenance and evolution. As defined earlier, a software language [CFJ+16, CBCR15] consists of: (1) abstract syntax; (2) concrete syntax; (3) semantics; and (4) context conditions. In order to project the domain and its constructs effectively, language engineers must acquire skills that transcend simple language development. These language engineers must integrate suitable DSML-based development methods, techniques, and concepts within the industrial DSMLs so that users easily model complex industrial systems. The integration of the DSML, relevant

methods, and the modelling tool ensures that in a single modelling environment, a user is confident in their modelling and that it improves the overall modelling experience. This is achieved through active and synchronised recommendations and suggestions provided to users during their modelling.

Recommender systems [AT05] aim to predict the preferences of users by prioritising a list of potentially interesting items. They are nowadays commonly used in commercial applications and help software developers integrate newer and more effective methods of providing user preferences [RWZ09]. Model-driven engineering (MDE) describes models as the primary assets in the software and systems modelling world and they are used for a variety of purposes. Naturally, proposals to combine recommender systems with modelling tasks have gained traction in the modelling community [PK15, CRM16, ARKS19]. MDE tasks involve the creation of models and metamodels, reusing existing artefacts, or correcting the information for the individual models. A systematic mapping review of recommender systems in MDE has found that most recommender systems are built for models that help in either completing a model or repairing an existing model [AGCDL21]. While some ways to recommend users are through a combination of static documentation and training material, it does not consider the context of the currently designed models and is often very tedious to search through.

Method engineering has been studied to provide mechanisms to design, construct, and adapt various methods, concepts, and tools that help design information systems [Bri96]. These methods are meant to provide a sequence of steps that ultimately accomplish a certain goal [dOCCFD22]. Therefore, they are inherently designed to help benefit all kinds of users. Method engineering has been studied for MDE and DSMLs in particular [MKHdK22, Hon13], but is lacking the necessary bridge to real industrial projects. Often there has been a presumption that users of industrial DSMLs are mostly experts in modelling. However, in reality, users frequently need ample guidance and support in using the DSMLs and for gaining more domain know-how and expertise. To solve this challenge, users must be supported with modelling information that is not only according to their current modelling situation but also provides proposals and actionable items that make their modelling more efficient and equip them with more confidence.

As well as with providing suggestions and recommendations to users, a set of tasks, activities, or processes in the form of a process model [Sch16] allows users to align themselves to their models. This provides them with a more focussed view on the models and advises them with yet-to-be accomplished processes that are part of their incomplete models of the system under development. Passive information is a kind of static information that exists on documents and webpages, which is mostly outdated and not suitable to be searched on. It becomes rather tedious to find any modelling relevant data and even less, how to use a DSML properly to realise the domain's concepts. Alternatively, if users are provided with a rather active and up-to-date information, they would be greatly benefited. As systems grow more complex, it becomes imperative that only specific recommendations related to the modelling are provided to the user

and not information that burdens the user with all possible scenarios thereby reducing their productivity. The literature discusses a few reasons why integrating such methods directly into the DSMLs is a big challenge [Hon13, NR08], and a primary cause is that DSML projects are time-bound and resource-bound and therefore almost all of the effort in a project lifecycle is spent on developing the syntax of the modelling language itself. Such time constraints, resource constraints, and even lack of either software or language development skills means language engineers often struggle to provide suitable methods and stakeholders often resist committing to integrating such methods within the DSML itself.

Overall, language engineers must provide integrated DSML-based development methods that guide and supports DSML users in improving the engineering and use of modelling languages, thereby improving the overall modelling experience. While certain businesses have a single modelling environment, it becomes a necessity to provide all users of this environment with dynamic recommendations that are tuned to their individual state of their models. This thesis will therefore explore the possibilities of defining and integrating model-aware recommendations for industrial DSMLs that are ultimately independent of a specific implementation or a modelling tool and are focussed on a user-centric view such that it is more widely adopted in the modelling community.

2.3 Domain Models

This section lays down the foundation of feature models (Section 2.3.1) and function models (Section 2.3.2) that are used extensively in the thesis to describe the domain-specific world for assisting modellers in developing software using product lines or describing decomposition of systems. While further domains are also briefly discussed in Chapter 8, the concepts described in this thesis are exemplified in the individual chapters mainly using feature and function models.

2.3.1 Feature Model

In software engineering, a *(software) product line* is considered a set of products that together address a specific market segment or accomplishes a particular mission [CN02]. Therefore, the discipline of achieving a consumer's personal preference in software through the production of a family of commonly related software concepts is called *software product line engineering (SPLE)* [PBVDL05]. SPLE therefore produces families or variants of similar languages rather than building each system individually. To model such variability is a challenge, even more so by using modular concepts of reusing parts of a language [BEK+18].

Variants of a software language in a product line are differentiated by their features. These features are functionalities in a software that is incremented gradually by language engineers for a given software [BBRC06]. Each software therefore consists of

these individual features which means consumers using one variant of a software may have functionalities that is different from other variants of the software. Software product lines are therefore specified in terms of *feature models*. Consequently the DSMLs that describe or support feature models are referred to as feature model languages and such feature model languages are used to represent entire software product lines. While some features in a product are mandatory for the efficient functioning of a system, other features may be optional or non-essential. Examples of mandatory features are features that must be included in safety-critical systems in aeroplanes, as the failure of such systems could be catastrophic [WDS+10].

A feature model represents the common and variable features of a software product line at different levels of abstractions and relations. Relations between features define the way a feature model is represented visually. All possible products of a software product line are modelled by including these features and their relations. Therefore, a single feature model represents a family of products so that as a software evolves, the different variants of this software are easily configured. For example, in a graphical user interface, preferences of a user change based on the kind of product that is offered to these users. These are achieved with configurations of variabilities at various points in the model, also referred to as *variation points*. However, as feature models grow in complexity, so does the manual efforts in analysing errors in those models.

In the remainder of the thesis, we consider the following definition of a feature model that is based on the definitions proposed by [BSRC10, Bat05]:

Definition 3 (Feature model)**.** *A feature model consists of*

- *a hierarchically arranged set of features,*
- *three relations OR, AND, and XOR, between a parent (or compound) feature and its child features (or subfeatures), and*
- *constraints between the specified features based on inclusion or exclusion statements, e.g., if a certain feature f is included, then other features, or a combination of features, such as $f1$ or $f2$ must also be included or excluded.*

The relationships between a parent feature and its child features are categorised as:

- **Mandatory.** A mandatory feature is a child feature that must be included in all products and variants of a product.
- **Optional.** An optional feature is a child feature that may or may not be included in subsequent products and variants of a product.
- **And.** A set of child features where all of these features must be included in all products and variants of a product.

- **Or.** A set of child features where at least one of these features must be included in all products and variants of a product.
- **Alternative (Xor).** A set of child features where exactly one of these features must be included in all products and variants of a product.

In addition to the relations between the features, the following basic constraints are also defined for a feature model:

- **Requires.** The inclusion of a feature $f1$ in a product implies the inclusion of feature $f2$ in the case $f1$ requires $f2$.
- **Excludes.** Both features $f1$ and $f2$ cannot be part of the same product when $f1$ excludes $f2$.

While other relations and constraints may be used, this thesis considers only the aforementioned notions of feature models.

2.3.2 Function Model

A function model describes a system as it is observed from the outside. Such a kind of model does not consider architectural details, but how different functions of a system behave and relate to each other. Functions are considered as a bridge between a human intention and the physical behaviour of artefacts [UTY95]. Therefore function modelling can be thought of as a representation of the knowledge of different functions in a system. This is based on the concern that a single developer cannot manage the development effort of an entire system. Therefore, a system must be decomposed into smaller subsystems, meaning functions that describe each system or a subsystem must also be decomposed so that individual aspects of a system are better described. In the remainder of the thesis, we consider the following definition of a function model that is based on the definition proposed by [EKVB+08]:

Definition 4 (Function model)**.** *A function model is a model that describes either a system or parts of a system, including any subsystems, from a purely functional perspective, i.e., as the system is observed from the outside.*

Functional modelling therefore reduces the gap from the high-level requirements of a system and the lower-level implementations [EKVB+08]. While functions are described for individual aspects of a system, they are either related to other domain-modelling constructs or allow the flow of data through them. For example, a function can be logically connected to certain kinds of requirements or certain kinds of features in a DSML project. Functions can be configured with input and output ports that allows the passage of different kinds of data such as signals, materials, human-machine interfaces (HMIs), and various forms of energy. Additionally, a *functional context* provides

an external view of the whole system along with its interactions with the external environment, such as lights, temperature, and so on. Interactions with the environment are achieved through actors (human actors interacting with a function), sources (such as a power supply source), sinks (such as data storage facilities), or any other external functions.

Different kinds of views are provided that support function modelling. A context view provides a functional context view of an aspect of a system. Structural view shows the structure of the main function of the system along with its decomposition. A behavioural view depicts the interaction between the internal functions of a system in a time sequence manner showing the operation of a system under various circumstances. Thus it is beneficial to decompose a system to describe the various parts of a system at an objective level [GRV09].

A primary goal of function models is to show the decomposition of a system. It must be therefore possible to model functions on different layers of abstraction [BBK+21]. While decomposition of functions lead to a better understanding of a system, function models must also sufficiently describe how the functions are composed for showing the overall view of a system. As projects evolve, inputs and outputs of functions on these layers of abstractions must also be adapted and remain compatible during the modelling process. Fail-safe techniques must be considered during functional modelling to detect errors during modelling.

Chapter 3

Systematic Engineering of Industrial DSMLs

The efficient and systematic engineering of user-friendly DSMLs suited for industrial practitioners is quite challenging [MvdSC+18]. DSML practitioners often do not model on a daily basis, meaning models are built over a period of time with assistance from the provided DSMLs [GFC+08]. While efforts must be made to foster the reduction of repetitive modelling tasks, there must also be consideration for modularising concepts of common language parts across a DSML and for providing a holistic UX to such users [KKP+09]. This is partly because industrial language engineers struggle in providing a methodical support that separates various facets of DSML engineering while also reusing language modules. Another reason for developing common language parts is the familiarity when using another DSML such that common concepts are interpreted similarly across different DSMLs. For efficient deployment of DSMLs in an industrial context, the combination of using reusable modules as well as methods to support in the modelling journey of a practitioner is of utmost importance. This chapter lays the foundation of the concepts described in the remainder of this thesis for helping language engineers systematically engineer industrial DSMLs using the concept of DSML building blocks. To achieve this, a definition of a DSML building block is provided, followed by describing various parts of the DSML building block, and how they are composed together to form a DSML. Further, this chapter provides details on how engineering of graphical DSMLs and their building blocks are achieved in MagicDraw. Some results of this chapter have been published in [GKR+21]. Therefore, passages from the paper may have been quoted verbatim in this chapter.

To reduce the conceptual gap that domain experts face during systems engineering [FR07], modelling is being introduced at early phases of the systems engineering process [PB20]. DSLs and DSMLs [Fow10] help reduce the gap between the problem space and the solution space by better supporting abstractions necessary for domain-specific concepts. Generally speaking, DSMLs help understand the syntheses of systems. Graphical DSMLs assist in modelling in various domains, such as MATLAB Simulink [Cha15], or SysML [FMS14], but still do not completely reflect every single domain aspect that are often overlooked during language engineering. Engineering DSMLs that capture a domain's terminology (abstract and concrete syntax), semantics [HR04], and rules (well-formedness checks) is still complicated [MWCS11]. Therefore, it is essential to reuse

parts of a DSML that easily facilitates the creation of newer DSMLs or versions of a DSML to achieve effective domain-specific systems modelling. This means the separation of these engineering concerns must not only include reusing parts of the language definition, but also additional parts such as describing effective methods for using a DSML, and fostering interoperability of language components. Therefore, involving industrial stakeholders from the start of a DSML project helps improve the general usability of such DSMLs [BAG18].

Industrial DSML engineering must therefore consider concepts of reusable units, defined as DSML building blocks, that essentially combine language definitions, methods to help practitioners achieve their modelling goals, and UX concerns that elevate a DSML practitioner's modelling experience. To this end, each DSML building block serves a domain-specific purpose, for example, the support for the creation of feature models, which is eventually combined with other DSML building blocks (e.g., requirements or function models) to create a truly integrated and heterogeneous DSML. In the following, a conceptual model and the definition of a DSML building block is provided in Section 3.2, followed by describing the various parts of a DSML building block in Section 3.2.1 as well as the roles involved industrial DSMLs in Section 3.3. Further, Section 3.4 discusses the engineering of a DSML in MagicDraw including developing DSML building blocks illustrated with an example in Section 3.5. Finally, Section 3.6 discusses the central design decisions, while Section 3.7 compares related approaches.

3.1 Core Elements of a Proposed Method for Industrial DSML Engineering

The ubiquitous GPLs used for software development presents their challenges in modelling software and systems [PB20] as they focus more on the technical implementation details. This not only aggravates analysing systems under development but also prevents domain experts from addressing solutions related to the domains directly. Therefore, DSLs and DSMLs [Fow10] are built and aimed at reducing this gap by supporting domain-specific abstractions and are more accessible to analysis and synthesis of systems and their parts. Despite employing domain terminology, concepts, rules, and meaning, modelling with DSMLs is often less effective than expected. This is partly due to the deployment of DSMLs to their users which often incurs various challenges that include the reusing of syntax and semantics across different stakeholders in different projects. Further, users model less often, probably once a week or even less. This means reusing encapsulated DSML parts systematically facilitates the engineering of new DSMLs and ultimately foster truly domain-specific systems modelling. Industrial DSMLs, in particular, must consider that users are modelling less often than expected and therefore there is a greater need for integrating modelling support and usability considerations through reusable language parts that help such users.

The core elements of a method for industrial DSML engineering are the modelling language, its parts, and an accompanying modelling tool that the language is built and used in. Such a method must not be restricted to a particular kind of meta modelling language, meaning, the meta language described for a method must be interchangeable with any other meta language. This provides the flexibility for language engineers to develop languages that are contextually not bound to certain decisions or standards that DSML projects often come with.

3.1.1 Modelling Language

DSMLs are software languages and software languages are software too [FGLP11]. Hence, DSML engineering is also subject to the challenges of software and systems engineering [TK05]. This means multiple languages and (meta)languages must be considered to define the complete domain area that a DSML represents. In addition to the graphical syntax and semantics for industrial DSMLs, the methodologies and techniques that are highly specific to individual departments must be considered. This is often considered a time-consuming effort in SLE. To solve this problem, either central research units in organisations must become more common in developing truly domain-specific solutions or language engineers must be trained with software engineering development processes to develop solutions that are aimed at helping practitioners easily adopting various modelling techniques. While technical improvements such as explicit language interfaces or language types do foster the reusability of methods in language engineering, there exists a lack of modularly developed language infrastructure that provides not only reusable language components, but also provide the means to guide users, and improve their overall modelling experience.

Modular Language Parts. Reusing a language and its parts is challenging in any modelling environment. While fostering modularity of languages leads to a structured reusable benefit of commonly using domain concept across different environments, often the effort it takes to build a modular language is counter-productive to the needs of an industrial project. However, to truly achieve modularity of languages and for individual languages to be bundled as part of a library of languages, the reusability of languages must be considered by language engineers. Industrial languages represent heterogeneous domain concepts and is often not limited to an independent domain or domain concept. Therefore, a method to provide a library of languages is beneficial in ensuring complex languages are logically and independently developed through various building blocks. These building blocks of languages must capture the essential language infrastructure of a particular view of a domain. DSML building blocks serve as a stepping stone in presenting views such as a technical view, a logical view, or a functional view that would otherwise be intertwined together with minimal logical separation in a DSML. These blocks must be constructed in a way that they are flexible enough for extensions and adaptations, and ensure they provide the necessary interfaces to interact

with other building blocks. Essentially, a building block consists of a reusable language infrastructure including the language components, methodological techniques, and usability aspects that provides the complete modelling capabilities for a particular view of an industrial domain.

Language Components. From a technical perspective, SLE and component-based software engineering (CBSE) communities promote the reuse of existing and future software solutions. Therefore, language engineers undertake the task to maintain and reuse software artefacts, while breaking larger artefacts into smaller decomposed artefacts. Similarly, in DSMLs, language components primarily describes the definition of a language, while providing the necessary artefacts and interfaces required for the component to interact within its modelling environment [BHP+98]. The artefacts generated as part of the definition of a language component must be stored in specific file formats that are generated as part of the compiled source code. This means that artefacts could be grammar files to parse a textual language infrastructure or through storing information related to the graphical language definitions in software file systems. Language components are composed through mechanisms of language inheritance, extension, embedding, and aggregation for ultimately fostering the reusability of independent language parts to build larger DSMLs. Therefore, language engineers must consider a thorough specification of the language infrastructure to ensure that the individual language components comply to the concepts that they describe for a domain. Defining language infrastructure that are used across modelling environments fosters the need for modularity of DSMLs and their building blocks. To effectively, interchange models between tools [LMT+18], methods to exchange language constructs across modelling environments must be developed. A modular and reusable component of a DSML provides an effective solution in exchanging DSML concepts across modelling tools [HRW18]. Such language components, when developed independently, require minimal transformations when integrated in another modelling tool [BGN+04]. However, extensive efforts are required when conceptually building a common infrastructure that allows the modelling of similar aspects of a domain across different modelling environment.

Recommendations and Process Models. Methodological techniques aimed for improving the experience of modellers must provide modelling solutions that help in either completing the design of a certain model or to repair existing through recommender systems models [AGCDL21]. Various tasks such as the creation of models and metamodels, reuse of existing models, or interlinking already designed models must be considered as part of the overall modelling in an industrial DSML project. While static information are readily available to modellers through documentation, training materials, and handbooks, conveying active and dynamically changing information pertaining to the models is still a challenge. This thesis, therefore explores methodologies that are integrated directly within the individual DSML building blocks and the modelling tool for providing a more user-centric set of suggestions and recommendations to users that are based on the current modelling context of users. This aims at providing modellers with a focussed

view on those models and parts of their system that are currently under development. Further, process models, activities, and tasks that describe the extent to which models have been currently developed ensure that modellers are provided with a more holistic view of their modelled systems and take various actions to further improve their models.

User Experience Concepts. The methodological techniques contribute largely to the modelling experience of modellers. Thus, user experience and usability serves an important role in ensuring that modellers feel positively during their interaction with the DSML and its constructs. Any negative feelings must be lessened to ensure that the resulting models not only are closer to their actual systems, but also integrate well with the other models of the system. Therefore, language engineers must ensure that DSMLs consist of methods, techniques, and usability design aspects that enhance the overall experience of all stakeholders in the project [PRS$^+$]. This ensures that DSMLs are efficient and that modellers are satisfied both with the use of DSMLs and their constructs as well as within the modelling environments.

Modelling Stakeholders. Various language engineering tasks are undertaken by various stakeholders. While, the responsibilities of these stakeholders in an industrial DSML project vary, each stakeholder must have a clearly defined role in the organisation. In essence, this thesis considers the roles of language engineer, domain experts, and modellers to be the most important ones in the context of the engineering and use of DSMLs. The development and maintenance of DSMLs and their functionalities must be performed by language engineers who build the language infrastructure for the complete DSMLs and their building blocks. They must also consider integrating certain methods and techniques that eventually help ease the difficulties in using a DSML along with a modelling tool. Domain experts possess expertise in particular application or technical domains [FR05]. They are responsible for providing their domain know-how and familiarity with various domains to solve complex domain problems. Domain experts also closely work with language engineers and modellers to provide a bridge between the technical and business aspects of any industrial DSML projects. Finally, modellers are the users of DSMLs who create models towards reaching a certain modelling outcome. They are not necessarily experts in language engineering and may not also be domain experts, but must be skilled in working with DSMLs and the chosen modelling tools.

3.1.2 Modelling Tool

The efficient use of a DSML requires a corresponding modelling tool. The development of the tool itself is not considered an integral part of a DSML development as it requires extensive software engineering. Often, language engineers and modellers live in their individual modelling environment, therefore reusability aspects within a certain modelling tool are lost as projects become complex. Therefore, language engineers must consider the development of tool independent solutions at the start of any industrial project. As functionalities to import and export domain-specific constructs for similar aspects of

domain differ with every modelling tool, the exchange of semantics and methodological techniques also requires a deeper understanding that is answered subsequently in this thesis. To completely understand a described language part, graphical notations are an essential part of the modelling tool and the modelling language. They foster comprehensibility of DSMLs as otherwise in a graphical modelling environment a language will be rendered illegible. This means special considerations to the way DSML parts are expressed to the user must be considered and both the modelling tool and language be developed accordingly.

These described core elements: the modelling language, its reusable components, methodological techniques, process models, user experience considerations, and the corresponding modelling tool contribute to the proposed method for the systematic engineering of industrial DSMLs and have been studied subsequently in detail in the remainder of this thesis. The description of DSML building blocks and their constituents, methodological techniques, and usability considerations all contribute to the main research question. The concepts described in various chapters have been presented from an industrial perspective with various real-world examples being reused. The approaches consider engineering DSMLs from a graphical perspectives while reusability towards textual environments are also discussed to promote the development of these concepts independent of a technological space. Modularly designing software and systems is important in an ever growing heterogeneous and complex world with an extensive number of interdisciplinary concepts. The reuse of common language parts for building holistic DSMLs is necessary to facilitate a greater confidence in industrial language engineering. To this end, these building blocks of languages must be able to provide the necessary language infrastructure for representing a particular aspect of the domain.

3.2 DSML Building Blocks

The systematic engineering process of developing industrial DSLs using modular reusable DSL building blocks is described in [GKR+21]. While the mentioned study describes the engineering process in graphical DSLs, the same process is reused for graphical DSMLs as well. This means a DSML potentially consists of various DSML building blocks, each designed to solve a particular aspect of a domain-specific problem. Therefore, a DSML building block is defined as follows:

Definition 5 (DSML Building Block)**.** *A DSML building block is a reusable unit that helps in the creation of efficient DSMLs aimed at modelling software and systems in a variety of domains.*

In other words, a DSML building block provides the language infrastructure, i.e. the (potentially incomplete) language definition, methods, and concepts, to represent a particular view of a single system of interest. This means, if a DSML concerns with describing the concepts of only one aspect of the industrial domain, such as for describing

feature models, then a feature model DSML will consist of only one DSML building block that captures the complete language definition and concepts describing a feature model.

However, in practice, industrial DSMLs often require the representation of multiple interdisciplinary domains with intersecting concepts [HJK+23]. For example, a DSML that is used in the healthcare industry must provide the ability to model different aspects of a medical project such as the functional and non-functional requirements for a medical device, use cases, actors, tasks, features, system functions, and medical device architectures [MC08]. Here, the concepts of each of these individual aspects of the DSML are modularised into reusable DSML building blocks that are eventually composed into the final DSML. These DSML building blocks are reusable to the extent that the language infrastructure for a DSML need not be built from scratch every time a new DSML, with overlapping domain-specific concepts, has to be developed. This means, if a DSML building block x is used in DSML d, then the same DSML building block x or a slightly different version of x (x'), is reused to build DSML d' which is a DSML similar to d but developed for another customer. This fosters the reusability of language parts as well as allows for the efficient creation of versions or families of a language [Beu08], which are important in SPLE [PBVDL05]. Real world case studies of DSML building blocks and how they are reused is further described in Chapter 8.

The concepts presented in the remainder of this thesis, unless stated explicitly, apply to DSML building blocks and hence implicitly to DSMLs as well. Therefore in the course of answering the various research questions, if a DSML is described for modelling the concepts that enables the solution to specific aspects of a domain, we consider the DSML to consist of one or more DSML building blocks that are configured with the language infrastructure for that particular domain.

Figure 3.1 shows a conceptual model of a graphical DSML and the various DSML building blocks that compose together to form the complete language infrastructure. While Def. 1 in Section 2.1 defines the constituents of a DSML, the provided definition here considers a DSML from the perspective of defining a language that describes a particular domain. In practice, an industrial DSML must provide the ability to model different aspects of the domain in consideration. The individual aspects of a domain are logically grouped together and defined in the individual DSMLs building blocks. As an example, one aspect of modelling using an industrial DSML is the ability to model functions for an industrial domain. Therefore, describing the functional aspects of a system through function models (Section 2.3.2) must be possible using a DSML building block that describes the language infrastructure needed to model functions. Another aspect of modelling using an industrial DSML is the ability to model features for representing the common and variable features and their relations in a software or system product line. This is described using feature models (Section 2.3.1) that is possible using a DSML building block that describes the necessary language infrastructure needed to model features. The concepts for modelling different aspects of a domain are therefore captured

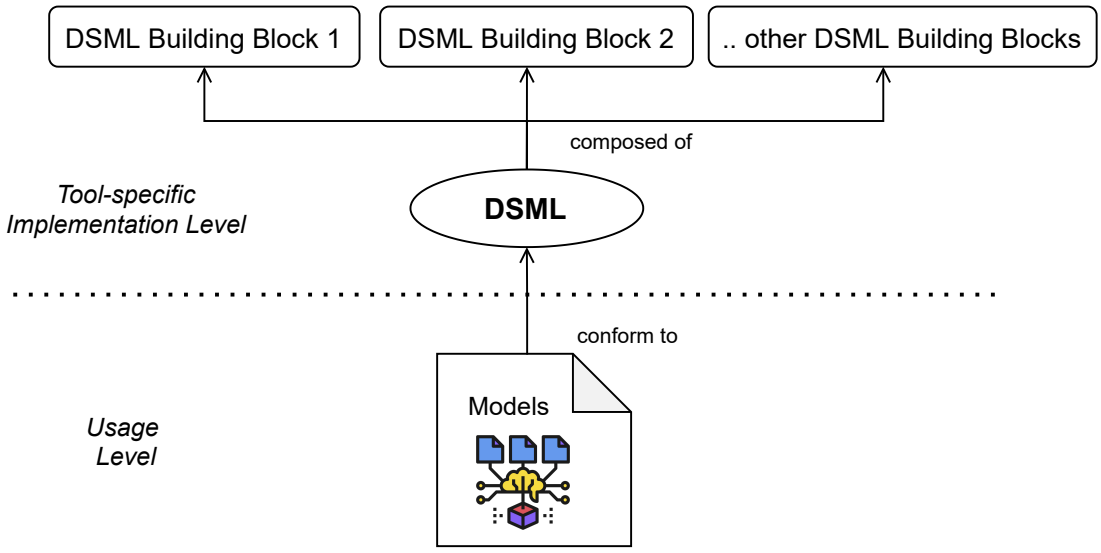


Figure 3.1: A conceptual model showing the artefacts of a graphical DSML. The DSML is composed of different DSML building blocks each representing the language infrastructure for solving individual modelling aspects of a domain. Figure adapted from [GKR+21].

and represented in the individual DSML building blocks. This means, `DSML Building Block 1` could be used to model the functional and non-functional requirements for a project, `DSML Building Block 2` is used to model features of the system, and `DSML Building Block 3` is used to model functions of the system. Other DSML building blocks are similarly used to model other individual aspects of the system which are described in detail in Chapter 8. Therefore, a DSML in reality is the combination of these individual DSML building blocks and such a language expresses the complete domain that is required to be modelled by a practitioner.

This approach separates the concerns of industrial DSML engineering and use into the following two levels:

1. the tool-specific implementation level, where language engineers realise the concepts of a domain by creating the required language infrastructure for a DSML using a graphical modelling tool, and
2. the usage level, where modellers are able to use the provided DSMLs in a modelling environment for designing models.

Because a DSML consists of different aspects that need to be modelled, and the kinds of DSML users are diverse, it is important that the combination of the DSML building blocks is described sufficiently in a way that it captures most, if not all, of the domain concepts.

3.2.1 Parts of a DSML Building Block

Every DSML building block consists of the language infrastructure needed to describe a particular aspect of the domain. In our earlier definition of a DSML (Section 2.1), we considered the abstract syntax, concrete syntax, semantics, and well-formedness rules, as the parts that form a DSML. While this is true in a strictly technical language sense, additional language infrastructure must also be considered for industrial DSMLs. This is because substantial efforts are spent in designing the syntax of a language because of time and resource constraints. However, this often leads to static and dull usage of DSMLs, with very little support or guidance to users. It is necessary that DSMLs must be equipped with more active and continuous support for the individual domains that go above and beyond the plain modelling language and its supporting tool. Therefore, a support for providing methods, guidance, and continuous domain knowledge must be an essential part of DSML engineering. To this end, a language infrastructure must also provide methodological guidance that is:

- context-sensitive,
- active, to the current state of models, and
- evolve as DSMLs grow.

In the context of industrial DSMLs, the creation of this kind of language infrastructure is achieved by combining available technologies, such as with commercial modelling tools, supported with existing usability guidelines to model complex scenarios.

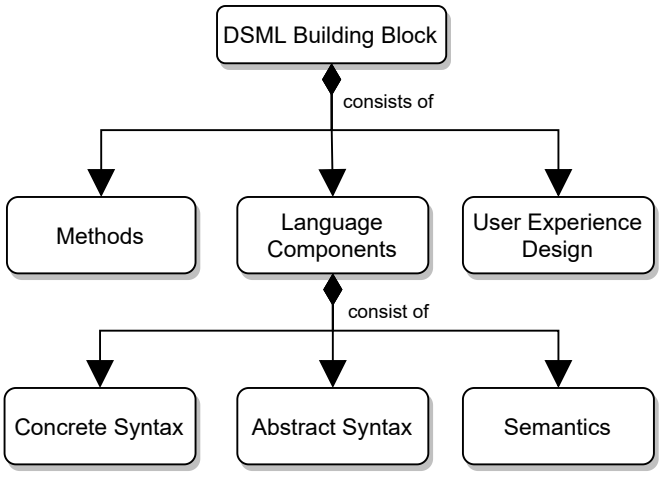


Figure 3.2: The different parts of a graphical DSML building block that consists of the language infrastructure for representing a specific aspect of a domain. Figure adapted from [GJRR22b].

Such a language infrastructure is provided for a DSML building block by defining the

parts as shown in Figure 3.2. Each DSML building block must essentially consist of a set of *language components* that describe partially, or in whole, the language definition needed to describe a particular aspect of a domain. For example, a feature model DSML building block consists of language elements and artefacts that effectively represents a feature model [Voe10]. *Methods* must provide active, synchronous recommendations, guidance, and supporting techniques to users so that they are assisted in their modelling that relate to the domain under consideration. For example, a feature model DSML building block must contain methods that help users in configuring mandatory or optional features, or simply take them through their feature model journey. Methods also correlate to *pragmatics* [Tha12], a notion used to describe how languages are used for intended deployment functions contributing to the purpose and goals of modelling situations. Finally, UXD aspects must be integrated directly into the infrastructure of a DSML building block, such that they help users in a seamless modelling experience Section 2.2.6. As an example, providing colours and icons to different kinds of features in a feature model DSML building block allows users to better understand and visualise the constructs of a feature model, such as for distinguishing the mandatory features from the optional features. The following discusses these parts of a DSML building block in more detail. Further, subsequent chapters of this thesis, Chapter 4 for language components, Chapter 7 for method support techniques, and Chapter 6 for UXD also detail the parts of a DSML building block at a more in-depth level.

Language Components.

The most essential part of a DSML building block is the definition of the language itself. Without describing the syntax and semantics of a language, it is impossible to use a DSML. To this extent, reusable *language components* that allow for defining a (potentially incomplete) language [Rum16] is used to achieve modularity in industrial DSML engineering. Therefore, a language component essentially consists of an abstract syntax, the concrete syntax, and the semantics as described in Figure 3.2. Building a family of languages that share common concepts is further promoted through reusable language components which fosters the creation of similar versions of DSMLs. Such a family, library, or catalogue of languages is beneficial in eliminating the need for building DSMLs from scratch. Language components that are part of a DSML building block, therefore, include artefacts of a (potentially incomplete) language definition for a particular domain [But23]. A language component is thus also defined as reusable unit consisting of language artefacts that allows for defining a particular domain's language. While language components are sufficient to describe a DSML building block, and therefore a DSML, it is not often adequate in describing the entire graphical DSML engineering process, as this means considerations for parts other than the pure syntax and semantics are not considered [Fra13]. Language components typically consist of software artefacts realised in the form of files in a file system directory format. The

abstract syntax is normally described for a graphical DSML as class diagram files, the graphical concrete syntax is described using graphical elements, e.g., box-and-line based, tabular, or tree-based [CFJ+16], and the well-formedness rules are realised using Java class files. Further concepts of language components and their composition techniques is described in Chapter 4.

Methods.

A DSML must provide the necessary domain constructs needed to describe a particular domain, or a set of domains. Thus the language engineering process must be supported additionally with reliable and active methods that help users understand and make effective decisions for the domains in consideration [Hon13]. With the increase in complexity in the syntax and semantics of a language [CGR09], multiple domains are now strongly getting interlinked, and an industrial DSML therefore typically represents concepts from multiple domains. It becomes increasingly necessary to guide and assist users in creating effective models that is representative of these interlinked domains. Such methods to guide and support users in their modelling need to be established to cover specific modelling aspects during various stages of designing models. Often, practitioners lack the know-how in every aspect of their domain, as there is not enough guidance in their current modelling scenarios [Roq16]. The method part of a DSML building block (Figure 3.2) describes various ways to provide such a guidance to users. Generally, the following methods (non-exhaustive) are described for each DSML building block:

- (1) general training material for models in the form of overview, documentation, guides, hyperlinks, training videos, methodical steps, and commonly asked questions need to be directly integrated within a DSML building block;

- (2) various configurable business rules helps provide active, synchronised, and dynamically changing recommendations to users that identify incomplete or missing parts of a DSML diagram, as well as specifying certain properties or values of DSML elements that could be based on historical or recommended data; and

- (3) prescriptive process models in the form of activity diagrams that detail the current state of the models, and describe tasks and activities that have either already been performed or needs to be performed to achieve the desired modelling.

Describing methods presents its own challenges as such additional methodical aspects are only defined on the composed DSML to ensure all domain aspects are captured in providing guidance and recommendations to users. Further concepts and realisation techniques for providing suitable methods that guide users in their modelling is described in Chapter 7.

User Experience Design.

As domains become more heterogeneous and the complexity of languages increases with respect to the syntax and semantics of these languages, providing a good UX to practitioners becomes a key notion. Methods described so far assist and guide users in their modelling, but providing such methods is also a challenge for language engineers. They need to consider both the combination of language components and methods, whilst also integrating techniques for improving DSMLs that make it easy for practitioners to model with such DSMLs. To this end, UX aspects are integrated directly within language components as well as the methods.

By defining and implementing standards of UXD and usability heuristics directly on the individual DSML building blocks and therefore the DSML and its accompanying modelling tool, the overall modelling experience of users are improved [GJRR22a]. It must be noted that UX is a subjective topic, and the preference of users often differ vastly for each domain. As an example, a user would like to see features in a feature model DSML more prominently defined and visible, rather than plain technical requirements that are important in the larger context but not necessary in a feature model. However, with a combination of DSML building blocks, addressing these challenges is also important to solve concerns of all key stakeholders. Improving the UX for users is key to achieving modelling goals with the accompanying language constructs and methods.

UX is described in a way it invokes positive instantaneous feelings for a user during interactions with various constructs of a DSML or within the modelling environment. To this end, certain design decisions, termed UXD, are integrated by a language engineer that ultimately leads to a good UX, one that leads to positive rather than negative feelings during modelling. Chapter 6 details extended definitions of UX and UXD, and lists down a set of categories, standards, and usability heuristics for designing a good UX.

3.3 Roles in Industrial DSML Engineering

As with any other software engineering task, language engineering also consists of a number of specific roles that are performed by people possessing certain skills. These roles are used for different kinds of tasks, such as for conceptualising, designing, implementing, or even using a DSML [KRV06]. The engineering and use of industrial DSMLs involves three main roles, *language engineers*, *domain experts*, and *modellers*. The different roles are specified such that the different kinds of tasks and responsibilities are separated based on the capabilities of each role. It is also however normal that a single person takes up responsibilities or be trained to accomplish tasks that belong typically to some other person, therefore representing multiple roles. This thesis primarily discusses the following three roles that will be referred to, throughout the thesis.

3.3.1 Language Engineers

A language engineer's main task is the development and maintenance of DSMLs and its building blocks. This includes building reusable language components, creating methods integrated with a DSML building block, and ensuring aspects of UX are considered within each building block, and hence the DSMLs (Section 3.2.1). They are therefore considered experts in engineering languages. Aspects of defining a language's syntax and semantics, defining extension and variation points in a DSML (Section 2.2.4), specifying methodical steps, recommendations, and process models, as well as improving the usability of DSMLs are some of the important responsibilities of a language engineer [HRW18]. However, they are not considered experts in designing user interfaces nor are they experts in a particular domain. In general, language engineers must possess a basic understanding of software development and therefore have the ability to create basic programs using a GPL, such as Java, to build customisations for the DSMLs, but is not a necessity and they must be trained to acquire this skill.

3.3.2 Domain Experts

Domain experts are not necessarily language engineers - sometimes they are not even software or systems engineers, but people who possess expertise in a particular domain [FR05]. A domain generally refers to application domains such as a business process or a discipline of engineering, or even to technical domains such as relational databases or even state-based systems [RW11]. Domain experts are generally people who have gained special knowledge or a skill in that particular domain. They are generally experts in the problem domain, sometimes even in neighbouring domains, and who are also familiar in modelling with this knowledge. Domain experts often work with language engineers in defining the specifics of a DSML and assist in customising different parts of a DSML based on stakeholder requirements. Examples of domain experts are system architects, key experts, business analysts, and subject matter experts.

3.3.3 Modellers

Modellers are often users of DSMLs who use the DSMLs to create models based on their modelling goals. A modelling goal is a statement that is aimed towards reaching a desired modelling outcome [CFB05]. Such users do not possess expertise in language engineering and are often not even software engineers or developers. While modellers can also be domain experts, it is rarely so in industrial projects as the number of stakeholders is quite high, and each stakeholder has their own responsibilities. Modellers must however possess basic knowledge of the modelling tools they work with, or gain experience in using such tools as they progress with their modelling. It is also helpful for modellers to be equipped with language engineering expertise as they can then comprehend the syntax and the semantics of a DSML much better. In an industrial context, modellers

are also often referred to as industrial practitioners or merely practitioners [GKR+21]. Throughout the different parts of this thesis, the terms *modellers*, *users*, *industrial practitioners*, or *practitioners*, are interchangeably used. However, they commonly refer to the roles defined for a modeller.

3.4 Industrial DSML Development in MagicDraw

This section describes a systematic engineering process involved in the creation of graphical DSMLs. It must be noted that other modelling tools also serve the same purpose of language development, hence the concepts described in this section are transferable to a certain extent to other modelling tools as well (Section 2.2.1). However, further research is needed to validate this concern and the scope of this thesis considers only the out-of-the-box and customisation capabilities that MagicDraw offers. For now, the remainder of this section describes how DSMLs for industrial contexts are engineered using a mixture of MagicDraw's functionalities and custom GPL code such as those written in Java.

3.4.1 Language Profile with Stereotype Definitions

Defining a DSML in MagicDraw requires the creation of a language profile that eventually consists of language components and their configurations [Neu06]. All the parts of a DSML are therefore created in a dedicated language profile. As a DSML building block consists of language components, these language components are therefore inherently located inside a DSML profile. A DSML profile in MagicDraw not only defines the various stereotypes of domain-specific elements, but also allows defining customisations to these elements. These elements are configured with additional rules such as context conditions. Further, methods and UXD aspects are directly included in a DSML building block, and are therefore tightly integrated with the language components themselves (Def. 5). However, because of budget or project constraints, language engineers may choose to design a language profile that solely consists of the set of language components for a DSML and ignore designing additional methods or UXD aspects as part of the language infrastructure.

In order to define DSML elements, individual stereotypes are created for each domain-specific concept. This allows only specific elements to be assigned a particular kind of stereotype and hence a particular UML metaclass. Each of these stereotypes are configured with customisations that help set the meaning and behaviour of the DSML elements, while also providing opportunities to integrate aspects of good UX directly for each configured language element. As an example (shown in Figure 3.3), a stereotype only allows a "Functions" package in a function model DSML building block to contain functions and sub-functions, but prevent inputs and outputs data flows types to be present in this package. However, input and output data flows are allowed to only exist

in another package ("In/Out Types") dedicated for such elements in the function model DSML.

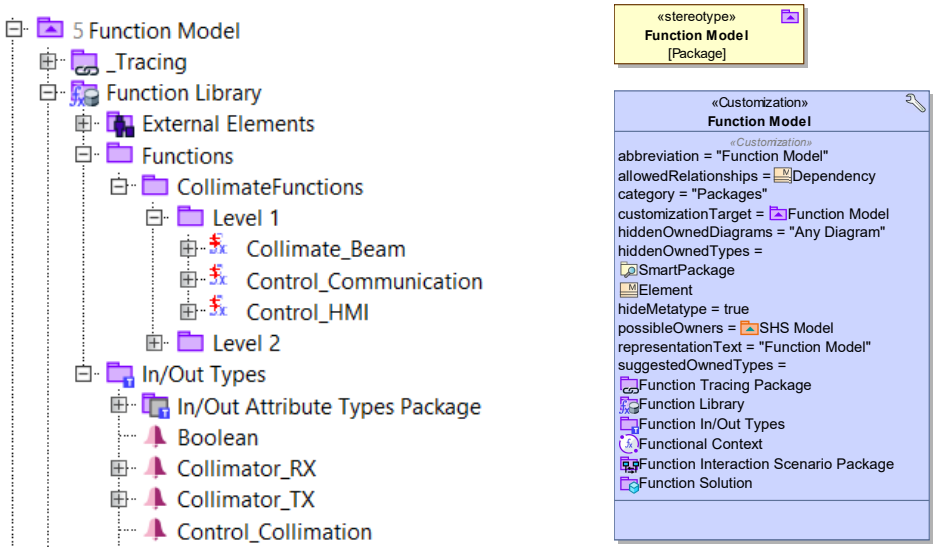


Figure 3.3: (Left) An example of a function model that consists of different function packages and functions, and (right) the stereotype definition of a function model package.

3.4.2 Customised UML Diagrams for a DSML

Once the language components are created as part of the language profile, language engineers create custom diagram definitions for their DSMLs. MagicDraw provides the capabilities to create these custom diagrams that are based on UML diagrams [Mag20]. Every custom diagram supports the creation of customised toolbars containing DSML elements that are specific to that diagram. This means a function context diagram must be able to provide functions and their interactions along with any other supporting data flows between these functions. These toolbars directly assist in creating model elements with the already applied stereotypes, thereby reducing manual efforts to configure domain-specific properties.

As part of these diagrams, language engineers also define custom matrices, tables, and relation maps, that are all considered types of diagrams in MagicDraw. The custom diagrams are always intended for specific purposes and are preconfigured to show only those elements, their attributes or relations, that are required for that specific purpose and is described later in this thesis in Section 6.3.1. A diagram for defining features, therefore, must restrict the creation of functions on the same diagram, unless the diagram shows the overall context of the features and its relations to certain functions.

3.4.3 Embedding Textual Languages in MagicDraw

The creation of graphical DSMLs in MagicDraw cannot be truly considered graphical, since some specific model elements are better written in a textual form. Therefore, one can argue that MagicDraw predominantly fosters graphical modelling, but allows support for integrating textual languages. The model elements such as guards and actions as part of state machine diagrams are considered better represented in a textual form [HKR21]. MagicDraw comes with a predefined set of textual languages that include object constraint language (OCL) [RG02], groovy [Koe07], or structured expressions that are represented in XML [XML23]. But the main challenge with these languages is that they are not extensible enough, meaning a modeller cannot reference arbitrary model elements in MagicDraw language profiles. Referencing arbitrary textual model elements requires models of such languages to be editable such that they are stored in the graphical editor of MagicDraw [DJRS22]. Further, those textual model elements must be checked for correctness such that their equivalent model parts in the graphical editor are also checked accordingly. To reference textual model elements, certain kinds of symbols must be used through the use of a symbol table infrastructure that promotes an AST to a graph based structure [But23]. Functionalities that are associated with a textual model element must be seamlessly integrated within MagicDraw, which means there must exist a mechanism to provide hook points to refer to these model elements [DJRS22]. Therefore, simple expressions such as validating or assigning values are usually not supported and must be developed. Finally, support for guiding users into using a combination of textual and graphical editors must be provided that alleviates the challenges of embedding textual languages into MagicDraw.

To solve this challenge, methods have been proposed to integrate custom textual languages in MagicDraw [GJRR22b, DJRS22]. To validate models, modellers enter String-based expressions in a text box for defining the guards or actions, and this expression is processed by a parser. This parser is based on an existing library of language components [BEH+20] using MontiCore. If the input text cannot be parsed, this validation mechanism reports an error, otherwise proceeds to construct the symbol table for the language profile. In this way, references are resolved to access certain attributes of model elements and the elements are type checked as well as checked if they are accessible from within the current scope.

3.4.4 Predefined DSML Project Templates

Once the language profile, custom diagrams, and any textual languages have been integrated into a DSML, a project template is configured. This template is a predefined project pattern that initialises the basic models for a DSML. This template therefore sets up a model quickly and serves as a starting point for a modeller. It contains not only a predefined numbered package structure, but also other language elements and

custom diagrams within specific packages as shown in Figure 3.3. Project templates aid in improving the UX and are therefore also described later in this thesis in Section 6.3.3.

3.4.5 Customising Functionalities of MagicDraw with Perspectives

Next, language engineers define certain kinds of *perspectives* that modellers benefit from. These perspectives allow further customisation of the language profiles and custom diagrams that add or remove certain constructs of the DSMLs as well as functionalities of MagicDraw itself. Adding or removing toolbar menu entries, context menu items, and displaying specific elements for specific diagrams are configured as part of perspectives. Modellers therefore focus more on their modelling concepts, with novice users guided more easily, and advanced users utilise more functionalities that the DSML offers.

In addition to such perspectives, sufficient levels of documentation and help must also be provided to modellers. This is achieved by using MagicDraw's functionality of providing dedicated hyperlinks to all the configured language profile elements and custom diagrams. Modellers additionally use a dedicated *Help* button, created using Application Programming Interface (API) extensions, to be redirected towards more information about their models. Concepts of perspectives and documentation are further discussed later in Section 6.3.2.

3.4.6 Extending MagicDraw Functionalities through APIs

To enhance a DSML in MagicDraw with further custom concepts, API-based extensions are created that are written in Java. Such extensions are used to visually enhance the DSML elements, or automatically attach a stereotype on a model element upon instantiation on a custom diagram, or create additional validation checks that are not possible by default with MagicDraw. Automatically assigning stereotypes to elements means language engineers configure the element to show only specific properties, colours, or icons, for a specific element to ensure consistency with the DSML concepts. Visualisations help improve the look and feel of model elements that make them distinct. Custom validation rules, or context conditions, for checking the accuracy, completeness, and correctness of models are written in Java, such that the models of a DSML are checked for well-formedness. These rules are configured to run either automatically during design time or are manually triggered to be executed.

3.4.7 DSML Development Overview for the Defined Roles

Language engineers develop and maintain DSMLs. Figure 3.1 described a conceptual model that shows the different artefacts in an industrial DSML development process. At the tool-specific implementation level, language engineers build the DSML building blocks as well as the resulting DSML. On the usage level, modellers understand the provided DSML including the constructs of the DSML, the various methods needed to

reach their modelling goals, and other language infrastructure that is needed to successfully model their systems and subsystems. While such users may or may not be domain experts, they must possess a basic understanding of the involved domains.

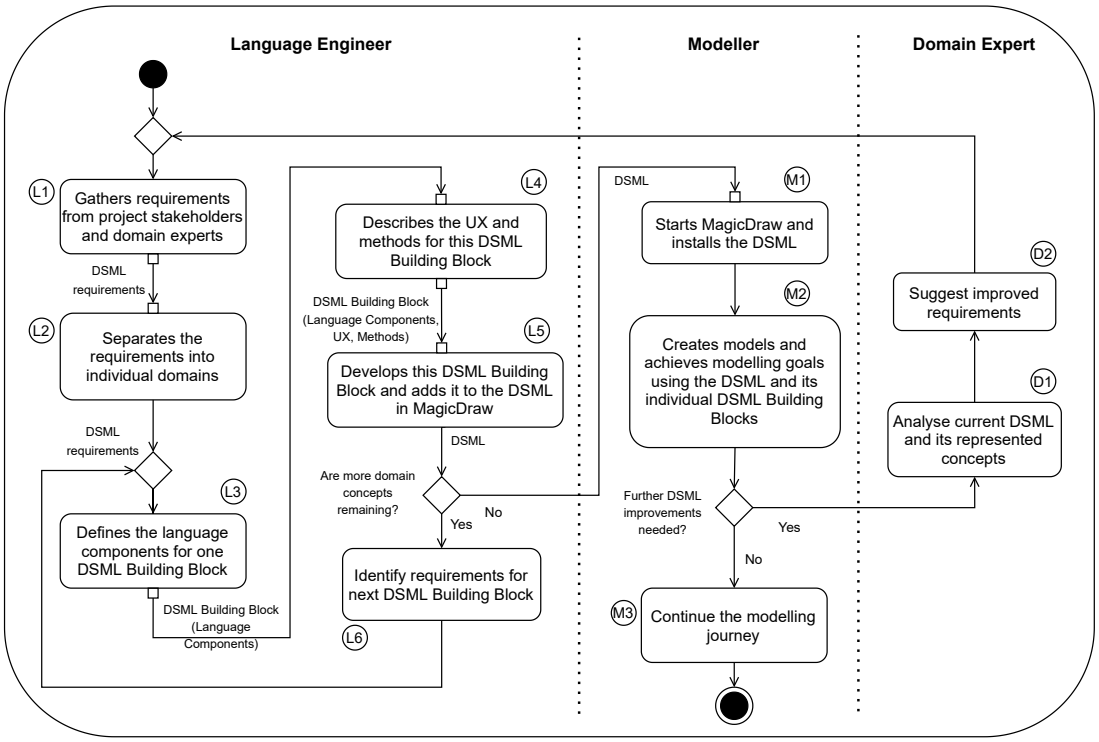


Figure 3.4: An activity diagram describing the tasks and activities involving the three roles in the development and usage of an industrial DSML. Figure adapted from [GKR+21].

Figure 3.4 shows an activity diagram that describes the tasks and activities performed by the three roles. As this methodology follows an iterative developmental approach [BT75], further requirements are continuously refined and added at different stages in the development lifecycle. Therefore, in reality, the different activities and blocks in the activity diagram will usually be executed in an agile approach. The following discusses the individual activities and tasks in more detail. To begin with, all project stakeholders and domain experts meet to discuss the modelling project requirements.

(L1) Language engineers understand and gather all the requirements for a DSML along with all the stakeholders. Involving all stakeholders early on in the projects ensures that the key project requirements are clear from the start and also influences integrating a good UX with suitable methods.

(L2) The language engineers then need to separate out the requirements based on the domain under consideration. For example, if a project requires the modelling support for both function and feature models, then the language engineers easily distinguish both of these concepts to create individual language infrastructures that support modelling these aspects of the domain. Therefore, in this case, language engineers will create:

1. a function model DSML building block, and
2. a feature model DSML building block.

(L3) However, to ensure that all the aspects for a modelling functions is completely developed, the language engineers first decide to develop a DSML building block for a function model by defining the various language components.

(L4) They further integrate various methods and UX aspects aimed at improving the modelling experience of a modeller.

(L5) In general, two scenarios exist for developing a DSML building block:

1. If the DSML for a given domain is already developed in another project, then the corresponding DSML building blocks are simply **reused** and modified with any special project requirements. In this case, creating versions of a language is easily achieved, since it does not involve developing all existing domain aspects completely from scratch.
2. If the DSML for a given domain has not yet been developed, and is completely **new**, then the language engineers must develop the complete language infrastructure for this domain from scratch. Here, although efforts need to be made to develop new DSML building blocks, ultimately such a language infrastructure are easily reused in other projects.

The outcome of both scenarios is to prevent the development of the complete language infrastructure for a domain more than once.

(L6) The language engineer next identifies the concepts of a feature model into a feature model DSML building block, develops it accordingly, and integrates it with the same DSML having the function model.

(M1) The finally built DSML is provided by the language engineers to a modeller, who installs the DSML in their own modelling environment, here a MagicDraw instance. With the previously described concepts in this section, the DSML is readily available at tool start-up.

(M2) Modellers use the concepts integrated in this DSML to create models to achieve their modelling goals. Each DSML building block serves an individual purpose of modelling specific aspects of a domain, e.g., either functions or features of a domain under consideration. As the development of a DSML is iterative, continuous feedback and improvements to the DSMLs is suggested by both the modellers and the domain experts.

(D1) Any further improvements to the DSML is then analysed by domain experts for suggesting improvements in the DSML. These domain experts also consult project stakeholders and modellers to ensure there is no gap in the domain-specific understandings of the concepts described with the DSML.

(D2) Domain experts further suggest improvements necessary to enhance the functionality and usage of the DSML. This feedback is sent back to the language engineers for continuously improving the state of the DSMLs and for providing up-to-date language components, enhanced UX, and newer methods that support and guide a modeller more effectively.

(M3) Modellers continue to use the DSML and create their models further continuing their modelling journey.

3.5 Example

To demonstrate the applicability of DSML building blocks in MagicDraw, let us consider the example of an industrial project for modelling a medical system. A medical company wants to model a specific part of their medical system that involves examining a patient and generating reports for this patient. This examination involves a patient entering a patient room consisting of various devices such as input and output devices, power supplies, and mechanical elements such as table that the patient will lay on. Thus, to model such a medical system or parts of this system, various functional and non-functional requirements, functions with different views supporting different types of data flows, as well as features for this system linked to either the requirements or the functions must be defined. This means that the aspects of modelling that need to be considered for designing the medical system involves feature models, function models, and requirement models. Conceptually, the DSML that is built for this medical company would be composed of individual DSML building blocks that provides the ability to model these individual models as well as the overlapping concepts from the other models.

Figure 3.5 shows an example of how the various DSML building blocks of the medical system are conceptually represented for this industrial DSML. Based on the development overview for the defined roles, language engineers first identify and gather the

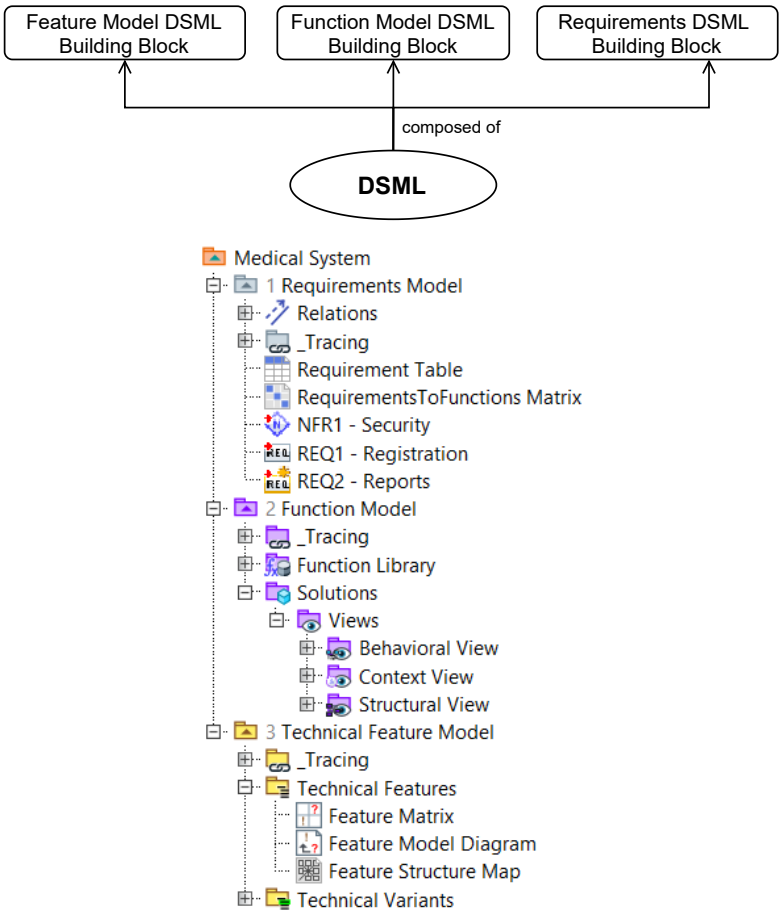


Figure 3.5: An example of an industrial DSML consisting of the various DSML building blocks (top) that represent the feature model, function model, and requirement model in MagicDraw (bottom).

requirements for this DSML along with all the stakeholders. They must decide which aspects of the domain must be enabled for the medical company for modelling specific parts of their system. Here, the emphasis is on separating out the concerns of the various models that will be designed by users. This means, a medical system must be able to provide the means to model requirements, functions, and features. Separating the language infrastructure for designing individual models allows the DSML more flexibility in terms of capturing different abstractions of the system. For example, the requirements model DSML building block provides the necessary language infrastructure to model the requirements of the medical system. These include modelling both functional (*REQ1, REQ2*) requirements and non-functional requirements (*NFR1*). An example of the lan-

guage definition for a functional requirements DSML element is shown in Figure 3.6. This figure shows customised attributes of the stereotype that are used to enhance the experience of a user. For example, the abbreviation ensures that new elements are created with the prefix *REQ* and the status of this requirement will be set to *New*. The icon on the top right distinguishes this element from other elements, e.g., the non-functional requirement seen in Figure 3.5. Further, matrices and tables are defined as part of the language definition to better visualise and represent requirements information directly to the user. Additionally, different icons, colours, and naming conventions contributes to the overall improvement in the UX for a modeller.

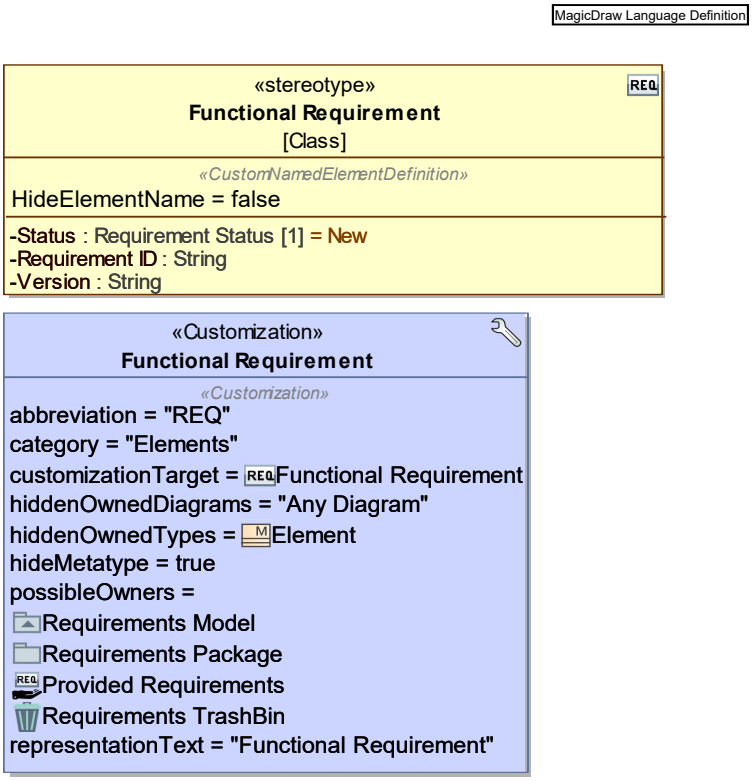


Figure 3.6: The language definition of a requirements DSML element.

Similarly, the language infrastructure for the respective DSML building blocks are constructed by language engineers and constantly enhanced due to the agile nature of such industrial projects. As an example, the language definition and the subsequent use of the function model to create function DSML elements was described in Figure 3.3. Thus, stereotypes and their customisations serve as the basis for defining the syntax of the language (Section 3.4.1), while matrices and tables are used to show specific elements,

their attributes or relations. The result of configuring a project template (Section 3.4.4) for this medical system is shown in Figure 3.5, where the requirements model, function model, and the feature model are all numbered in a sequence starting from one. This enables users to quickly start configuring their models as a predefined project pattern is already created for them.

Figure 3.7 shows the links between the requirements model to the functions in the function model for this medical system in a matrix (Section 3.4.2). The *RealizesRequirement* association links a function to a requirement and the numbers represent the number of links between between them. In this example, the *GeneratePatientReport* function realises *NFR1 - Security* and *REQ1 - Registration* requirements, therefore the sum of the number of associations (2) is displayed as part of this function. Here, the menu bar at the top is configured to show an additional perspective to the user (Section 3.4.5) that allows a user to export the contents of the matrix to an excel file using the extended MagicDraw functionalities through APIs (Section 3.4.6). The functions described in this figure involves positioning the patient and generating reports for that patient. Therefore, the language infrastructure for representing different concepts are separated through these DSML building blocks and provides for a modular approach to building interdisciplinary DSMLs.

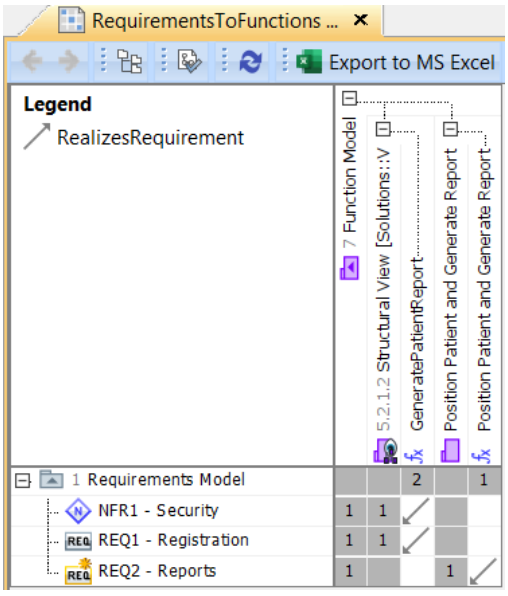


Figure 3.7: A matrix showing the links between functions (columns) and the requirements (rows) they realise.

Each DSML building block must also provide the support and guidance to modellers for making effective decisions in their modelling. This is described using training ma-

terials, documentations, recommendations, or activity diagrams. Figure 3.8 shows a prescriptive process model in the form of an activity diagram for describing the method of a requirements DSML building block. This process model details tasks and activities that the modeller must consider to model specific healthcare and software capabilities and requirements of an individual medical system. Modellers are guided to focus on identifying healthcare problems and needs, defining the quality of software required, and verifying, validating, and managing such requirements through innovative development processes [CR16]. Such a process model not only helps in the know-how of the current modelling situation but also provides a kind of guidance to modellers to further improve their models.

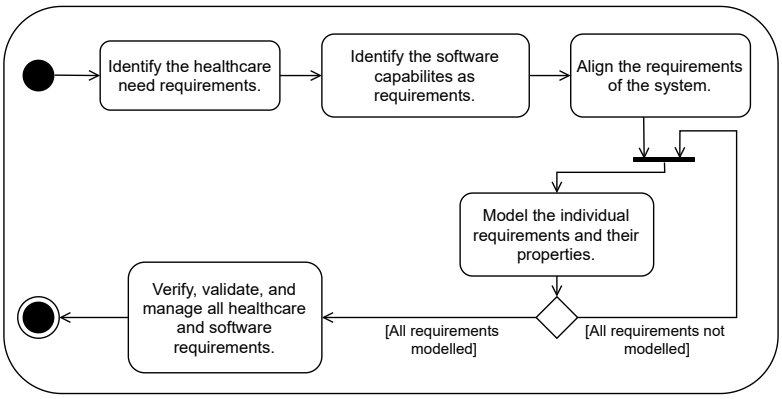


Figure 3.8: A prescriptive process model describing the sequence of methodical steps for modelling the needs and software requirements of a healthcare system.

An advantage of describing these individual building blocks is that if a different industrial project now requires only, for e.g., the ability to model features, functions, and requirements, then these building blocks are easily reused again without completely developing them from scratch. This example is later, in Chapter 8, extended to a more detailed medical system in the context of a medical company and further discussed for reuse in the context of a digital industry example for modelling software architectures and for implementing an MBSE methodology with MagicDraw. Therefore, the example described in this chapter briefly discusses the core elements of the proposed method and lays the foundations for discussing the concepts in subsequent chapters.

The implementation of the DSML for modelling a medical system was done to show how the different language components and UX techniques were imbibed into reusable language infrastructures through the DSML building blocks. The creation of stereotypes, their customisations, perspectives, templates, and API extensions through the examples provided in this chapter demonstrate how separating the concerns of different system concepts enable language engineers to build modular and reusable parts of

a DSML. In particular, each DSML building block must describe language components and their constituent artefacts for representing the concepts of a particular system under development. Figure 3.5 (bottom) showed the different model constructs relative to the medical system through its described DSML. This means, elements such as *REQ1*, *REQ2*, *RequirementsToFunctions Matrix* are all considered elements designed using the language infrastructure provided. The models that represent different software and system concepts, such as requirements, functions, and features, are captured in their individual DSML building blocks. For example, a *Technical Feature Model* consists of various kinds of features such as mandatory or optional, as well as other kinds of custom diagrams such as *Feature Matrix*, *Feature Structure Map*, and *Feature Model Diagram*. Products and product lines are modelled within the *Technical Variants* package that is further discussed in Section 8.1. The *Function Model* consists of various types of views (described in Section 2.3.2) that containing the different types of functions and their data flows within the *Solutions* package. Cross-references and relations between functions and requirements are modelled using the *RequirementsToFunctions Matrix*.

Later, in Chapter 8, the same, or slightly different versions of these DSML building blocks will be reused to demonstrate an actual implementation of the systematic process described in this chapter for developing industrial DSMLs. Therefore, DSML constructs that help model functions in a system are easily be reused across a variety of projects that require support for function model. By separating the concerns for representing individual software and system concepts into these DSML building blocks, a more seamless reusable process to develop, maintain, and use DSMLs and their constructs is achieved. Language engineers are able to develop such modular and heterogeneous DSMLs that help modellers and domain experts in designing models for their respective domains.

3.6 Discussion

The approach presented in this chapter allows language engineers to systematically engineer and maintain graphical DSMLs. The main aspects to consider during engineering of such graphical DSMLs that are deployed in industrial contexts are reusability of common domain parts and providing a more holistic experience for modellers. To achieve reusability of common domain aspects, the separation of concerns of industrial DSML engineering is essential. Every domain is represented using a set of domain-specific constructs that are used to model abstractions of real world systems and subsystems. Because DSMLs are often not used to model on a daily basis, efforts to create such reusable concepts are largely ignored [GFC+08]. This means modularisation of domain concepts are also ignored, which leads to language engineers often building DSMLs from scratch every time. This is not only counter-productive as more effort in development is required, but it also reduces the consistencies in defining domain-specific aspects. For example, a function modelled in one independent function model for a DSML may look

and be modelled completely different than a function for a completely different DSML from another organisation.

To address these challenges, this chapter presents a systematic engineering process of developing DSMLs that are based on the concept of reusable DSML building blocks. In theory, a DSML is developed to represent and model concepts of a single domain, hence *domain-specific modelling language*. However, in practice, especially in the industry, one DSML is often configured to represent multiple domains that help model a complete system and all of its constituents [BCCP21]. Therefore, a DSML actually consists of multiple domain aspects that are modularised and logically grouped together into reusable units that help create efficient graphical DSMLs. Each DSML building block therefore represents the concepts for modelling a single aspect of the domain and are reused across any industrial project that require the integration of these aspects of the domain. This is especially beneficial in the case when the same or a slightly different version of this DSML building block needs to be created. A single DSML building block is therefore used across different projects and is enhanced with newer language infrastructure or allow the removal of unnecessary language infrastructure.

The principle idea is that each DSML building block consists of the language infrastructure that is needed to describe a particular aspect of a domain. Therefore, defining the concepts of the domain along with describing any accompanying guidance for the domain constructs are also beneficial for modellers. To this end, DSML building blocks consist of language components that, partially or in whole, defines the language for the domain, methods that support active and continuous support for the domain, as well as any usability considerations that help improve the overall UX for using a DSML. Defining the language for a domain is the most important part for any DSML. Language engineers typically spend most of the project resources to create a solid foundation for representing strictly the domain aspects from a technical viewpoint [Gro09]. In doing so, usability considerations are often overlooked. Methods generally assist modellers in their modelling related to a particular domain by providing training materials, recommendations, and prescriptive process models for their current state of their modelling. UXD aspects are integrated directly on the language definition as well, such that the constructs refer more closely to real world abstractions which are growing more complex and heterogeneous for every modelling project.

Language engineers are generally responsible for the development and maintenance of DSMLs. This chapter discussed a development overview that considers the different aspects and software artefacts involved in a DSML development process. The development and use of a DSML is separated out into a tool-specific implementation level and a usage level. While both of these levels are closely interlinked, a separation of this concern enables a language engineer to strictly focus on language development while a modeller is designing their models. Initially, the project requirements is logically separated and grouped into different aspects of the domain that a language engineer identifies. For each such group, they build a corresponding set of language components, methods to

support the domain-concepts, as well as integrate UXD aspects that generally help improve the usability of a DSML. This is achieved and is demonstrated in this chapter using MagicDraw's default functionalities and customisation capabilities. Language profiles, customised diagrams, templates, perspectives, API extensions, and subsequently support for embedding textual languages help define a comprehensive set of DSML building blocks and the corresponding DSML. The idea of this development overview is to foster the reuse of commonly used domain constructs that prevents the reinvention of language infrastructure every single time.

To effectively adopt the concepts presented in this chapter, language engineers and domain experts must be trained with advanced concepts of software and systems engineering. These concepts include the ability of language engineers to design and develop customised solutions using GPLs that enable the generation of language infrastructure such as debuggers and editors. This is achieved by the use of design patterns [DJR22] and object-oriented techniques [BHRW21]. Domain experts on the other hand must be trained in skills that allow them to comprehend different aspects of individual domains and to subsequently analyse and understand complex software and systems that are typical in their application domains [CFJ+16]. This is critical in bridging the gap between the methodologies presented and its implementation. Problems such as ineffective usability of DSMLs, modellers' unfamiliarity with a multitude of DSML constructs, steep learning curve of the tools involved, and inefficient interoperability of modelling tools hinders the adoption of complex DSMLs in the industry [KWJM02]. Therefore, breaking down a DSML into modular reusable parts alleviates the overall engineering and improves the usability of DSMLs for modellers. A decisive link of communication between language engineers and domain experts allows the core elements of the proposed method of industrial DSML engineering to be effectively built through separation of concerns. This means clear modelling goals must be defined at the start of a modelling project with flexibility of iterative development of advanced domain concepts. Modellers using a DSML help measure the performance of such a methodology by how easily they are able to model different complex modelling scenarios that involves various domain concepts. In practice, this methodology is suitable for complex modelling problems that are further illustrated in Chapter 8. The concepts presented in this chapter provides ample flexibility such that newer domain concepts are seamlessly integrated into existing DSMLs or when completely new DSMLs need to be developed. Developing smaller DSMLs that concern only a specific aspect of a domain, which is well-known by a modeller and does not need to be extended in the future, does not require the use of the approach mentioned in this chapter. This means developing a DSML that supports the modelling of only feature models for modellers, who are domain experts in feature models, would require additional language development effort for integrating support methods and UXD. However, such a one-dimensional development approach would be disadvantageous for modellers who are unfamiliar with feature models. While other approaches to graphical DSML development on graphical modelling frameworks are beneficial in the development

of the syntax and semantics of a language [CJKW07, Voe09], the approach presented in this chapter discusses their extension to allow support methods and focus on UX that helps both language engineers and modellers. The concepts are presented in a generic way such that it is applicable to any modelling framework that supports the access of model and language construct information through APIs as well as with supporting the creation of a language (Def. 1). Therefore, a graphical modelling tool must support the flexibility to customise parts of the displayed functionalities of the tool for allowing the better usability of a DSML and its constructs through methods and improved UX.

The concepts presented in this chapter is realised using MagicDraw [Mag20]. However, it must be noted that other graphical modelling tools such as MetaEdit+ [Tol06] and Enterprise Architect [Ent23] have different technical capabilities, which poses a challenge towards adopting this methodology seamlessly across all tools. Language engineering tools such as MPS [VV10], Spoofax [WKV14], and Melange [DCB+15] provide certain means for language composition and customisation, but lack the techniques to provide methods for systematic reuse of similar domain concepts. The concepts described in this chapter is presented in a way that it is generic enough to be easily adopted in other modelling environments, as the core idea of graphical modelling tools is the same, language development. Further work includes providing methods that allows language components to be easily analysed to make it more machine-processable and accessible to automation techniques. Overall, the systematic engineering process of DSMLs described in the chapter, and in the remainder of this thesis, not only fosters reuse and interoperability of language parts for language engineers, but also provides a more holistic set of methods to actively support modellers in fostering continuous modelling.

3.7 Related Work

To develop, maintain, and easily evolve complex systems, it is important that the underlying technology that supports in building such systems support concepts of reusability. Systems and software therefore need to be designed in a way that it captures all relevant aspects of a single domain into reusable units, that are easily attached to grow larger, complex systems, or detached such that users are often not burdened with unnecessary concepts [CCF+15, CBCR15]. To scale model-based development in large heterogeneous systems, a modular approach using compositional modelling has been proposed in a previous study [HKR+09]. These compositional modelling aspects have been described in [BR07, RW18] to detail modular aspects in interacting systems. While a language and its components, or a set of artefacts, are necessary to detail the essential parts of a DSML [TAB+21, BEH+20], their applicability is rather restricted in offering language engineers the flexibility in quickly developing versions or catalogues of languages based on similar concepts of a domain for industrial practitioners. A DSML must therefore consider the usual challenges faced in software engineering too [FGLP10].

In general, a software language consists of [CGR09, CBCR15]: an abstract syntax [HR17, CBW17]; a concrete syntax [DCB+15, Bet16, Cam14]; semantics [HR04]; and context conditions [HKR21]. The research area of SLE [Kle08, HRW18] has discussed defining and improving the overall situation of engineering DSMLs, e.g., with language workbenches and graphical modelling tools [EvdSV+13, BLWPO13]. With the core idea of language development, many such tools provide language engineering support such as generation of debuggers [DCB+15], editors [Bet16], or reusable language modules [HR17] from abstract syntax descriptions. Even then, the methodologies for systematically engineering DSMLs in the large using SLE or MDE [BCOR15] techniques are rare. Those studies that do discuss details on how industrial graphical DSMLs are implemented [MGD+16, TK05], often describe such methodologies in a highly specific manner, meaning they are either tied to specific departments, or simply require more efforts in fostering reusability than creating DSMLs from scratch. There is therefore a need to either train more language engineers that implement a reusable DSML units along with better usability for users, or central research units must become more common in truly developing DSMLs from scratch every time.

Many studies have looked at providing a comprehensive DSML development approach [MHS05, Hud98, KBM16]. [ICLF+13] propose a collaborative method that is based on completing different stages, from process bootstrapping, to meta-model induction, to evaluation and discussion, to voting phase, and finally language development. While such a method engage users to play an active role in language development, often this leads to more overhead because of project constraints such as time or budget. Reusable and generic design decisions for developing UML-based DSLs have been proposed in the literature that are relevant and important in various software engineering projects [HSS17, CG11, Fra13]. These design decisions are certainly helpful in providing a more holistic development approach, but must be used in combination with various methods and guidance that help practitioners utilise a complete end-to-end language infrastructure. Using free modelling as an agile method to develop DSMLs is possible by allowing the building of both models and metamodels at the same time at different levels of abstraction [GBD+16]. Here, the authors discuss about the possible implications of using graphical DSMLs and how the tools for the development for graphical DSMLs must be flexible and support reusability and extensibility, while also ensuring consistency. Tools other than MagicDraw has been evaluated as language development platforms in a previous study [AFR06], therefore the results of this thesis provides language development and infrastructure techniques primarily in MagicDraw, but also reusable in other language development tools.

Various roles are defined in the development and usage of a DSML [KRV06]. Modelling with a DSML must be easy and seamless for users as it reduces complexities in using a language's constructs [CTVW19]. Usability of a DSML must be evaluated at various stages in a project [PPZ+21]. While general design guidelines for DSLs and DSMLs are found in plenty [KKP+09, BDH94, Fra13], their application is rather limited

to DSLs in general. This thesis discusses elaborated guidelines later that is intended for industrial DSML practitioners. So far, literature provides a handful of solutions in providing modelling recommendations to practitioners [ARKS19, AR20, NKBK12]. However, suggesting active modelling recommendations that are in sync with the current modelling situations of modellers is still missing and is addressed later in this thesis. The remainder of the thesis takes a deep dive into language components, active method recommendations, and usability guidelines that are essential in fostering the systematic engineering and use of graphical industrial DSMLs.

Chapter 4

Language Components in the MontiCore and MagicDraw Ecosystems

For a DSML, or parts of the DSML, and its building blocks to be reusable, the modularisation of the language is crucial. Modularisation is key to the reusability of languages and their components across individual technological spaces, such as in textual language workbenches or graphical modelling tools. Reusable parts of a language are often not directly transferable across individual modelling spaces [BCC+10]. To develop DSMLs from scratch, significant efforts by language engineers are required, and even with concepts of generalisation, creating DSML building blocks that are reused are challenging. This chapter discusses the development of language components and the various forms of language composition that are utilised primarily in a graphical modelling environment, such as in the MagicDraw ecosystem, and that fosters the reusability of common language components in order to build modular DSML building blocks that are reused across instances of the modelling tool, or even across the textual modelling space. To achieve this, a definition of a language component in MagicDraw is provided, followed by the various forms of language composition that is realised in MagicDraw. Then, a short comparison to similar language composition techniques in MontiCore, a textual language workbench, is discussed. Finally, unified cross-cutting concepts of language components are discussed that foster the development of versions or families of language with similar domain concepts in developing complex and modular DSMLs. Some results of this chapter have been published in [BGJ+23]. Therefore, passages from the paper may have been quoted verbatim in this chapter.

As discussed in Chapter 3, DSMLs and their individual building blocks consists of composable units that we call *language components* which is important for modularisation. While there are language workbenches and modelling tools that support such modularisation, defining them in a common way is important. In the textual space, some language workbenches such as MontiCore [HKR21], allow defining DSMLs using a textual syntax. In the graphical space, modelling tools such as MagicDraw [Mag20], allow creating languages with graphical concrete syntax [GKR+21] by separating out the concerns of language development. A difference to note here is the way in which language workbenches assign meaning to models [HR04], such as by using interpreters or code

generators. These workbenches are also capable of validating the language and its components using well-formedness rules as context conditions using OCL like notations or GPL implementations such as using Java classes. While, several definitions of language components are proposed in the literature [CKM+18, BW21, BEK+18, CBCR15], they are almost exclusive to a specific technological space and are almost entirely reusable only in their respective environment.

Reusing a language, or parts of it, is already quite challenging even in a single modelling environment. In some cases, it takes more effort to reuse a language, than to implement a new one, for example using clone-and-own [MAGD+16, ŞvdBV18] concepts. At the end of this chapter, we shall understand how a common notion for language components enable the composition of languages and how building blocks of a language are useful in bridging the gap between different technological spaces. In the following, using a running example of two basic languages in Section 4.1, the concepts and realisation of language components in MontiCore is first described in Section 4.2. Then the requirements and properties of language components are discussed in Section 4.3, followed by the individual definition of a language component in MagicDraw in Section 4.4. In Section 4.5, the various forms of language composition in MagicDraw are described, before Section 4.6 presents a joint definition for language components in both the textual and graphical-based technological spaces. Section 4.7 discusses the central design decisions and threats to validity, while Section 4.8 compares the approach to existing literature.

4.1 Running Example

The mechanisms of language components and how they are composed as complete languages in MontiCore and MagicDraw are illustrated in the remainder of this chapter using two basic, yet commonly used stand-alone modelling languages. First, a language *UseCaseDSML* similar to UML use case diagrams for modelling simple use cases and their relationships is described. Second, a language *ActorDSML* for describing actuators and their tasks is described. The aspects of language composition that are described for these languages offer additionally the possibility to model unique techniques. In the following, we describe these two languages in detail and elaborate on their implementation in MontiCore and MagicDraw.

4.1.1 Use Case DSML

The models of individual use cases are represented using a UseCaseDSML. Use cases allows defining the high-level functionality of a system. The UseCaseDSML is a language that is, in principle, similar to UML use case diagrams that allows the modelling of use cases and their relationships.

Figure 4.1 shows the context-free grammar in EBNF notation [Wir96], representing the realisation within MontiCore. It contains production rules, defining nonterminals on the

```
01 grammar UseCaseDSML extends MCBasics {
02   symbol scope UCDiagram =
03     "usecasediagram" Name "{" UseCase* "}" ;
04
05   symbol UseCase =
06     "case" Name ("extends" ext:Name@UseCase)? ";";
07 }
```

Figure 4.1: MontiCore grammar for the UseCaseDSML. Figure adapted from [BGJ+23].

left side, with terminals or references to other nonterminals on the right side, separated by an equals sign. Furthermore, nonterminals are augmented with stereotypes, such as symbol or scope, indicating unique access via their name, enabling structuring and cross-referencing in the symbol table. The grammar defines the overall diagram (ll. 2-3), starting with a respective keyword, identified via a name, and containing an arbitrary number of use cases. These use cases also have a unique name, as they are defined as symbols, and may extend other use cases by referencing these via their name (ll. 5-6).

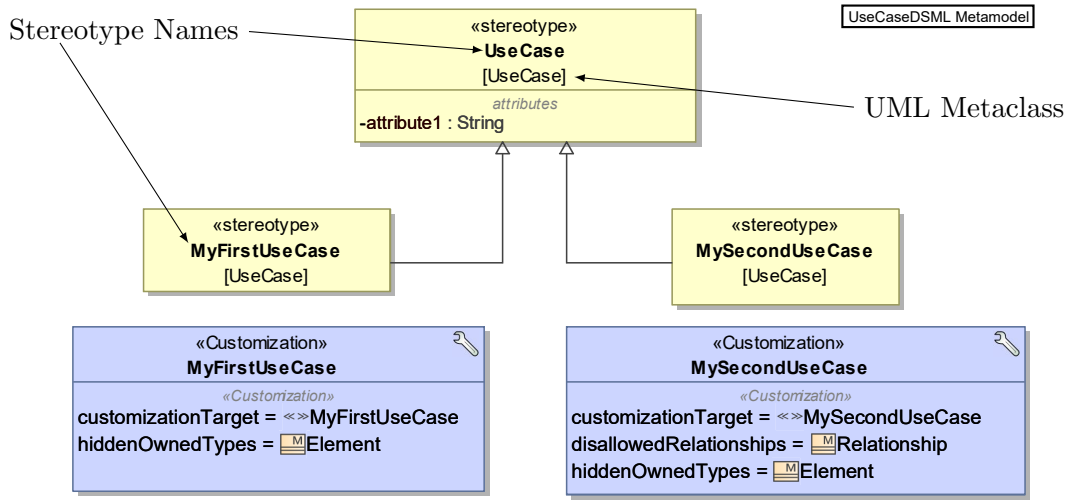


Figure 4.2: A MagicDraw metamodel for the use case diagram modelling language detailing use cases. Figure adapted from [BGJ+23].

Figure 4.2 shows the MagicDraw metamodel for realising the UseCaseDSML. The stereotypes are indicated using the «stereotype» keyword, while the customisations are indicated using the «Customization» keyword. An arbitrary number of use cases are modelled using the stereotype `UseCase` and its attribute, `attribute1`. In this exam-

ple, two specialised versions of use cases, `MyFirstUseCase` and `MySecondUseCase`, are configured. The `MySecondUseCase` explicitly disallows all relationships for this diagram element, that is specified in the `disallowedRelationships` property. The definitions of the stereotypes and customisations are configured in a DSML Profile (Section 4.4.2) and is sufficient to describe the syntax of the language.

4.1.2 Actor Task DSML

The models of individual actors and the tasks they perform are represented using an ActorDSML. Actors specify a role played by a user or a system that interacts with the use cases defined previously [Spe07]. The ActorDSML is a language that enables the modelling of different actors, their tasks, and the corresponding relations.

```
01 grammar ActorDSML extends MCBasicTypes {
02   symbol scope RoleDiagram =
03     MCImportStatement* "rolediagram" Name
04     "{" RDElement* "}" ;
05
06   interface RDElement;
07
08   symbol Actor implements RDElement = "actor" Name
09    ("extends" sup:Name@Actor)? ";" ;
10
11   symbol Task implements RDElement = "task" Name ";";
12
13   Relation implements RDElement =
14     actor:Name@Actor "->" task:Name@Task ";" ;
15 }
```

Figure 4.3: MontiCore grammar for the ActorDSML. Figure adapted from [BGJ+23].

Figure 4.3 shows the textual grammar for defining the ActorDSML language within MontiCore. The model (ll. 2-4) starts with a set of import statements, followed by the actual diagram definition containing the respective keyword, a name, and an arbitrary number of diagram elements (cf. `RDElement`) in curly brackets. The import statements allow referencing and hence importing other models and their features, thereby facilitating the composition of different artifacts. `RDElement` is defined as an interface nonterminal, enabling the grouping of several nonterminals. Thus, each nonterminal that `implements` this interface (cf. `Actor` l. 8) is employed where the general `RDElement` is referenced. This mechanism offers an explicit hook point intended for extension in inheriting languages. The diagram itself contains actors, tasks, and relations. An actor (ll. 8f.) has a name and extends other actors by referencing them via their name. Tasks (l. 11) start with a corresponding keyword and are also identified via their unique name.

Finally, relations (ll. 13f.) associate actors with their tasks, utilising their property as a symbol for unique cross-referencing access via their names.

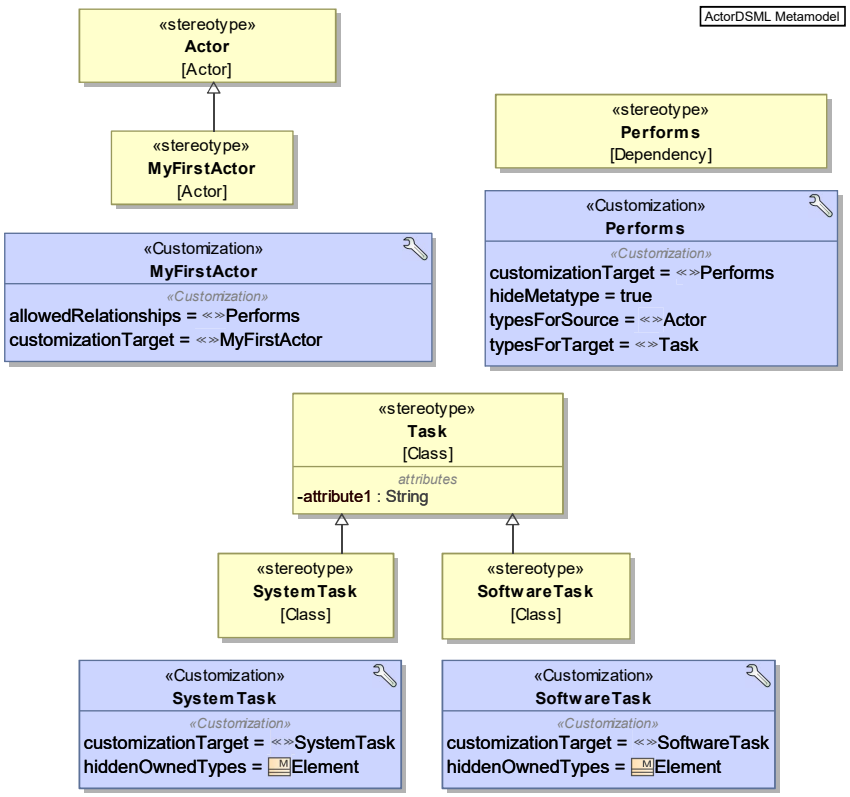


Figure 4.4: A MagicDraw metamodel for the actor task modelling language detailing stereotypes for `Actor`, `Task`, and their `Performs` relations. Figure adapted from [BGJ+23].

Figure 4.4 shows the MagicDraw metamodel for realising an ActorDSML. The main elements containing the stereotype definitions for the ActorDSML are the `Actor`, `Task`, and `Performs`. These elements are either directly customised or extended with special variants such as the `MyFirstActor`, which extends the general actor class and is also specified with an allowed relationship to the `Performs` association. Using this configuration, a *hook point* is established via the general stereotype for easily extending an actor into specialised actors. An arbitrary number of tasks (activities) are modelled using the stereotype `Task`, by referring to and changing the name of the attribute `attribute1`. Similar to actors, tasks are also specialised into further distinct types of tasks such as `SystemTask` or a `SoftwareTask`. In this metamodel, dependencies such as the `Performs` relation is configured such that it indicates the associations between

tasks and actors, as specified with the *typesForSource* and *typesForTarget* customisation properties. In general, the relations also apply to the extended actors and tasks as defined within these properties, meaning both the `Actor` or `Task` and their specialised stereotypes are also associated with the `Performs` relation.

4.2 Language Components in MontiCore

As discussed earlier in Section 2.2.3, MontiCore [HKR21] is an open-source[1] language workbench for the development of textual DSMLs. A MontiCore grammar describes the abstract and concrete syntax of a DSML in the form of an EBNF-based notation for CFGs which defines language infrastructure such as parsers, ASTs, symbol tables, and pretty printers. MontiCore provides symbol tables that cross-references to the AST, while the underlying infrastructure realises type checks as well as the different forms of language composition. Symbols correspond to the model element names, and their scopes, either artefact, global, or neither of them, define their visibility. An artefact scope describes the visibility of symbols for the entire model artefact, whereas the global scope describes the visibility of symbols across different model artefacts.

4.2.1 Concepts of Language Components in MontiCore

This section discusses the concepts of modular language components and analyses their applicability in the MontiCore ecosystem.

Requirements for Language Components

The flexible reuse of (potentially incomplete) languages as language components in the form of self-contained logical units is a requirement for language composition in MontiCore [But23]. An important feature is that language components are developed independently of each other, although potentially designed with extension in mind. Moreover, it is necessary to understand the relation between incorporated modelling languages, as their artefacts (i.e., grammar definitions and generated or hand-written Java classes) depend on each other. In MontiCore, a language component comprises a grammar to define the concrete and abstract syntax as well as additional validation and tooling, such as well-formedness rules, symbol management, and syntheses via pretty-printing or language-specific code generators. To enable their reuse, MontiCore provides design patterns for compositional language design [DJR22] for facilitating seamlessly integrating implemented functionality in a black-box fashion.

[1] MontiCore is available at: `https://monticore.github.io/monticore/`

Properties of Language Components

The language composition via language inheritance, extension, embedding, and aggregation in MontiCore relies on the symbol tables of the languages. The forms of language composition, except for language aggregation, are controlled via grammar inheritance, which is automatically solved by the generation of specific language infrastructure. In language inheritance, language extension, and language embedding, a composed language is aware of the other composed languages, whereas in language aggregation, the languages are unaware of each other.

4.2.2 Parts of a Language Component

This section describes the parts of a language component, specifically, the definition of a language in the MontiCore ecosystem along with the realisation of such language components.

Definition of a Language Component

A language component in MontiCore is defined by [BW21] as follows:

Definition 6 (Language Component in MontiCore)**.** *A language component in MontiCore is a reusable unit encapsulating a potentially incomplete language definition. A language definition comprises the realisation of syntax and semantics of a software language.*

This means that all the artefacts that help in realising a DSML are contained in a language component. By default, this language component definition does not distinguish between different kinds of artefacts that contribute specific parts to the language, such as a grammar file or a context condition class. Related language component definitions [BEH+20] often rely on such concrete artefact kinds (e.g., a grammar file) or describe a language in terms of the conceptual contributions that these artifacts make (e.g., abstract syntax). However, these definitions tend to be highly specific to only a certain technological space, and needs to be further generalised. Language composition in MontiCore is carried out by employing an artefact model [GHR17] that describes the complete implementation of a language, extracts the implementation of a language component as artefact data, and realises language composition for each artefact kind of the language individually. As an example, an artefact extractor determines which MontiCore grammar file(s) a language component uses and which parts of the language components are composed, using a composition operator for MontiCore grammar files. Further, the language component definition makes no assumption about a language component interface.

Realisation of Language Components

Language components are designed in a way that *hook points*, or *extension points*, exist within them that are used to refer to other languages and their components. For instance, productions in MontiCore that are marked as *external* serve as the hook points, while named nonterminals are inbuilt features even though they can be overwritten to adapt a language as well. Further, a language component comprises of well-formedness rules in the form of Java classes, a generated symbol table consisting of references to the symbols as file artefacts, and code generation functionalities. The realisation of *language component models* in MontiCore has been extensively discussed in [But23] and is out of scope of this thesis.

4.2.3 Forms of Language Composition in MontiCore

MontiCore supports language composition via language inheritance, extension, embedding, and aggregation [HKR21, But23] for integrating models or their constituents which is briefly discussed in this part of the thesis.

Language Inheritance

Concepts. Language inheritance in MontiCore follows grammar inheritance which means a language inherits from another language, if the grammar of the language inherits from that language. The grammar can reuse, extend, and override the nonterminals either by reusing the start rule of the inherited language or as a novel grammar rule. The mechanism in MontiCore is to separate the compositional parts of a language infrastructure into interfaces to avoid the problem of multiple inheritance in Java, which is the underlying GPL for MontiCore. For other parts of the language infrastructure, such as context conditions, MontiCore generates the respective context conditions interface for the nonterminals, meaning the classes implemented for the context conditions from the inherited language will be reused in the inheriting language.

Realisation. To realise language inheritance, MontiCore inherits the existing constructs such as the abstract and concrete syntax, the well-formedness rules, and the generated language infrastructure. The inheriting language can also override any existing productions and also exchange the start rule, thereby fostering reuse and extension.

Language Extension

Concepts. In language extension, a language adds novel parts to any reused language which are termed *conservative* [HKR21], meaning models belonging to the original language is still valid in the extended language.

Realisation. Extension of a nonterminal in MontiCore adds novel right-hand sides to a given grammar rule. This means language extension is achieved when the inheriting language reuses the start rule of an inherited language. An example of a conserva-

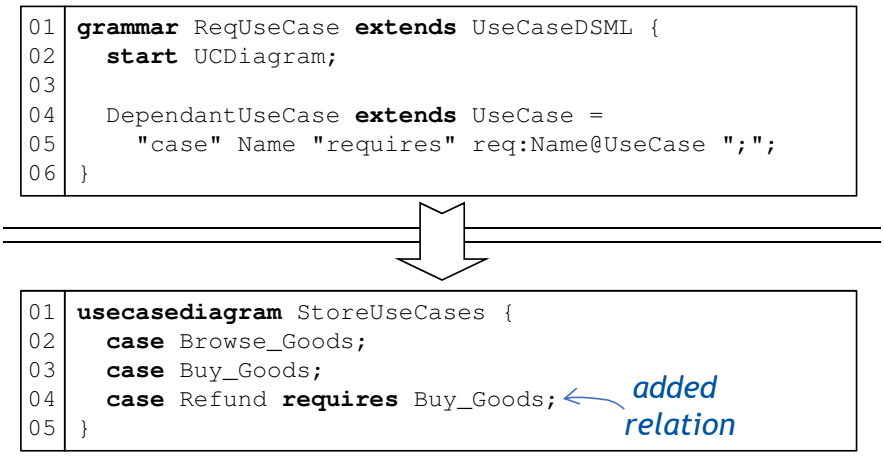


Figure 4.5: MontiCore grammars for an extended UseCaseDSML to enable additional relations between particular use cases. Figure adapted from [BGJ+23].

tive extension is shown in Figure 4.5 (top). The grammar `ReqUseCase` extends the UseCaseDSML and the starting nonterminals remain unchanged (l. 2). A specific use case (l. 4f.) extends the original use case thereby introducing an extended use case. In Figure 4.5 (bottom), a model that conforms to the extended language is shown that represents simple use cases for a store. Here production rules are defined, such as in (l. 4), that describes a `Refund` use case for which the `Buy_Goods` use case must also be part of the transaction.

Language Embedding

Concepts. Language embedding describes that a language Em is embedded into a host language Ho without the need for languages to be aware of each other's constructs. This is possible in textual language workbenches, such as MontiCore, via a novel grammar that inherits from both the Em and Ho grammars. The integration of the syntax of the two languages is done at a point, where one or more nonterminals of the language Ho are overridden, extended, or newly implemented with novel syntax. In this form of composition, the context conditions of the two languages are directly unified.

Realisation. In MontiCore, the realisation of language embedding is done using a common, unifying grammar, where multiple grammars are extended such that language definitions are jointly available. As an example, this is illustrated by integrating use

cases into the ActorDSML. Here, the definitions of the use cases remain unchanged and they are used similarly when embedded in the ActorDSML. To further establish any relations between the two languages' components, a relationship that allows actors to refer to use cases in a similar way to tasks is created as illustrated in Figure 4.6. The interfaces created as part of this mechanism serve as the predefined *hook points* to foster extension as well as embedding.

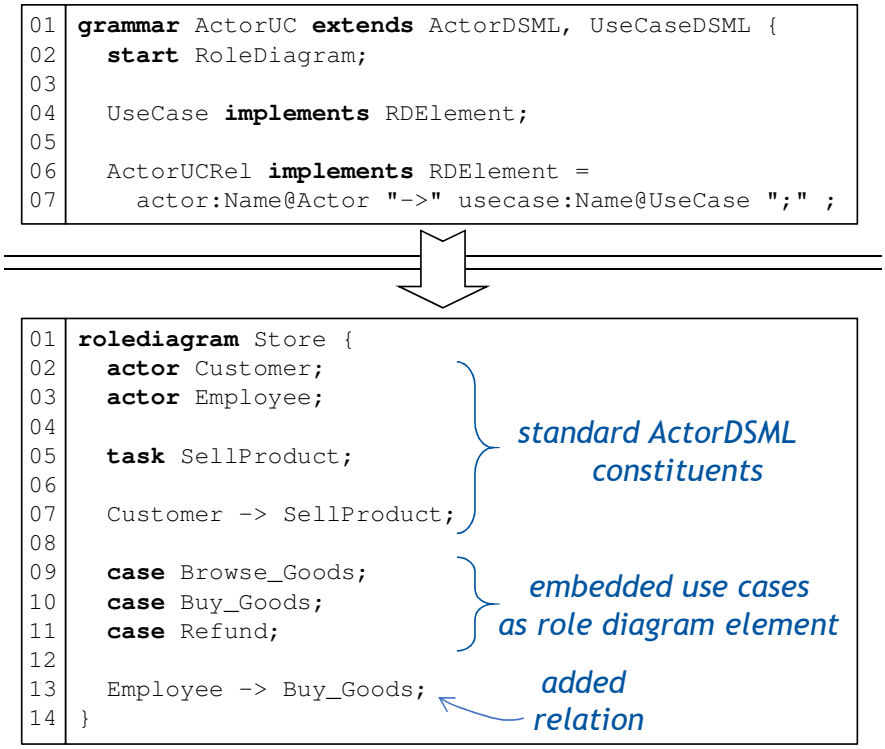


Figure 4.6: Embedding the UseCaseDSML into the ActorDSML, allowing relations between actors and use cases. Figure adapted from [BGJ+23].

Language Aggregation

Concepts. MontiCore differentiates language aggregation from the other forms of composition by not requiring integration of models of the languages into a single artefact. This means, there the coupling between the involved languages is only done via symbol exchange which allows modellers to refer to the model elements of the aggregated language using their names and the models remain independent.
Realisation. Language aggregation in MontiCore is achieved using the symbol table that stores unique identifiable elements resolved ultimately using their qualified names. A

shared grammar is used to achieve language aggregation wherein all models share the same symbol kinds and symbol table structure. A unifying grammar is used, alterna-

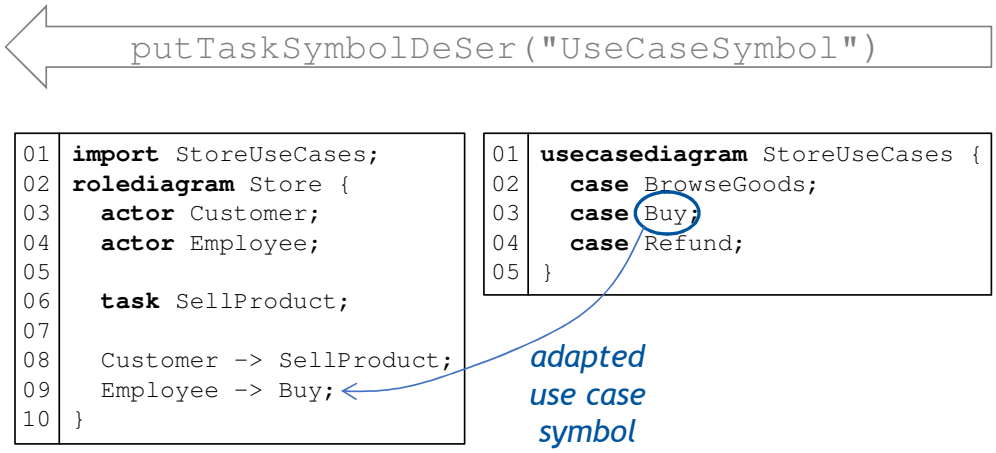


Figure 4.7: Adapting use cases as tasks by aggregating the UseCaseDSML and the ActorDSML. Figure adapted from [BGJ+23].

tively, where artefacts of models remain independent but is part of a common language definition with the symbol kinds integrated for both the languages. However, models may refer to other models of languages using defined associations. Such associations are defined on the composed language and allow providing the required relations between the languages being reused. Therefore in practice, this form of composition normally requires a certain level of knowledge of the individual languages such that a language engineer defines at which extension point an association must be established. In contrast to language embedding, language aggregation ensures that model elements only refer to other model elements at the specified extension point and is otherwise unaware of any other references or interfaces. Resolvers to derive symbols from the symbol table are used to interpret the symbols without a unifying grammar. Symbol tables of models are also persisted in files which require a certain kind of deserialisation of each symbol kind individually. Figure 4.7 show how the ActorDSML and UseCaseDSML are aggregated using symbol tables by adapting use case symbols to task symbols. This is achieved using deserialisers in MontiCore that load a symbol table into memory and translates the symbol kind such as with the provided `putTaskSymbolDeSer()` method. In this way, the models remain in their individual artefacts and are only referred to using their names, while a link from the actor *Employee* to the use case *Buy* is established in terms of a shared context.

4.3 Concepts of Language Components in MagicDraw

This section builds upon the concepts of modular language components described in the textual space of MontiCore and analyses their applicability in the graphical space of the MagicDraw ecosystem. In particular, modularity refers to the self-contained units that are distributable across various technological spaces, for example, as archive artefacts. In this regard, building blocks that are developed for a DSML are designed and used by language engineers independently but provide the necessary interfaces for their functionalities. Therefore, this chapter discusses in length the notions of modular language components to finally deduce a generalisable definition.

4.3.1 Requirements for Language Components

Certain requirements for language components in graphical DSML engineering must be met to support the flexible reuse of a (potentially incomplete) DSML definition to achieve modularity. Achieving reusability of language components [VV10], and therefore of the DSML and its building blocks, is crucial in understanding the concepts for providing a family of similar DSMLs that are composed using language composition techniques [EGR12, HLMSN+15]. This is particularly helpful in building a library of DSMLs and their components [BEH+20] that further eliminate designing versions of a language from scratch every time.

In this regard, MagicDraw provides a wide range of customisations and mechanisms (Section 3.4), helping language engineers define modular software artefacts that are reused as-is or in an enhanced format. This means that language components include artefacts of the (potentially incomplete) DSML definition. Here, we must note that in comparison to textual language workbenches such as MontiCore, the problem of ambiguous grammars is eliminated [BEK+19] as the language itself is not defined through a context-free grammar, which could introduce ambiguities such as with nonterminal name clashes [But23]. The likelihood of clashes between graphical language elements in MagicDraw belonging to different DSML building blocks is increased when such graphical language elements are assigned identical names. In MagicDraw, different language profiles contain identical named language elements, as the language element name resolution is done through a combination of the language profile and the stereotype definition name discussed in Section 3.4.1. However, MagicDraw also provides an in-built mechanism to prevent language engineers from defining language elements with the exact same name within the same profile, further reducing the likelihood of such clashes in different DSML building blocks. In addition, graphical artefacts that are exclusive to MagicDraw and are not considered part of language development, such as comments, notes, and text boxes, are also considered separately from language composition techniques, but they belong in the individual language components. Finally, the need for language components arise from integrating aspects of good user experience (Chapter 6) and techniques

for providing methods, recommendations, and guidance to users (Chapter 7), directly within a DSML building block.

4.3.2 Properties of Language Components

In theory, composing languages is not a property of languages. It only implies that any two languages are capable of being composed with a novel syntax and semantics that are valid for the composed language [EGR12]. The distinction here is that the composition of languages is a property of language definitions, which means that two language components work together without needing any modifications towards a mutual goal. This means a language component is itself used in the composition of a variant of a previously developed independent language. Therefore, it is important that language components are precisely defined. To easily compose new languages or versions of languages, such language components must be flexible to allow modifications and extensions such that languages are composed without being completely rewritten. To this end, a language component must provide sufficient *hook points* that act as interfaces for other language components. This is achieved using a syntactic interface that provides the necessary connections to the syntax, variables, names, or a technical interface that allows the connection of the behaviour of two or more language components.

Certain forms of language composition work differently [EGR12, Rum13], as we also discuss them later in this chapter. Language inheritance and language extension are considered import mechanisms, meaning a composed language is well aware of the DSML constructs of the parent language, but not the other way round [VV10]. In the case of language extension, any extensions to a parent language must be *conservative* (or safe) so that the off-chance that a language engineer exploit and incorrectly modify inherited classes are avoided [HKR21]. In language embedding, a language Em is embedded into a host language Ho without any prior knowledge of either languages. In the case of language aggregation, a loose form of conceptual coupling exists between two languages, wherein generally no new language infrastructure needs to be generated [But23] or at least one language needs to be modified to achieve language composition [GJRR22b]. Therefore, language composition is not only concerned with the graphical concrete syntax, but also to other parts of the language infrastructure such as abstract syntax, context conditions, and so on. In all scenarios, the actual language composition is performed when all the language components have been generated and are available to be used in a composition.

4.4 Parts of a Language Component in MagicDraw

This section describes the parts of a language component, specifically, the definition of a language component along with the realisation of a language component in the MagicDraw ecosystem.

4.4.1 Definition of the Language Component

To further understand how DSMLs and their building blocks are composed, the following definition of a *language component* in MagicDraw [BGJ+23] is used throughout the remainder of this thesis:

Definition 7 (Language component in MagicDraw)**.** *A language component in MagicDraw is a reusable unit consisting of artefacts that, entirely or in part, encapsulates a (potentially incomplete) graphical language definition. A graphical language definition in MagicDraw consists of the abstract syntax, the graphical concrete syntax, well-formedness rules (context conditions), and the semantics of a software language.*

A language component in MagicDraw, therefore, contains all the artefacts that are part of the complete graphical language definition or parts of the graphical language definition. This is further illustrated using an example in Section 4.1. All the software artefacts that are generated and belong to a language component are stored as files in a file system directory which is part of the MagicDraw compiled source code. These artefacts primarily consists of a language profile and its stereotype definitions (Section 3.4.1), as well as customised files necessary to represent the syntax and semantics of the language. These artefacts are then bundled together into the final DSML plugin that is installed for use in any instance of MagicDraw. The abstract syntax in MagicDraw is described using UML class diagrams (UML CDs), the graphical concrete syntax is described using language elements based on UML stereotypes, and the context conditions that check the well-formedness of a DSML's model is realised using Java classes.

In addition to these conventional artefacts, the tool MagicDraw itself is considered a language component. This is because the tool is an integrated development environment (IDE), meaning its has editor capabilities that support language development, and is comprised of a set of artefacts needed to define a language. These editing capabilities of MagicDraw help language engineers in describing not only the language definition, but also aspects related to user experience design, or custom guidance and methods for industrial DSML users in domains such as energy, healthcare, IT, and digital industries, to assist them in their modelling [MAGD+16, TK05]. For simplicity, we sometimes refer to language components as components in the remainder of the thesis.

4.4.2 Realisation of Language Components

Chapter 3 described MagicDraw's ability to provide editor capabilities to develop languages with sufficient customisations. This means a DSML in MagicDraw is composed for a specific domain using a very specific set of artefacts, as part of a language component. Further, by using the Open Java API [Neu06], *custom plugins* in MagicDraw enhance the DSML to support complex constructs that are generally not possible using the default MagicDraw capabilities. Figure 4.8 illustrates the artefacts that are archived

together into a single plugin file. A language component consists of the artefacts listed under "files (Directory)" in the figure. The language components are then composed together into the final DSML plugin file. The DSML is made available to the users as

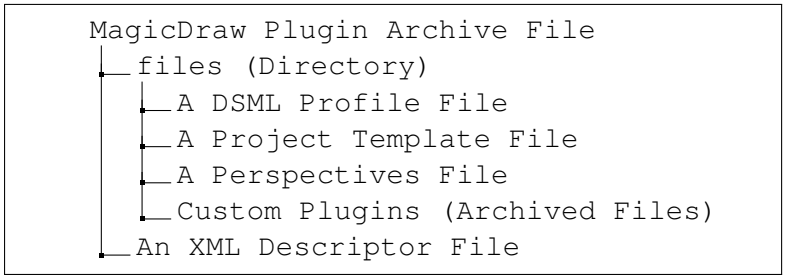


Figure 4.8: A MagicDraw DSML archived plugin file structure consisting of the artefacts that belong to individual language components.

soon as the plugin file is installed in a MagicDraw instance, and is ready for use at tool start-up. The following elaborates the concepts of industrial DSML engineering in MagicDraw in respect to the language components introduced in Section 3.4.

DSML Profile. The DSML profile of a MagicDraw plugin generally contains all the necessary artefacts required to complete a language definition. An example of an artefact in terms of language element contained within the DSML profile is identified as a *stereotype*, and configured with *customisation* properties, as well as relations that help in defining the behaviour of a language element. Figure 4.2 shows an example of a DSML that is used for modelling use cases. The `customizationTarget` property for an element specifies the configured stereotype. The `hiddenOwnedTypes` property for the element specifies which stereotypes (and their metaclasses) are hidden from display, such that no elements specific to that metaclass are created. Further, the language definition contains artefacts that allow the configuration of tables, diagrams, and matrices. These DSML elements are exported and stored as XML files in the DSML plugin. Validation rules are set on DSML elements and are located under the *activeValidationSuite*, for checking errors during design of models. These rules are configured as Java classes and the corresponding class files are also bundled together in the DSML.

Project Template. A project template in MagicDraw consists of a customised model representing the starting point in designing models using the DSML. It consists of a set of artefacts that are created by language engineers as part of the DSML and are instantiated when new modelling project is created by a user. In the case of MagicDraw, these predefined models are created directly on the graphical modelling canvas. This is particularly helpful designing complex systems as the users of such DSMLs may not be familiar with all the language concepts and such models avoid designing parts of the model always from scratch. The project template is itself also generated as a MagicDraw `.mdzip` file consisting of all the artefacts that belong to this component.

Perspectives. MagicDraw offers a large number of functionalities, such as diagram toolbars and context menu items, to assist users in their modelling. Perspectives in MagicDraw contain a set of artefacts in the form of *.umd* files. Internally, these are XML files storing information on which DSML constructs or tool functionalities are displayed for a specific user. Each part of the DSML, its building blocks, or the tool is adjusted by specifying exactly how a novice or an advanced user of the DSML should be able to see.

Custom Plugins. Modelling capabilities of MagicDraw and the DSML are enhanced using custom plugins. Such plugins consist of artefacts such as compiled Java files, packaged `jar` files, and any additional dependent files that must be used to enhance the DSML. For example, a *validation plugin* checks the DSML constructs against rules that cannot be simply performed directly by the tool. Other plugins, such as those that are capable of enhancing the look and feel of DSML elements during design time, consist of artefacts such as scalable vector image files or *.jpeg* files.

XML Descriptor File. The descriptor file contains the references to all the other artefacts that are described for the DSML constructs and helps in extracting and installing them to MagicDraw. In practice, the descriptor file itself is considered an artefact, and is bundled together with the final DSML plugin.

4.5 Forms of Language Composition in MagicDraw

MagicDraw supports language composition [EGR12, BMR22, Rum13, BEH+20] using the concepts of:

- language inheritance,
- language extension,
- language embedding, and
- language aggregation.

All of these forms of language composition rely on the composition of the artefacts of the language components into a single composed language.

4.5.1 Language Inheritance

Concepts. Inheritance relationship concepts defined in OOP are reused in general to graphical languages that are developed and maintained in MagicDraw. Language inheritance in MagicDraw is realised using the concepts of inheritance between the language elements defined in a metamodel. This means that a language inherits from one or more languages in the case when the components of a composed language inherently inherits from the individual languages. To this end, in MagicDraw, inheritance is realised at

the level of the individual classes, that are represented using the graphical notations of generalisation and specialisation via UML stereotypes. For representing domain-specific elements, this is also possible via customised UML stereotypes. One or more superclasses are therefore reused and extended to another subclass. In this way, the attributes, methods, and features of those superclasses are also naturally reused. Further, MagicDraw provides the functionalities to naturally compose the well-formedness rules in the form of Java context condition classes that are checked against the inherited language elements and are reused in the composed language. Therefore, multiple inheritance is achieved through those classes that implement an interface for providing the necessary inheritance.

Realisation. A language in MagicDraw is created by composing the artefacts of the individual language components, e.g., the stereotypes. Figure 4.2 shows a MagicDraw metamodel that illustrates the inheritance of language components for the UseCaseDSML. In this example, the specific classifiers of a `UseCase` is defined, namely the `MyFirstUseCase` and `MySecondUseCase` stereotypes. These classifiers naturally inherit the properties of the `UseCase` classifier, therefore all relevant configurations are automatically propagated through the inheritance chain. This means any new attributes configured for the inherited stereotype with a same name as the base language elements are internally distinguished using identifiers that are unique. This helps in eliminating problems of multiple inheritance [BEH+20]. To illustrate this as an example, if an instance of the `MySecondUseCase` is newly configured with an attribute `attribute1`, then `attribute1` of `MySecondUseCase` is configured via the *Properties* tab of the *Specification* window in MagicDraw. The difference to configuring `attribute1` belonging to an instance of `UseCase` is by the configuration of the same in the *Tags* section under the *Specification* window of the model element in MagicDraw. As discussed previously, the well-formedness rules in the form of context conditions for the `UseCase` stereotype are valid on all the instances of the specific classifiers, in this example, the `MyFirstUseCase` and `MySecondUseCase` stereotypes.

4.5.2 Language Extension

Concepts. Language extension is a specific form of language inheritance [BHH+17, BEK+19, HKR21, But23]. The principle concept in language extension is that a language extends another language. The language extension in itself has little meaning because of its dependence to the base language. This is primarily because the extension always requires adding novel parts to a base language that is being reused. This base language is reused without any modifications, which ensures that any subsequent extensions in MagicDraw remain safe, meaning the models of the base language continue to be valid in the extended language with the added novel parts. We therefore say that the extended language is a *conservative extension* [HKR21] of the base language since

the models in the base language also conform to the extended language. This means that there exists a rather strict form of coupling between the two languages, the base and the extended, giving it the primary characteristic for this form of language composition. Further, the remainder of the language infrastructure does not contain any distinct characteristics between a language composed via inheritance or via extension. For example, if a language component restricted the use of certain constructs in the base language, such as relationships on language elements, they continue to be automatically restricted in the extended language (elaborated previously in Section 4.1). Such kinds of language restrictions ensure that the restrictions imposed on the constructs of the base language hold for the extended language as well. Any well-formedness rules, i.e., context conditions, continue to naturally compose in the extended language.

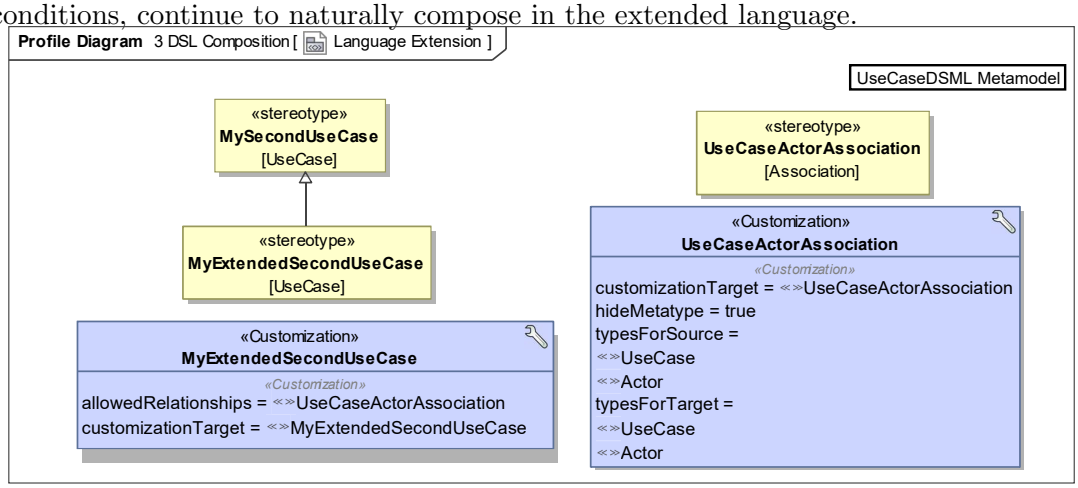


Figure 4.9: An example of the `MySecondUseCase` being extended and customised to `MyExtendedSecondUseCase` with an association dependency `UseCaseActorAssociation` between a `UseCase` and an `Actor`. Figure adapted from [BGJ+23].

Realisation. An example for configuring a language extension on a language component for the UseCaseDSML is shown in Figure 4.9. The `MyExtendedSecondUseCase` language element extends the `MySecondUseCase` element using a MagicDraw generalisation. This means the specialised `MyExtendedSecondUseCase` element automatically inherits the same stereotype metaclass configured for the `MySecondUseCase`, which is the `UseCase` metaclass. In principle however, the classifier for the `MyExtendedSecondUseCase` can be modified during the composition, although this could lead to undesired errors during modelling. In this example, the `MyExtendedSecondUseCase` element is configured additionally with a `UseCaseActorAssociation` as part of the allowed relationship, which is an association relation between a `UseCase` and an `Actor`, as seen on the right side of the image. Such an extension demonstrates how additional constructs of a language are composed without any modifications to the base language.

It must be noted that since `MySecondUseCase` stereotype disallows all kinds of relationships (Figure 4.2), the `UseCaseActorAssociation` relationship element is also inherently disallowed for any instance of the `MyExtendedSecondUseCase` stereotype. If this novel association would have been configured on the `MyFirstUseCase` element, then the association between instances of `MyFirstUseCase` and `Actor` would have been allowed. In the extending language, instantiation of `MyFirstUseCase` is still possible. A benefit of under-specifying the configuration of these language elements is that it allows for easily configuring relations in variants and instances of the constructs of the UseCaseDSML. However, language engineers may choose to restrict language elements in other scenarios to prevent undesired language functionalities further along the inheritance chain making this form of inheritance non-conservative. It would however be beneficial to only configure specific types of relations, instead of completely restricting all kinds of relationships, as allowing such a mechanism gives language engineers more flexibility in developing versions of a DSMLs easily, rather than completely changing many parts of the language infrastructure. Well-formedness rules for `MySecondUseCase` continue to be valid and apply naturally to the `MyExtendedSecondUseCase` element and new rules are defined as novel parts for the `MyExtendedSecondUseCase`.

4.5.3 Language Embedding

Concepts. While language inheritance and extension are based on the idea of reusing a single language for creating a new language, language embedding works on a principle that uses at least two languages. In this form of composition, one language is reused as the host language (Ho), whereas the other language is reused as an embedded language (Em). In doing so, a language, and therefore all of its constructs, are completely encapsulated into the host language Ho. Language engineers achieve this in MagicDraw by introducing a novel syntax called *integration glue*, which acts similar to an API and is used for communicating between specific constructs of the host language Ho and the embedded language Em. This allows any underlying connections and relations between language elements in Em to be interfaced directly with the constructs of Ho. In language embedding, the host language Ho is introduced with the novel syntax, while both of the languages, Ho and Em, are reused as-is. In MagicDraw, language embedding is realised using an in-built tooling property setting called *showPropertiesWhenNotAppliedLimitedByElementType*, which is defined as an attribute on the novel stereotype part of a host language's construct. When this attribute is initialised, the corresponding language element that is configured on this attribute, is embedded into the host language Ho. This means that the novel stereotype, or syntax, inherits both the host language Ho and the embedded language Em constructs, and the language element configured by the specified tooling property described above simply acts as the interface to glue both languages. As this novel syntax is configured, the metaclass is automatically configured as one or more of the classifiers that are inherited from both the host and the

embedded language element metaclasses. Here, we must note that inconsistencies between the classifiers occur due to the inherent redundancy in UML [EB04, LEM02]. To avoid this problem, language engineers must carefully select the final classifier for the novel syntax. Using these concepts, truly modular language composition is achieved, as only the necessary and generated implementations from both the host and embedded language infrastructures are completely reused. As with other forms of composition, the other parts of the language infrastructure compose naturally. However, if required new language infrastructure, such as new well-formedness rules or further extensions may be defined that is applicable on the composed language.

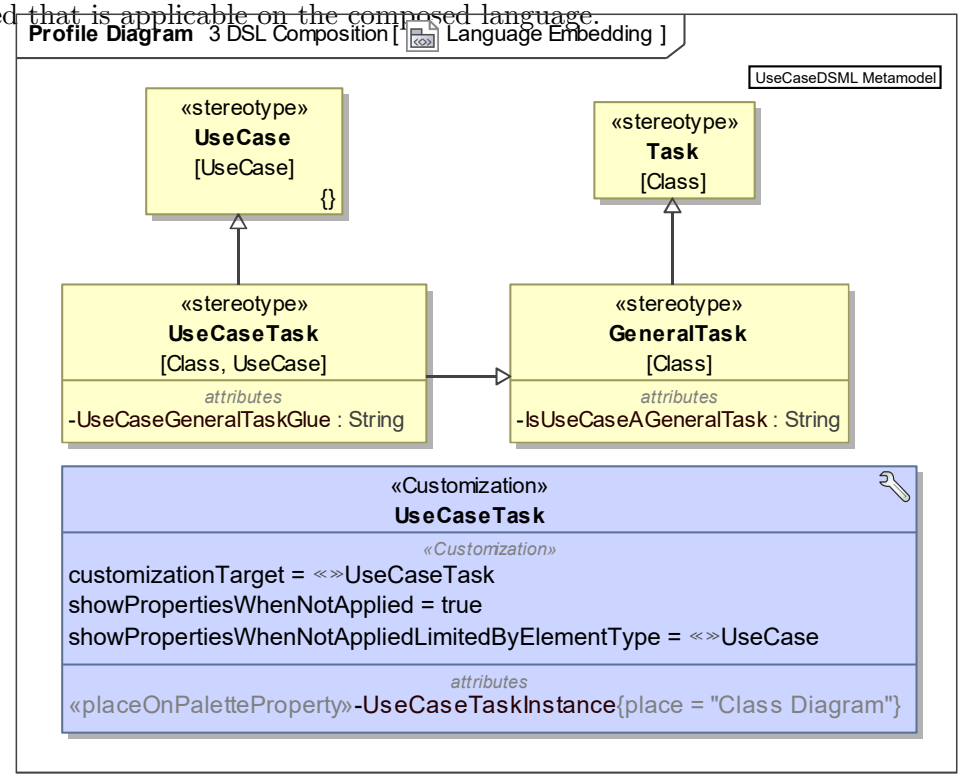


Figure 4.10: Embedding the ActorDSML into the UseCaseDSML via an integration glue (`UseCaseTask`) in MagicDraw. The `UseCaseTask` stereotype is the novel syntax specifying the `UseCaseGeneralTaskGlue` attribute as the integration glue on the `UseCase`. The ActorDSML is embedded through the integration glue attribute of the `UseCaseTask` stereotype into the UseCaseDSML using MagicDraw's *showPropertiesWhenNotAppliedLimitedByElementType* property. Figure adapted from [BGJ+23].

Realisation. The realisation of the configuration of a novel syntax for the composed language that embeds the ActorDSML into the UseCaseDSML is shown in Figure 4.10. The example illustrates the complete reuse of the UseCaseDSML and the ActorDSML.

In MagicDraw, these are set as read-only projects, meaning both languages are reused as-is without any further modifications. In this example, the `UseCaseTask` is the new stereotype that is created by a language engineer which ultimately inherits from the `UseCase` stereotype belonging to the UseCaseDSML host language and the `Task` stereotype belonging to the ActorDSML embedding language. Because of this inheritance, the `UseCaseTask` consists of both the metaclasses defined previously for the `UseCase` and `GeneralTask`, hence `Task`, stereotypes. In this situation, the `UseCaseTask` is either instantiated with a `UseCase` or a `Class` classifier. As the `GeneralTask` stereotype is a specialisation of a general `Task` it automatically inherits all the properties of the `Task` stereotype. The integration glue earlier mentioned is the `UseCaseGeneralTaskGlue` attribute part of the `UseCaseTask` stereotype and is configured to the `UseCase` stereotype using the property setting *showPropertiesWhenNotAppliedLimitedByElementType* for this novel syntax. By setting this attribute value, `UseCase` is now aware of the constructs of the ActorDSML enabling embedding. Conversely, the `IsUseCaseAGeneralTask` attribute on the `GeneralTask` is not configured to the UseCaseDSML, meaning constructs from both languages are unaware of each other and it cannot act as an integration glue. Therefore, for a UseCaseDSML to access the constructs of the ActorDSML, the integration glue is a necessity.

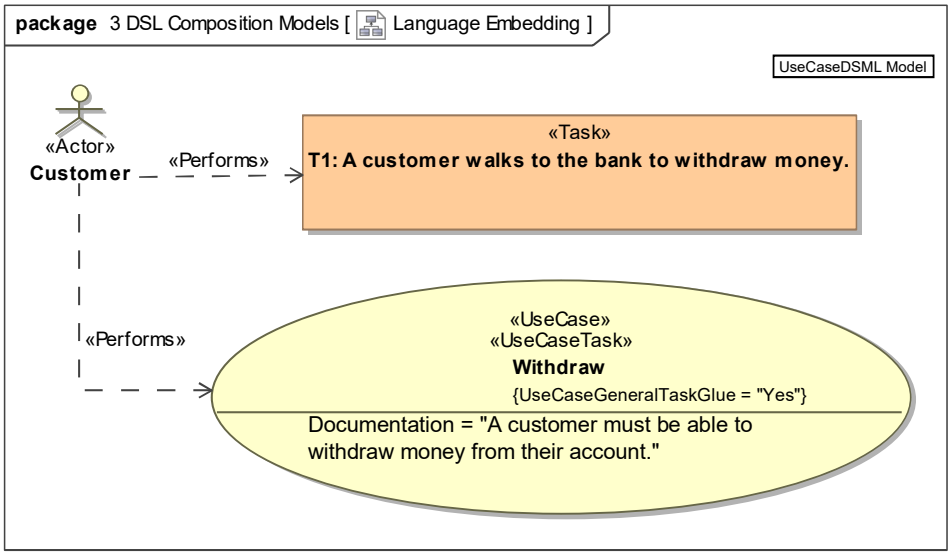


Figure 4.11: A MagicDraw model showing language embedding using an integration glue. The attribute `UseCaseGeneralTaskGlue` acts as the integration glue between the ActorDSML and the UseCaseDSML. As the integration glue is set to "Yes", *Withdraw* is configured both with a `UseCase` and the `UseCaseTask` stereotype, therefore embedding the ActorDSML into the UseCaseDSML, allowing a *Customer* to now configure an outgoing *Performs* relationship to *Withdraw*. Figure adapted from [BGJ+23].

To illustrate this with an example model, Figure 4.11 shows model elements from both the languages: *Customer* (`Actor`), *T1* (`Task`), and *Withdraw* (`UseCase`). At the start, a *Customer* is only allowed to perform *T1* and the *Withdraw* is configured only as a `UseCase`. This means, the *Withdraw* is unaware of the model elements *Customer* or *T1*, apart from being modelled on the same UML CD. As the user continues modelling, they realise that the description of the *Withdraw* is simple to be considered a task. The user therefore initialises the `UseCaseGeneralTaskGlue` attribute for the *Withdraw* to a value, e.g., "Yes". With this setting, the `UseCaseTask` is automatically added to the *Withdraw* meaning the constructs of the ActorDSML is now embedded into the UseCaseDSML, and the *Withdraw* is configured both as a use case and a task. As discussed previously, the integration glue (`UseCaseGeneralTaskGlue`) has created the interface required such that the user now further sets an outgoing *Performs* relationship from the *Customer* to the *Withdraw*, which was not possible without the setting of the integration glue. This is shown in the figure with the respective relationships configured. The integration glue therefore ensures the *Customer* performs both *T1* and *Withdraw*, without which the languages remain completely aware of each other. The well-formedness rules for the UseCaseDSML and the ActorDSML continue to remain valid independently, while adding new language infrastructure does not modify the individual languages.

4.5.4 Language Aggregation

Concepts. In contrast to language inheritance and language extension, and similar to language embedding, language aggregation is a form of composition which reuses at least two languages. Here, the respective models of the individual languages continue to remain in their own artefacts. While this separation holds overall, the models in MagicDraw may refer to other models belonging to other languages using association relationships. These associations are defined on the composed language so as to provide the necessary connections between the reused languages. This means that in practice, language aggregation requires a certain level of know-how between the individual languages. This know-how is configured by language engineers at different *extension points*, which denotes the location in a language where the association is being established. In general, associations implies that one language component (or a part of it) are connected to another language component (or a part of it). In language aggregation, the configurations are only such that the model elements refer to other model elements only at these specific extension points, and apart from these references, the existence of any other constructs, references, or interfaces remain unknown to either languages. A central benefit of language aggregation is that it requires no new language infrastructure to be generated. The individual languages are only connected via the extension points, that are associations between language components consisting of language elements. Similar to the other forms of composition, other parts of the language infrastructure, such as

the well-formedness rules, also naturally compose. A loose form of conceptual coupling between the languages is established when such extension points are created for all the languages under consideration.

Realisation. The configuration needed to achieve language aggregation in MagicDraw is shown in Figure 4.12. The ActorDSML and the UseCaseDSML, along with all their constructs, are reused in their entirety. To realise language aggregation in MagicDraw, a UML dependency is created as a novel syntax. In doing so, the language components in each language remain decoupled and are reused as-is without any modifications. Therefore, the ActorDSML and the UseCaseDSML are read-only projects in MagicDraw and the models for each language remain in their respective individual artefacts. In this example, the *TaskBelongsToUseCase* dependency is created as a novel syntax in the composed language *AggregatedDSML* for achieving language composition. This is possible as a dependency is implied by an association. For this association, the customisation properties *typesForSource* is set to the source language element, while the *typesForTarget* is set to the target, or the destination, language element. In this example, the source is configured to the `Task` and the target is configured to the `UseCase` (defined previously in Figure 4.10). In doing so, an outgoing dependency is only be initialised in the direction from an instance of a `Task` to an instance of a `UseCase`. A loose

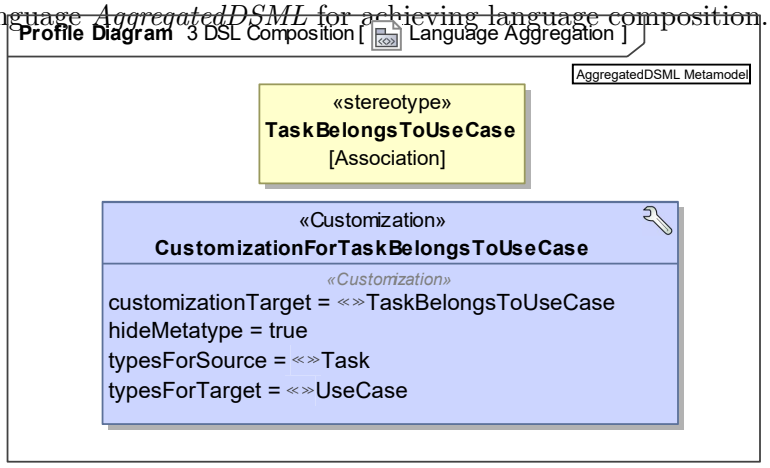


Figure 4.12: An example of the configuration for language composition using language aggregation in MagicDraw. The *TaskBelongsToUseCase* association is a novel syntax in the AggregatedDSML. The *typesForSource* and *typesForTarget* property configurations in the customisation of the novel syntax specifies the direction of the dependency between the models of the ActorDSML and the UseCaseDSML. In this configuration, the association is configured to exist from a `Task` to a `UseCase` model element. Similar to language embedding, both the base languages are completely reused unchanged in language aggregation. Figure adapted from [BGJ+23].

form of coupling is established between the languages, using bi-directionality, when both the source and target properties of the customisations are set to both the `Task` and the `UseCase` stereotypes. Constructs without any restrictions will then be known to either language. Further, this kind of configuration is done on the initial definition of the languages, which means the language engineer is aware of both the languages and accordingly configures the aggregation.

To illustrate this with an example model, Figure 4.13 shows model elements that were described earlier in Figure 4.11 for a *Customer* (`Actor`), *T1* (`Task`), and a *Withdraw* (`UseCase`). Once again, initially, a *Customer* is only allowed to perform *T1* and the *Withdraw* is configured only as a `UseCase`. During modelling, the user decides that

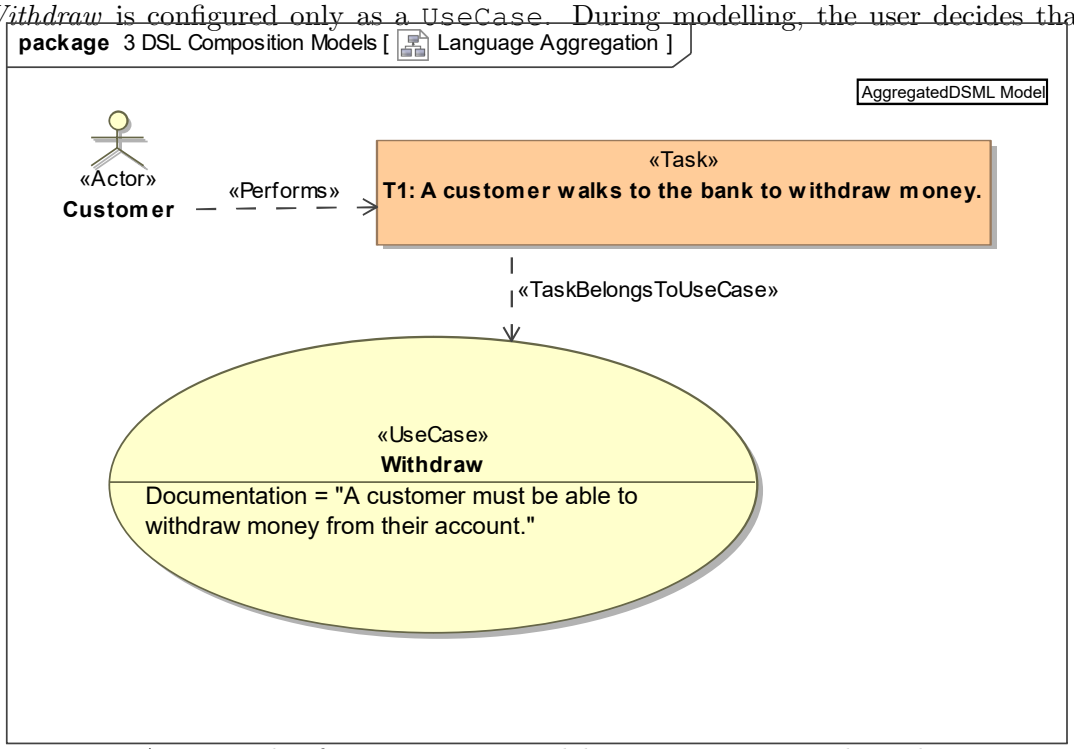


Figure 4.13: An example of a MagicDraw model using aggregation through association, *TaskBelongsToUseCase*, between the task *T1* and the use case *Withdraw*. *T1* and *Withdraw* are models in their individual DSMLs but modelled here on a common diagram. When the dependency is not set, *Withdraw* is completely unaware of either the *Customer* or *T1*. In this example, the direct dependency is only configured between *T1* and *Withdraw*. Figure adapted from [BGJ+23].

there could exist a list of tasks that belong to a single use case. Using the already configured *TaskBelongsToUseCase* association, the user creates the necessary outgoing relation from *T1* to *Withdraw*. This association creates the required aggregation be-

tween the ActorDSML and the UseCaseDSML. Now, as the *Customer* has a *Performs* dependency to *T1*, there exists no direct relation between the *Customer* and *Withdraw*. However, as the *TaskBelongsToUseCase* is already configured between *T1* and *Withdraw*, the element *Withdraw* now traces back to the *Customer* using the set of relations recorded between these model elements and derives the link between the two languages. In order to directly relate *Customer* and *Withdraw*, a new association must be created in the composed language, without which the languages are completely unaware of each other. Thus, when the languages remain unchanged, the models in Figure 4.13 exist independently. The well-formedness rules for the UseCaseDSML and the ActorDSML continue to remain valid independently, and new language infrastructure are created as desired.

4.6 Unified Concepts for Language Components in MagicDraw and MontiCore

In this chapter, we have so far discussed in detail the definitions of language components in the MagicDraw and MontiCore ecosystems that belong to the graphical and textual modelling spaces respectively. While, these exist independent of each other, describing cross-cutting concerns that enable a seamless unification of these language components is crucial in fostering wider adoption of such composition mechanisms and is interoperable irrespective of any technological space, modelling tool, or language workbench. As observed from the definitions, language components are principally quite similar in nature, therefore the concerns are to address reusing these definitions, both in syntax and in semantics. To this end, the main difference observed in the definition of language components is the specific format of the software artefacts. MontiCore primarily focusses on grammar files to generate artefacts for a DSML, while in MagicDraw, the language and its components are defined directly on the modelling tool and the file system artefacts that are later bundled into a single plugin file.

4.6.1 Mutual Definition of a Language Component

In general, a language component consists of reusable units, in the form of software artefacts, that encapsulate a language definition, either wholly or partially. These software artefacts could be, but not limited to, either grammar or java files in MontiCore to define textual languages, or as files in a file system directory in MagicDraw to define graphical languages. To this end, we use the definition of a language component independent of a technological space described earlier in Def. 2 as the mutual definition of a language component.

This mutual definition, therefore, does not differentiate between the software artefacts that belong to different technological spaces. The definition considers realising software

artefacts in general as part of the language definition, and is persisted as files, including any generated source code, in a file system directory. The definition of a language is typically facilitated with additional artefacts or applications, such as the modelling tool or language workbench itself, and therefore they are also considered parts of a language component. Mutual notions for language composition using such language components must also be considered for unifying the concepts across various technological space.

4.6.2 Mutual Notions of Language Composition

Based on the language composition techniques discussed so far in this chapter (cf. Section 4.5 and Section 4.2), mutual notions of such composition techniques are described.

Both **language inheritance** and **language extension** mechanisms rely on inheritance concepts that are used widely in the OOP world, and are therefore quite closely related. Reuse, extension, and overriding for all language components consisting of software artefacts such as language elements in a language definition must be supported by language inheritance. This includes nonterminals in a textual DSML and stereotypes in a graphical DSML. In language extension, a composed language must add novel parts to a language being reused. The models of the original language, with language inheritance, may not necessarily be valid in the resulting language, but continue to remain valid in the original language. In contrast to language inheritance, language extension conserves the properties of the inheritance mechanism, meaning only new, optional constituents are added as part of an extension mechanism, hence the notion of conservation extension.

Language inheritance and language extension reuses a single language. However, **language embedding** reuses at least two languages, a host Ho language and another language Em to be embedded. An extension of the involved language forms the basis for language embedding. This is usually achieved with an integration glue, that is a novel syntax introduced in the composed language, which inherits and extends the syntax of both Ho and Em languages. Normally, the language Em is embedded directly into Ho at some predefined *extension points* that allows the constructs of the embedded language Em to be visible to the host language Ho. This is also referred to as the *integration glue*. These extension points are normally configured as part of Ho's specification that provides the extension for the constituents of Em. This means language embedding realises and implements extension points within the overall composition mechanism.

Language aggregation is similar to language embedding in the sense that it also reuses at least two languages to compose a new language. In this form of composition, the models continue to remain in their individual artefacts. In theory, the languages are reused completely, however certain modifications could be made to one of the existing languages in a way that one of the languages refer to the constructs of the other language. In this scenario, a reconfiguration and regeneration of the infrastructure for one of the languages is required. Therefore, a reference must exist between the languages, for example, via a symbol names for textual DSMLs or via association relations in graph-

ical DSMLs. Such references establishes the necessary interface between the accessible constructs of the respective languages. This leads to languages exhibiting a rather loose form of conceptual coupling.

Properties. For individual languages to be bundled as part of a library of languages, it is important for language components of such languages to be *reusable*. Building such a library of languages is beneficial as it provides the opportunities to develop complex versions of languages based on extensions and adaptations, hence being *flexible*. Some forms of language composition describes building language components with *extension points* that as interfaces to refer to the constructs of other languages. A logical grouping of individual language components ensures that similar language components are reused in a language definition. Further, the realization of language components must be *under-specified* by language engineers, wherever possible, such that they avoid problems of language restriction later on during language extensions.

4.7 Discussion

This chapter describes the notions of language components and their compositions in the MagicDraw and the MontiCore ecosystems. However, to define and implement such composition techniques that are independent of technological spaces is a complex undertaking. While this chapter aims to provide a rather unified notion of language composition, the implementation may not necessarily be the "go-to" solution. Therefore, the concepts described in this section provide a kind of guidance for further work on language components and composition techniques for current and future tools and language workbenches, including those that support a mix of textual and graphical modelling. To this end, it is important for language engineers to be aware of the various concepts and realisation techniques that are required to compose languages or versions of languages.

To avoid problems of maintaining and reusing large software artefacts, language engineers are encouraged to design solutions that make use of modularity techniques. Therefore, there is a large interest in the SLE and CBSE communities to introduce reuse through modularity for existing and upcoming software solutions. Software components within these software consist of interfaces that provide the required interaction of the component to its environment [BHP+98]. The approach presented in this chapter does not promulgate explicit interface or interface providers for language components as they are generally considered an overhead in maintenance and evolution. This is because such interfaces require anticipating extension points that may not necessarily lead to better reusability of language components. The implementation of such interfaces is also a challenge as it could introduce undesired inconsistencies from the interface to the respective software components. This is because without a substantial knowledge of the workings of a language component, it is hard for language engineers to reuse language components

based on its properties. Therefore, this chapter discusses the composition of languages by introducing novel syntax and reusing mechanisms of extensions. To design languages without needing to know the internals of a language, their language components must be underspecified so that future extensions are possible. This kind of a black-box development helps language engineers in designing language components that are beneficial in being independently reused for developing complex versions of languages.

Language components typically consist of software artefacts that are generated and stored in specific file formats in different technological spaces. These artefacts are generated as part of the compiled source code during the definition of a DSML or could simply be a grammar file that is parsed to generate the language infrastructure such as an AST. The modelling tool or the language workbench is itself considered a language component, as it contributes to the language definition. However, they may be independent of consideration to the concepts of language composition. In all the discussed forms of language composition, at least one language is reused. While language inheritance and extension reuses one language, language embedding and aggregation reuses at least two languages. A central property of all forms of composition described in this chapter is that they foster the reusability of independent language components.

The concepts described in this chapter sets the foundation for describing language composition aspects in a hybrid modelling environment that uses a combination of textual and graphical representations, such as with projectional editing. MontiCore, a textual language workbench, is used to derive proof-of-concepts (PoCs) for a variety of domains such as with cyber-physical system (CPS) [ZPK+11, KRRvW17], automotive industry [HRRW12, KKR19], and Internet-of-Things (IoT) [KMR+20, BDJ+22]. The concepts and results have been actively used in many successful industrial projects, an overview of which is found at the end of this thesis. Similarly, MagicDraw has been used to design graphical DSMLs for many industrial scenarios and is widely used for practical modelling by practitioners. Mutually defining language composition aspects between these two technological tools and spaces creates an in-depth complement of research in academia and their applications in the industry. These aspects are described in a way that they are generally applicable to any modelling tool or language workbench that supports language development.

Principally, the language components do not differentiate between the artefacts that belong to their individual technological space. These software artefacts are realised as files in a file system directory, which are generated as part of the compiled source code. In the sense of mutually describing the notions of language composition, language inheritance reuses a single language while providing the necessary concepts to reuse, extend, and override language elements. Novel parts are added to a language infrastructure to achieve language extension. The valid models continue to exist in their respective original artefacts. In language embedding, at least two languages are reused, but are glued together at a single point in the host language that embeds another language. This kind of integration glue is necessary to extend the host language with the visible constructs

of the embedded language. This is achieved using unifying grammars in MontiCore, or using a novel integration glue with inheritance and extensions in MagicDraw. Language aggregation generally requires modifications on one of the multiple languages being reused, but it requires language engineers to reconfigure and regenerate the existing language infrastructure for a reference to be established. Symbol tables in MontiCore and association relations in MagicDraw are used to technically realise language aggregation in the textual and graphical spaces respectively.

To create a library of languages, it is important to engineer reusable language components. Therefore, these language components must not be strictly specified, to avoid restrictions on developing new language constructs, such as by disallowing relationships for a language element where the extensions for this element can no longer configure any further relations. The examples of UseCaseDSML and ActorDSML demonstrate the applicability and implementation of the concepts described in this chapter. Various methodologies to realise these concepts exist in the literature as described in the next section, however describing unified concepts of language composition from the textual and graphical modelling spaces serves as an important step in further developing component-based independent language composition techniques in the large.

Threats to Validity Certain threats to validity for these mutual notions must be considered for independent language composition to ensure the concepts are free from systematic errors or biases. This section details the concepts of language components using two actively maintained, advanced, and OMG standard-compliant ecosystems. MontiCore has been developed in academia for engineering textual DSMLs and has been used in the industry as well [HKR21]. On the other hand, MagicDraw is a popular commercial graphical modelling tool that supports UML and SysML standards, also has support for other frameworks and tools such as SysML, UAF, SoaML, and Enterprise Architect in the form of plugins [Mag20]. Therefore, the implementation details presented in this chapter are considered tool-specific or vendor-locked as they are presented in the technological space of MagicDraw and MontiCore.

Compositional reusable language design has been discussed in the literature independently in MontiCore [BEH+20] and MagicDraw [GKR+21]. Often various workbenches and tools do not provide a similar set of modelling functionalities thereby making the realisation of this study difficult in those ecosystems. Although the example languages of UseCaseDSML and ActorDSML are not complex, they cover the distinct language composition concepts described in this chapter. Further, these DSMLs, or parts of them, are commonly used in real-world projects as well [TG15] as they are based on UML and SysML constructs. While both of these languages are constructed independently and represent distinct domains, the composition of both shows how they offer integrated modelling techniques. It should also be noted that in very complex scenarios, all the individual language components may not be completely reusable, therefore a large portion

of the language infrastructure may be unused. While this chapter does not explicitly consider projectional editing, the threat can be ignored as projectional editing mostly combines the textual and graphical concepts described in this chapter. Some of the practical implementations already detail the research of language components and building blocks such as in the SpesML MBSE project [GJRR22b].

4.8 Related Work

To maintain and evolve complex software and systems, reusable units that capture system details are investigated in related approaches. One such modular approach for MBSE in complex heterogeneous systems is discussed in [HKR+09]. Aspects of compositional modelling for interactive systems is discussed in [BR07, RW18]. When describing a DSML, their software components, the corresponding artefacts, and the models must consider compositional approaches [TAB+21, BEH+20]. Many of the approaches present in the literature are described in the textual space [BHRW21, BPRW20, HRW18]. Certain forms of language composition have been widely discussed in the modelling community [EGR12, VV10]. While language related patterns have been studied [Spi01, MHS05], there is a general lack of identifying which language components are related to which patterns. Language implementations are reused and combined to create new DSMLs [ŞvdBV18, Val10]. However, such concepts of reuse introduces challenges in how the language is specified and how restrictions on certain forms of language composition are beneficial in globalising DSMLs [CCF+15, CBCR15]. This chapter presents comprehensive mechanisms of language composition in the graphical and textual technological spaces for language engineers to develop reusable language components.

This chapter presents a single definition for a language component valid across different technological spaces. This is important as language component definitions are almost exclusively present in the textual modelling space [But23, BEK+19, HLMSN+15]. In this space, the systematic engineering of languages as well as its implementation in practice is part of related approaches [SRVK10, KRV06, MCGDL12]. Similarly, a systematic development of graphical DSMLs in industrial scenarios has been studied [GJRR22b]. Modularising the compositional concepts with the JetBrains Meta Programming System (MPS) in the projectional editing space has also been researched [Voe11]. The integration of a textual syntax inside a graphical modelling tool is discussed in [DJRS22, Sei14], but they are hard to develop and maintain and only provide a loose relation to the concepts described in this chapter. One example is embedding the eclipse modelling framework (EMF), based on textual editors, and the eclipse graphical modelling framework (GMF), for developing graphical editors based on EMF as well as tree-based editors generated within the EMF [Sch08]. Model transformations that provide a textual and graphical notation have also been proposed [MSA+15, EvdB10], that illustrate the transformation of language concepts during design time. In contrast, the concepts described in this

chapter can be interchangeably used in the textual and graphical spaces. Further, a model can be projected in a combination of textual and graphical spaces [Voe10], which is desired if true modularity of language components needs to be achieved.

Language composition mechanisms described by [EGR12] in the textual technological space serves as the key related work for this chapter. Their described mechanisms are: "language extension, language restriction, language unification, self-extension, and extension composition". Similar to the concepts presented in this chapter, their notion of "language extension" also requires the reuse of a base language completely unchanged. This kind of composition also subsumes language restrictions prohibiting the use of certain language constructs. Their notion of "language unification" fits our notion of language aggregation, with two completely independent languages are reused unchanged with the introduction of a *glue code*. However, our notion of language embedding introduces an integration glue for embedding one language completely unchanged into another language at predefined extension points, thus relating somewhat to their idea of "self-extension". The notions of "extension composition" described in their work is supported by both MontiCore and MagicDraw through incremental extensions. One important distinction to their work is that the notions of language composition presented in this chapter is primarily extended to the graphical technological space. A conceptual framework for language composition in the graphical technological space would thereby foster the modular development of reusable language components.

Graphical modelling tools such as IBM Engineering Systems Design Rhapsody [IBM23], Enterprise Architect [Ent23], and MetaEdit+ [TR03] support language development concepts needed for reusing language components [Kha09, Ozk19]. On the other hand, textual language workbenches such as MontiCore [HKR21], Spoofax [KV10], EMF-Text [HJK+09], and Xtext [EV06] support textual modelling and have also been studied for modular language development [EvdSV+13]. In the context of language components presented in this thesis, the primary tools considered are MagicDraw and MontiCore.

In pattern-based modelling [DJR22, BGdL10], the formalisation of a pattern to provide compositional aspects by analysing conflicts is discussed. Certain templates to map concepts with their metamodels in model transformations discuss reusability of models [SCGL11]. Multi-level modelling paradigms [LGC14] help separate the concerns of modelling by segregating the various stages of modelling for complex language development. Language composition in textual modelling is achieved using a kind-typed symbol management infrastructure [BMR22] for identifying symbols, while the reusability of graphical language components in industrial DSMLs has been briefly discussed in the literature [GKR+21].

Chapter 5

Exchange of Language Constructs between Modelling Tools

This chapter discusses enabling the composition of a DSML and its constructs between different modelling environments. Interchanging DSML constructs between modelling tools or language workbenches is imperative in ensuring domain-specific constructs such as language components, consisting of reusable software artefacts, are reused as-is across modelling tools and language workbenches that support modelling in the textual, graphical, or projectional technological spaces. By defining such an exchange, individual units of the DSML, including components belonging to the DSML building blocks or the entire DSML building block itself, are reused as-is across technological spaces. This means language engineers and DSML users benefit from a single domain-specific modelling solution across different modelling environments without the need for building similar language constructs from scratch every time. Some results of this chapter have been published in [GBJ+24]. Therefore, passages from the paper may have been quoted verbatim in this chapter.

As domain concepts increasingly grow in complexity, there is also a growing need for the simultaneous exchange of domain-specific information between various stakeholders across modelling projects. Since such stakeholders often use a variety of modelling tools, it is necessary to provide techniques to develop reusable DSMLs. Further, stakeholders continue to use the same modelling tools relevant to develop and use their DSMLs. Therefore, there is little thought for generalising these domain-specific concepts across modelling tools or language workbenches. In Chapter 4, the composition of languages in the textual space of MontiCore and the graphical space of MagicDraw was discussed. This was demonstrated by writing grammars for textual languages in MontiCore and by defining stereotypes in the graphical language profile of MagicDraw. To further extend and validate these concepts of language components and their composition, it is necessary to describe these concepts in other modelling environments. As this thesis is aimed at primarily providing the means to engineer graphical DSMLs in industrial contexts, this chapter therefore extends the concepts of language design described so far to another commercial graphical modelling tool, Enterprise Architect (EA). Precisely, this chapter describes a bidirectional exchange mechanism between EA and MagicDraw, two com-

mercially established modelling tools by exchanging DSML constructs, that serves as the basis for generalising DSML exchange between different modelling tools. This mechanism allows individually created DSML components to be seamlessly exchanged, thereby fostering tool interoperability. As DSMLs represent domain-specific characteristics within a single area, the proposed mechanism evaluates the applicability of exchanging DSMLs and their constructs by extracting and translating these characteristics across modelling tools. This is demonstrated using a simple case study. To achieve this, DSML artefacts from a modelling environment are transformed into formats that are subsumed under the language definitions in the other modelling environment. Later in this chapter, we discuss how this mechanism is able to extend to established modelling frameworks and language workbenches other than EA and MagicDraw.

The use of models in designing complex systems help various stakeholders in understanding possible solutions through abstractions [Sel03]. Thus, stakeholders of modelling projects must consider the entire software or systems product lifecycle to deal with the rising challenges in such multi and inter-disciplinary projects. Various modelling tools and frameworks, such as MagicDraw [Mag20], Rational Rhapsody [IBM23], and EA [Ent23] are therefore used to engineer various parts of a model within each modelling project [OWH+21]. Today, the exchange of DSML constructs between such proprietary modelling tools, like MagicDraw and EA, is still challenging, especially for small and medium businesses. These modelling tools have also been used in the systems engineering community and are well established. Further, the need for exchanging DSML constructs has also been discussed in real world industrial projects [BCC+10]. The mechanism described in this chapter allows DSML artefacts to be exchanged between modelling tools, where the language definition is transformed into a format that is reused by the other tool. A benefit of using this mechanism is that all stakeholders equally take advantage of the availability and interchange of the DSML and its constructs across various modelling tools, projects, and even across all stakeholders in an organisation. To realise the implementation of this mechanism, a GPL, such as Java code is used, that enhances the existing and default functionalities of the respective modelling tools. Language engineers utilise this mechanism to foster the development of DSMLs across different modelling environments. In the following sub-sections, methods against the mentioned modelling tools of MagicDraw and EA is described. A demonstration of how this approach is applied to modelling tools needing a similar DSML exchange is also discussed. To summarise, this chapter presents tool interoperability and consistent DSML development viewpoints between all kinds of language engineers, novice or advanced, and across inter-disciplinary modelling projects.

The interoperability of DSML constructs requires a generic set of bridges to be established between different tools. Such bridges must ensure that both metadata and behaviour interchange leads to tool and language evolution such that stakeholders work on different viewpoints in a collaborative manner. Metamodels, i.e., the internal schema of the language or the tool, must not be assumed to be conforming to a fixed metamodel

such as with UML metamodels. Instead, considerations must be made for representing domain-specific constructs with different metamodels [BCC+10]. Thus, manipulating different metamodels in a way that they are correctly interpreted in a different modelling environment is imperative for a bi-directional exchange mechanism to work in a consistent manner. In the following, Section 5.1 describes the DSML specification for EA and MagicDraw modelling tools before we discuss a general concept for a DSML exchange between modelling tools in Section 5.2. Section 5.3 discusses the implementation of the exchange mechanism. Section 5.4 demonstrates the applicability of the approach. Section 5.6 discusses the approach and Section 5.7 explains related work.

5.1 DSML Specification in Enterprise Architect and MagicDraw

This section describes the DSML specifications of EA and MagicDraw that are used for demonstrating the exchange of DSMLs and their constructs.

5.1.1 Defining a DSML in Enterprise Architect

In the modelling tool EA, a DSML is realised using model-driven generation (MDG) technologies that encapsulate a logical collection of resources such as UML profiles [Mod23b]. In the remainder of this chapter, we refer to MDG technologies that is specific to EA and represents an individual modelling notation. A DSML in EA is realised through extensions to the UML profiles for representing the domain-specific elements and its interconnections. Thus the underlying metamodel inherits all the necessary elements required for the desired application domain. The metamodel also shows the necessary relationships in correlation with the actual real world elements. The creation of these metamodels is out of scope of this thesis, and so the focus is on the implementation of a DSML using the MDG technology in EA. EA's modelling functionalities are extended to allow domain-specific concepts using the MDG technology's profile capabilities, that allows for the creation of a set of model constructs. The extension is achieved using the following three main profiles:

- **UML Profile**: The UML Profile for a DSML consists of the relevant domain-specific elements and their respective UML metaclasses and relationships. The stereotypes are attached to the elements and their associated tagged values allows the element attributes to be configured either as a set or enumerations.
- **Diagram Profile**: The Diagram Profile for a DSML allows for the creation of specific types of diagrams in EA. Here, the diagram represents a DSML diagram, extended from a UML diagram, that is mapped onto the EA toolbox. This ensures that each diagram only allows models to be created with a certain set of elements that are relevant in describing the constructs of the domain.

- **Toolbox Profile**: As part of the Toolbox Profile, all UML elements are logically grouped together to ensure consistency among different elements. According to the modelling situation, only a particular group of elements are therefore displayed for modelling. Here, there is a distinction between the grouping of the elements and their relationships that distinguishes between logically separate elements. In this profile, all the elements not part of the UML Profile are selected, including the DSML elements.

Apart from these profiles, additional information are also stored in the MDG file. Certain model elements are visualised graphically using relevant images or additional attributes. Since this information is not part of the UML Profile, they are additionally extracted from the respective elements and stored in the MDG file in an XML format. The individual parts of the MDG file is then combined and stored in a single exportable file, that includes the UML Profile, the Diagram Profile, and the Toolbox Profile as well as all the additional information. The creation of such an exportable file allows the DSML constructs to be exchanged with other instances of the tool as well as being deployed to other stakeholders, and therefore even to other modelling tools or frameworks.

Finally, the exportable file is located within the MDG Technologies folder of the current EA instance, and follows the structure described in Figure 5.1. Any other DSMLs or their constructs that have been referenced are stored in the same folder for provisioning and final deployment. Versioning of such DSMLs is achieved by maintaining different folders each containing different DSML versions. This includes constructs that implement the Zachman framework [Zac03] for providing a formal and structured way of defining an enterprise. This ensures that all the information related to the DSML is loaded directly in EA during tool start-up and no further configurations are needed. Any diagrams part of the DSML that consist of the model elements are directly added using EA's context menu.

5.1.2 Defining a DSML in MagicDraw

Similar to EA, MagicDraw is also primarily based on UML and provides customisations such as extensions for modelling with SysML. Previously in Chapter 3, we discussed how MagicDraw also provides a extensive list of customisation possibilities so that the modelling goals of DSML users are achieved through an enhanced modelling experience. Such customisations are used to create modular and reusable units of DSML building blocks, which is important in achieving interoperability of DSMLs and their constructs. To this end, MagicDraw offers various possibilities to customise a language that allows for the creation of language component artefacts. One such example of an artefact is a language element which is eventually referred to as a *stereotype* in MagicDraw and corresponds to the same in EA. For each individual stereotype, a set of rules are configured that defines its semantic behaviour in a domain context. These customisation

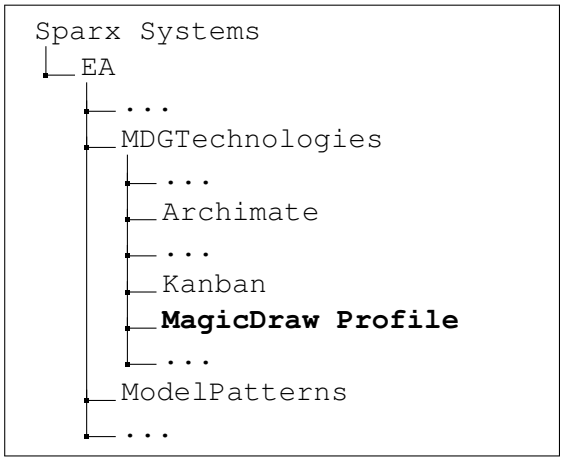


Figure 5.1: An EA DSML plugin hierarchy with the different artefacts.

rules, therefore, allow language engineers to integrate various facets of language development using Java Open API, from automation to creation of functionalities that is not supported by MagicDraw out-of-the-box.

In comparison to EA, the following extension mechanisms are used to ensure language artefacts are created in a similar fashion contained within XML files thereby allowing easy export of the DSML and its constructs to other modelling tools. Model templates are a way to automatically create predefined models in MagicDraw that is instantiated during design time. Additional information that help display tool and DSML functionalities are configured as perspectives in MagicDraw. Further, the combination of UML and domain-specific diagrams are typically bundled in a single file that is used across other MagicDraw instances. The language profile, that consists of the definition of the language, and its additional information is exported as an XML file, which is then reused across other projects and is therefore reused in other modelling tools as well. Similar to EA's mechanism, the language artefacts are loaded in-memory to ensure that during tool start-up, the DSML and its constructs is readily accessible to modellers. The structure of this archive file was described earlier in Figure 4.8.

5.2 Concept for a DSML Exchange Mechanism

We discussed the capabilities of both tools to show how language engineers design DSMLs in graphical modelling tools as well as set the precedent for discussing how the customisation capabilities of such tools be utilised to derive a generic concept for the transformation of DSMLs between them.

Figure 5.2 shows the general concept for interchanging DSML elements between two modelling tools. The concept is described in a way it is independent of the modelling

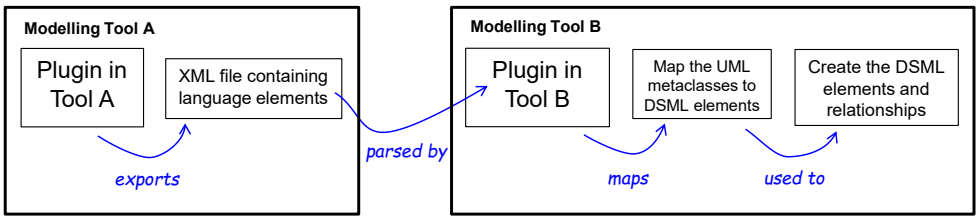


Figure 5.2: The general concept for interchanging DSML elements between two modelling tools. Figure adapted from [GBJ+24].

tools. However, modelling tools must be capable of providing the functionalities that allow the different parts of the concept to work. In the figure, the concept details both the export and import processes from one modelling tool to another. First, a language engineer working in the modelling environment of modelling tool A creates a plugin that is able to extract the specification and properties of language elements of a particular language into an XML file. Next, a language engineer working in the modelling environment of modelling tool B creates a plugin that is able to parse the XML file that was created in modelling tool A. The XML file must be defined in a generic way so that both plugins in modelling tools A and B are able to use the same import and export mechanisms, i.e., through a GPL code to parse the language elements. This fosters the general reuse of similar logic in the import and export of XML files. Then, the DSML elements are identified and mapped into their respective UML metaclasses. Finally, this plugin in B creates the mapped DSML elements and their relationships. This process allows a complete language in modelling tool A to be mapped and defined in another modelling tool B. In detail, the following steps are required for exchanging a DSML across modelling tools.

Tool Plugins. Modelling tools come with a predefined set of functionalities. They are also extensible such that functionalities be added or modified using customisations that are possible using plugins or add-ins. In the described concept, plugins for both the modelling tools A and B are developed by language engineers. The extension of the modelling tools allow customisations such as adding a new GUI or writing custom code in any GPL for enhancing its default functionalities. A's plugin extracts language elements and their properties, such as symbols, relations, and even layout information into an XML formatted file. The plugin is written in GPL code, such as Java, that performs the extraction and subsequent export of the language elements through various APIs exposed by the modelling tools. B's plugin must be able to perform an import of the XML file and further provide capabilities to parse, map, and create the DSML elements.

Common File Format. The XML file extracted in the previous step serves as the common file structure that is parsed by B's plugin and consists of both UML as well as DSML constructs. This common file structure that is defined by the language en-

gineers is beneficial in ensuring consistency between different versions of a DSML and for ensuring the same structure is used for importing and exporting language elements in both modelling tools *A* and *B*. In essence, this XML file consists of a set of nodes and attributes that convey information about the various elements of a DSML and is considered a kind of a meta-metamodel [SRVK10]. The file contains only the necessary information needed to build a DSML profile in *B*, and must disregard any tool-specific information. Thus, both *A* and *B*'s plugin need not be modified frequently once it is configured to parse the XML file.

Mapping of UML Metaclasses. In the following step, *B*'s plugin goes through the language elements exported from *A* and maps the individual elements to its appropriate UML metaclass. In profile-based tools such as EA or MagicDraw, the underlying metaclass of every DSML element is a UML metaclass. While this assumption holds primarily for UML-profile based tools, the mapping of DSML elements to its respective metaclasses must be performed to avoid the loss of metadata information for the DSML elements from *A*. An example is to map a product in a product line to a UML class metaclass, such that the metaclass of a product stereotype is configurable as an object.

Create the DSML Profile. In the final step, the mapped data needs to be translated into elements of a DSML. This is either achieved by designing a UML profile consisting of DSML elements or is configured using files in a file system directory. *B*'s plugin utilises *B*'s API capabilities to automatically create elements and their properties directly on a profile in *B* using GPL code. *B* must also provide sufficient APIs that allows the creation of language elements automatically. The DSML, therefore, consists of the language profiles, elements, and their relationships that were exported from *A* to *B*.

5.3 Implementation of the DSML Exchange Mechanism

To describe the implementation of the concept of DSML exchange in the previous section, a series of steps is performed between MagicDraw and EA. In particular, individual plugins are created in both the modelling tools that allow metamodel information to be transformed from one format to another. This information is exchanged using standard file formats such as XML. In the following, we examine how constructs of a DSML are imported and exported between the two modelling tools. The implementation serves as a basis for an exchange mechanism that is reused across other modelling tools.

5.3.1 Add-in for Enterprise Architect

To successfully import and export constructs of a DSML we reuse and extend the functionalities of the add-in for EA, called Reference Architecture Model Industry 4.0 (RAMI 4.0) Toolbox [BNL21]. The RAMI 4.0 Toolbox supports engineering complex production systems for a number of industrial domains. The RAMI 4.0 framework [ZMVS16] describes various domain-specific concepts relevant to a particular domain,

the Industry 4.0, and therefore the RAMI 4.0 Toolbox is capable of modelling both domain-specific and UML constructs. This toolbox is subsequently reused for importing as well as exporting domain-specific constructs such that the information that is generated is also reusable across different modelling tools that support standard file format exchange mechanisms such as XML imports and exports.

The export of domain-specific information from EA is done using the XML file format. The XML file is generated from the MDG technology file by implementing a custom GPL code. This XML file stores all the relevant data within a single file that is reused in other instances of EA and across other modelling tools that allows the import of XML data. On the other hand, the import of domain-specific constructs that are not exported from EA, back into EA, using the RAMI 4.0 Toolbox is not a straightforward import process. Here, the data needs to be extracted from a similar structured XML file that is generated from MagicDraw and that contains the language definition. As XML is a commonly used and standard file format, defining a different file format that is only restricted to EA and MagicDraw would hinder generalising the exchange mechanism, as designing individual file formats for each modelling tool is cumbersome. The elements of the language, including any additional data such as the relationships between the elements, have to be seamlessly integrated within an MDG file that is translated by EA.

Listing 5.1 shows the common structure of an XML file that is exported from MagicDraw. For simplicity, only the necessary parts of the XML is shown here. For the import process, the information from this XML file needs to be extracted to find all the domain-specific elements. This information is contained within the *packagedElement* XML nodes. The RAMI 4.0 Toolbox add-in then iterates through the XML file and stores all the elements extracted in-memory for processing the domain-specific information. Additionally, the types of information that is extracted from this *packagedElement* is described in Table 5.1. The first attribute, *xmi:type* classifies the element according to its respective classifier. Only those *packagedElement* nodes that are classified with the *uml:Stereotype* are considered for the actual DSML exchange, as other *packagedElement* nodes do not contain domain-specific information. Within this stereotype, each name is identified using the *name* attribute. The metaclass belonging to this stereotype is the main part of the extracted element as it must be consistent for identification in other modelling tools. While the *xmi:id* attribute is used to uniquely identify the element for the import and export mechanism, the *name* attribute is used to reconstruct the DSML element in the resulting DSML, which corresponds to the symbol names in the textual space Section 4.2. This allows the attribute *referentPath* to be used commonly across both EA and MagicDraw, as they are UML standards.

Apart from the attributes described, the relationships between elements must also be considered in the correct order for export from MagicDraw's XML file. The XML node *DSL_Customization:Customization* is processed, and consists of three attributes, the source element, the target element, and the type of relation between the elements. For the relationship type, the *customizationTarget* attribute is processed, as it is used

to refer to the actual *packagedElement*s. To extract the source and target elements, the attributes *typesForSource* and *typesForTarget* are parsed. This information is extracted by the add-in to identify and connect the DSML elements within the language profile that is being currently generated.

```
<?xml version='1.0' encoding='UTF-8'?>
<xmi:XMI xmlns:uml='http://www.omg.org/spec/UML/20131001'
    xmlns:xmi='http://www.omg.org/spec/XMI/20131001'>
  <uml:Model xmi:type='uml:Model' xmi:id='
      eee_1045467100313_135436_1' name='Model'>
    <packagedElement xmi:type='uml:Profile' xmi:id='
        _b54029e_1668094323094_836062_1271' name='
        UseCaseActorTaskLanguage'>
      <packagedElement xmi:type='uml:Stereotype' xmi:id='
          _b54029e_1668094364465_665740_1302' name='UseCase'>
        <ownedAttribute xmi:type='uml:Property' xmi:id='
            _b54029e_1668094577791_675126_1383' name='base_UseCase'
            visibility='private' association='
            _b54029e_1668094577791_302941_1382'>
          <type>
            <xmi:Extension extender='MagicDraw UML 2021x'>
              <referenceExtension referentPath='UML Standard
                  Profile::UML2 Metamodel::UseCase' referentType='
                  Class'/>
            </xmi:Extension>
          </type>
        </ownedAttribute>
      </packagedElement>
      ...
    </packagedElement>
  </uml:Model>
  <DSL_Customization:Customization xmi:id='
      _b54029e_1668094615937_364570_1424_application' base_Class='
      _b54029e_1668094615937_364570_1424' customizationTarget='
      _b54029e_1668094598037_119985_1393' hideMetatype='true'
      typesForSource='_b54029e_1668094385264_750751_1327'
      typesForTarget='_b54029e_1668094497073_782354_1353'/>
  ...
</xmi:XMI>
```

Listing 5.1: An excerpt of an XML file detailing elements of a DSML generated in MagicDraw consisting of use cases, actors, tasks, and their relations.

To generate and use the DSML in EA, the information that has been extracted so far needs to be stored within the EA-MDG file. The parsed DSML elements, their

Attribute	Example	Goal
xmi:type	uml:Stereotype	Classifier
name	Task	Element Designation
referentPath	... ::UML2 Metamodel:Task	UML Class

Table 5.1: The attributes of an XML file and their descriptions for interchanging DSML elements between MagicDraw and EA.

metaclasses, and their relationships are stored internally into a UML Profile of the MDG file. Any additional domain-specific information is hence captured by enhancing the add-in to support further DSML constructs. After storing all the UML and DSML elements in the UML Profile, the relevant elements are further logically grouped in different toolboxes using the Toolbox Profile. A difference to note here is the distinction between the relationships and various elements, as they are logically separated. Next, the Diagram Profile is adapted so that the toolbox is now well integrated within the tool, and is ready to be used on tool start-up. Currently, the exchange of images and icons between MagicDraw and EA is not supported by default, but if required they must be transmitted in addition to the XML file for subsequent integration. The RAMI 4.0 Toolbox offers the functionality to import images within the MDG that is individually selected for each DSML element. To ensure consistency, if no image is selected, the UML representation is used without any modifications.

5.3.2 Plugin for MagicDraw

MagicDraw uses an XMI standard [XMI23] that is defined by the OMG for export into an XML file. It is a commonly defined file format that stores UML and DSML data in a way that allows language information exchange between other instances of MagicDraw and other modelling tools. However, a limitation of XMI is that it provides a rather limited amount of additional information that a DSML captures. Therefore, the way in which modelling constructs are managed in different modelling tools makes DSML information interchange using this common XMI format challenging. Any domain-specific constructs that are exported from EA and contains XML nodes and attributes along with EA-specific data becomes a challenge for MagicDraw's default import mechanisms, largely due to the inability of MagicDraw to successfully parse such information. Consequently, there is a need to resolve these conflicts that enable a seamless exchange of DSML and its constructs.

As discussed previously in Section 3.4, customisations to a DSML is done using additional MagicDraw plugins that is eventually bundled together in a single *.mdzip* file. Hence, to achieve the import of a language profile from EA, we in this thesis developed a

plugin in MagicDraw using the Open Java API that acts as the interface to import XML information generated from EA. This plugin adds an additional context menu item to MagicDraw's standard import mechanism. Hence, a plugin in MagicDraw is considered equivalent to EA's add-in. This context menu item allows users to select the exported XML file from EA. The plugin then parses the information from this XML file and extracts the necessary DSML constructs. The same attributes defined in Table 5.1 is reused in MagicDraw as well to provide a consistency between the modelling tools. UML classifiers such as an *Actor*, *Activity*, or a relationship *Association* is directly translated to their respective stereotypes in MagicDraw, while any domain-specific types needs additional handling, such as mapping a *Task* to a UML class. After this information is extracted and translated in-memory, the plugin creates a language profile to add the necessary stereotypes and their customisations. Once the information is bundled into an archive and installed within MagicDraw, on tool start-up users easily create their models using these DSML constructs that were previously defined in EA.

Listing 5.2 shows the structure of an XML file that is exported from EA. The structure when compared to Listing 5.1 indicates the common *packagedElement* nodes in the XML files used for storing the DSML elements. The MagicDraw plugin uses the *java.xml* Java package to parse the XML nodes, tags, and attributes. The elements of the DSML is configured under the *uml:Model* and *packagedElement* nodes, similar to the information that MagicDraw exports. To parse this information, the *DocumentBuilderFactory* class within the *java.xml* package is utilised, and a set of *Node* and *NodeList* objects is used to find the relevant elements and store them in-memory into custom objects defined in the plugin. Then, using MagicDraw's *ProjectManager* API, the DSML project, language profile, and a UML profile diagram is created. This is required to create the domain-specific elements, its relationships, and its customisations. Finally, the classifiers, UML or domain-specific, for each created stereotype is automatically configured.

To avoid XML conflicts such as with domain-specific classifiers, the identification of the most relevant metaclass for a specific *xmi:type* attribute must be done. As an example, a *uml:Task* type is assigned the *Class* metaclass, because in a domain-specific scenario, tasks are considered as activities in a UML activity diagram (UML AD) to be the most general UML classifier. For a relationship connector, such as a *uml:Association* type, the plugin automatically assigns the same classifier as the one defined within EA. However, such associations must be configured automatically so that the association only exists between two defined stereotypes. To achieve this, the *ownedEnd* nodes of a *packagedElement* in the EA XML file is extracted and the source and destination ends are looked up in the respective identifiers. These identifiers are used to configure the *typesForSource* and *typesForTarget* customisation properties in MagicDraw that has been discussed in previous parts of this chapter to ensure that the connectors are only used between particular model elements during design time. A difference between the two modelling tools is that in MagicDraw, these relations are configured outside of the parent nodes, while in EA, they are configured within the respective nodes that include

the DSML. As with EA, the import process in MagicDraw also specifies that images and icons cannot be directly stored in the XML file, but needs to be transmitted independent of the overall exchange mechanism.

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmlns:uml="http://www.omg.org/spec/UML/20110701"
    xmlns:xmi="http://www.omg.org/spec/XMI/20110701">
  <xmi:Documentation exporter="Enterprise Architect"
      exporterVersion="6.5"/>
  <uml:Model xmi:type="uml:Model" name="EA_Model">
    <packagedElement xmi:type="uml:Package" xmi:id="
        EAPK_FF1CC5DB_4eb9_AFF6_12A8F389A6BF" name="Business Layer"
        >
      <packagedElement xmi:type="uml:Actor" xmi:id="
          EAID_5831D1A8_4068_A436_7A67C0F0F4ED" name="Actor"/>
      <packagedElement xmi:type="uml:InstanceSpecification" xmi:id
          ="EAID_3B0EAB4A_4262_B612_3368AFB781B2" name="UseCase"/>
      <packagedElement xmi:type="uml:Association" xmi:id="
          EAID_35FB499D_40c2_B3F1_01991CB5C999" name="Performs">
        <memberEnd xmi:idref="
            EAID_dstFB499D_40c2_B3F1_01991CB5C999"/>
        <ownedEnd xmi:type="uml:Property" xmi:id="
            EAID_dstFB499D_40c2_B3F1_01991CB5C999" association="
            EAID_35FB499D_40c2_B3F1_01991CB5C999">
          <type xmi:idref="EAID_17344233_4ca7_BB8A_AE65EC647D22"/>
        </ownedEnd>
        <memberEnd xmi:idref="
            EAID_srcFB499D_40c2_B3F1_01991CB5C999"/>
        <ownedEnd xmi:type="uml:Property" xmi:id="
            EAID_srcFB499D_40c2_B3F1_01991CB5C999" association="
            EAID_35FB499D_40c2_B3F1_01991CB5C999">
          <type xmi:idref="EAID_5831D1A8_4068_A436_7A67C0F0F4ED"/>
        </ownedEnd>
      </packagedElement>
      <packagedElement xmi:type="uml:Class" xmi:id="
          EAID_17344233_4ca7_BB8A_AE65EC647D22" name="Task"/>
    </packagedElement>
</xmi:XMI>
```

Listing 5.2: An excerpt of an XML file generated in EA detailing a DSML consisting of use cases, actors, tasks, and their relations.

In the final step of the import process, the language elements as stereotypes and their customisations are created within a DSML profile. As soon as all the artefacts, such as the language elements, their relations, and their customisations are created,

they are released from in-memory to an archive *.mdzip* file. This also includes any additional plugins that are referred internally by the main plugin. Further, icons, that are stored externally as scalable vector graphic images, and model templates are added to the language elements, such that the modellers utilise them during design time. The *.mdzip* file consists of the domain-specific constructs exported from EA, and is installed directly on the tool. The exchange mechanism therefore supports language engineers in configuring families of similar DSML profiles across both EA and MagicDraw. As stated in the EA description, MagicDraw provides a default export functionality for exporting a DSML profile that is used as-is for import in EA.

5.4 Application of the DSML Exchange Mechanism

The application of the bidirectional exchange mechanism between EA and MagicDraw is described in this section.

5.4.1 DSML for Use Cases, Tasks, and Actors

The application of this mechanism is studied using an example involving actors, tasks, and use cases. In this example, individual use cases, actors, tasks, and their relationships are created as part of a *UCACTaskDSML* in one modelling tool and is then exchanged in the other modelling tool. To illustrate the applicability, Figure 5.3 shows all the language elements including their properties that have been designed by a language engineer in EA for a modelling project. This consists of the DSML elements inherited from their respective UML metaclasses and the relation *Performs* between an actor and a task. This metamodel consists of all the language components necessary to define the *UCACTaskDSML* and used to model use cases, actors, tasks, and their relations. We now use the RAMI 4.0 Toolbox add-in to generate the corresponding MDG file that stores all the information related to this DSML. While use cases and actors are represented by UML classes, the tasks need to be specially handled, i.e. assigned the UML metaclass *Class*, since it is a domain-specific element for this example. Therefore, the XML file generated from the EA add-in consists of the entire language definition that includes use cases, actors, tasks, and the relations between them as is defined in the DSML configured in EA.

Subsequently, in another modelling project the need arises to design a similar language involving use cases, actors, tasks, and their relations, but in the MagicDraw ecosystem. Language engineers now easily use the MagicDraw plugin designed for importing the exported XML file from EA. The MagicDraw plugin is able to parse the XML file, extracting the domain-specific constructs of the DSML from the file and create the required stereotypes and their customisations in MagicDraw automatically in a MagicDraw UML profile diagram. Figure 5.4 shows the constituents of the resulting profile diagram from this import mechanism. It consists of the various stereotypes for the use case, task, actor,

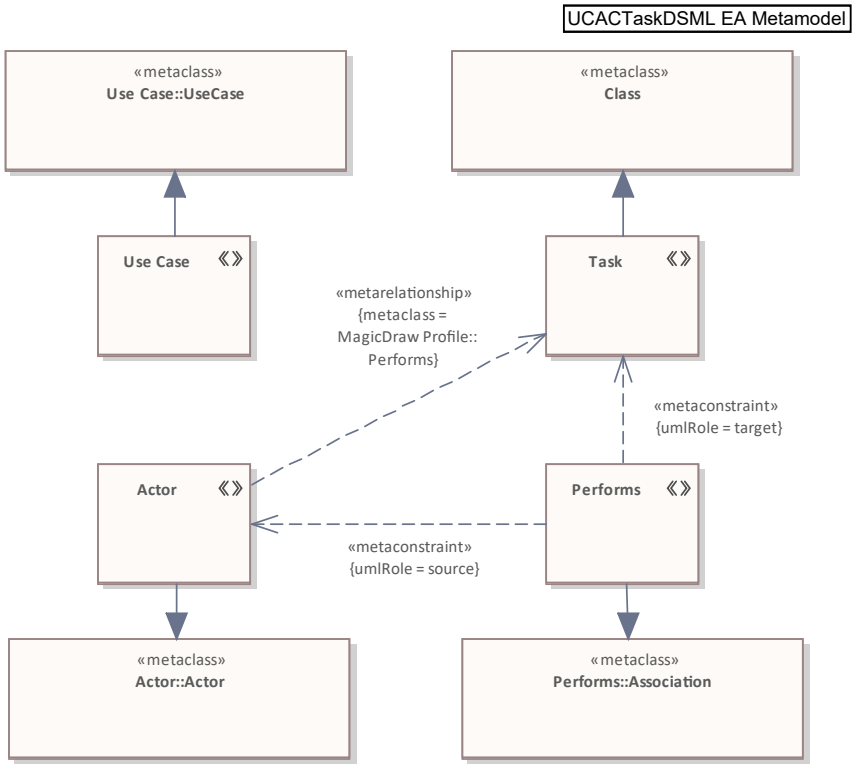


Figure 5.3: A UML Profile Diagram in EA consisting of the use case, task, actor, and association DSML elements and their properties.

association, along with their configured properties. This allows the DSML constructs to be reused completely in a different modelling tool, that belongs to a completely different ecosystem. However, using a standard XML file, some properties of the DSML elements, such as icons or images cannot be directly interchanged from EA to MagicDraw, but are transmitted independent of the XML file. In a similar fashion, if the DSML consisting of the use cases, actors, tasks, and the relations are created first in MagicDraw, then the XML file is directly exported for import purposes into EA. On successfully importing the DSML using the EA add-in, the DSML elements are directly stored in a MDG file and all the elements of the DSML are logically grouped together and installed automatically within EA for direct use on tool start-up.

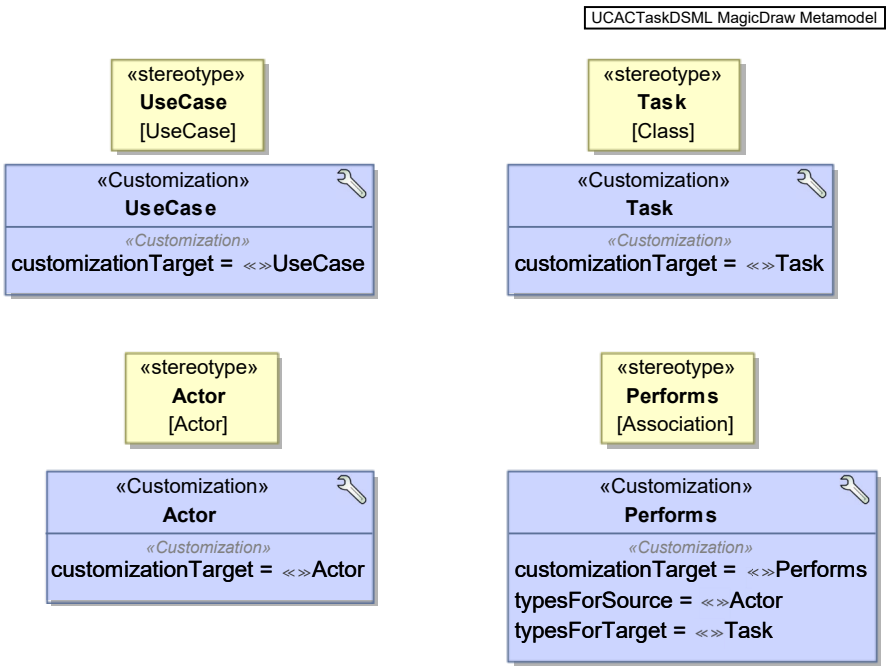


Figure 5.4: A UML Profile Diagram in MagicDraw consisting of the use case, task, actor, and association DSML elements and their properties extracted from EA. Figure taken from [GBJ+24].

5.4.2 Example Model using the Use Case, Task, and Actor DSML

Based on the example DSML described in the previous section, Figure 5.5 is a model designed using the DSML constructs created in MagicDraw from the import of the EA XML file. In this model, the elements of the model, namely *Customer*, *UC1*, *Performs*, and *T1* are designed by a user using the DSML. By default, stereotypes are automatically configured for each of these model elements. Additionally, the user decides to modify the already configured stereotypes manually. This gives users flexibility in using a DSML element with a different classifier, that reflects variability in a product line engineering.

In this example, the *T1* model element is automatically assigned the stereotype *Task*, which is of a UML *Class* metaclass. In a similar way, the *Customer* model element is automatically assigned the stereotype *Actor*, which is of a UML *Actor* metaclass. The *UC1* model element is automatically assigned the *UseCase* stereotype, which is of a UML *UseCase* metaclass. Finally, the *Performs* model element is automatically assigned the *Association* stereotype, which is a UML relationship. The relation between the *Customer* and *T1* only allows a *Performs* association, as this association has been assigned the source and target stereotypes that were exported from EA as shown in

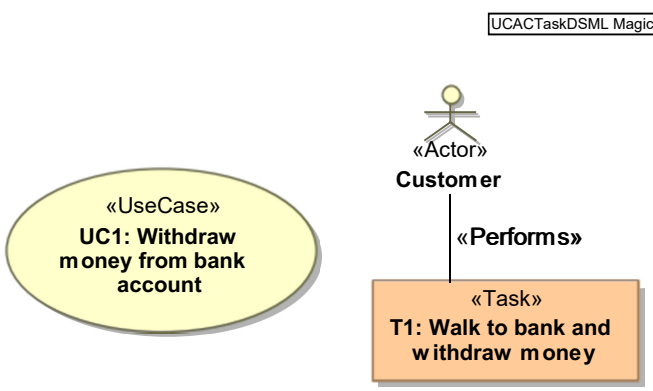


Figure 5.5: An exemplary MagicDraw model using the DSML elements configured in Figure 5.4 showing an actor performing a task for a respective use case. Figure taken from [GBJ+24].

Listing 5.2. This example model demonstrates how the DSML and its elements that are created in EA, exported to an XML file, and finally imported by the MagicDraw plugin is reused as-is in MagicDraw to create the respective models. Similarly, the XML file containing the DSML definition generated from MagicDraw is imported into EA using the add-in specified earlier. This is shown in Figure 5.6 where the model elements *UC1*, *T1*, *Customer*, and the relation *Performs* between the *Customer* and *T1* is established. This model is created using the metamodel defined earlier for the EA UML profile in Figure 5.3. Thus, language engineers reuse domain-specific parts of a DSML in initially incompatible modelling environments and achieve the interchange of DSML constructs across different modelling tools.

5.4.3 Findings

Defining a common interchange method allows the exchange of DSML constructs between EA and MagicDraw. With the described example, the use of both UML and domain-specific constructs are illustrated for both the modelling tools. To achieve the bi-directional exchange of domain-specific constructs between two independent modelling tools, additional steps are required. These steps include defining a common file format for representing the constructs of the DSMLs in the respective modelling tools, the development of additional plugins or add-ins that allows the parsing of such a common file format, the mappings that need to be made in order for the underlying metaclasses in the individual modelling tools to match correctly, and finally designing models that conform to the DSMLs built from the exchange mechanism. The mapping of individual DSML constructs shows the synergies between the two modelling environments for representing a common language construct. Therefore, such a mechanism reduces the overall efforts

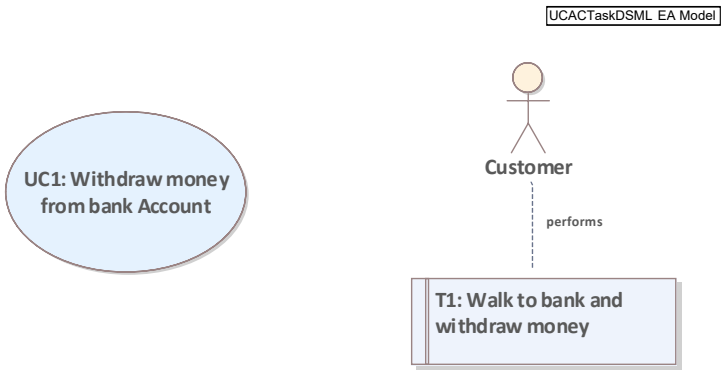


Figure 5.6: An exemplary EA model using the DSML elements configured in Figure 5.3 showing an actor performing a task for a respective use case similar to the example described for MagicDraw in Figure 5.5.

that language engineers undertake for designing similar languages in two incompatible modelling environments.

5.5 Reproducibility Considerations

The implementation described so far in this chapter is reproducible for arbitrary DSMLs in the respective modelling tools. This section, therefore, aims at providing language engineers with the necessary technical details to reproduce the exchange mechanism in the context of EA and MagicDraw. Algorithm 1 details a pseudo-code that enables the extraction of DSML elements from the EA XML file (Listing 5.2) and store the contents of the file in-memory for further processing in MagicDraw to create the corresponding DSML definition. In summary, the mechanism to import an EA file to MagicDraw requires the availability and selection of the EA file in read-only mode, parsing of the XML file using a XML parsing library in Java, extraction of the various nodes and attributes of the XML file, and the assignment of the appropriate UML metaclass in preparation for creating the DSML definition.

Algorithm 2 details a pseudo-code that is used to create a DSML profile in MagicDraw based on the contents that were parsed, extracted, and stored in-memory using Algorithm 1. Here, MagicDraw's Open Java API is used to create project and profile instances and sessions of Java packages that allows maintaining projects, stereotypes, and their properties. In particular, the *StereotypesHelper* package using the MagicDraw Open Java API is used to create the stereotypes and assign certain properties to them. This allows the creation of all the UML as well as the DSML constructs from EA, and allows setting up the respective relationships between the DSML elements.

Algorithm 1 Import an EA XML file to MagicDraw

```
1: EAfile ← select the XML file from a file import UI
2: Parse EAfile using DocumentBuilderFactory (java.xml.*)
3: rootElement ← uml:Model element of the EAfile
4: xmiType ← xmi:type attribute of rootElement
5: name ← name of rootElement
6: while packagedElement of rootElement exists do
7:     get xmi:type and name of packagedElement
8:     if xmi:type = Association then
9:         get ownedEnds of the packagedElement
10:        get source and destination IDs
11:    if xmi:type ≠ UML standard then
12:        Assign correct UML metaclass
```

Algorithm 2 Create a Language Profile in MagicDraw

```
1: Start ProjectsManager instance in MagicDraw
2: Create a project P
3: while packagedElements from Algorithm 1 exists do
4:     if packagedElement ≠ Association then
5:         Create stereotypes with StereotypesHelper
6:         Assign xmi:type and name to stereotypes
7:         Set the customizations for this stereotype
8:         Add to project P
9: Repeat step 3 for packagedElement = Association
       and assign typesForSource (source) and typesForTarget
       (destination) properties
```

In a similar fashion, Algorithm 3 outlines the pseudo-code for converting a MagicDraw profile and embedding it into EA for the DSML definition. Here, the application was done using C# as the GPL for the plugin to show that the implementation is not limited to a single GPL. In this case, the information is parsed and extracted from the MagicDraw XML file (Listing 5.1) and is directly integrated into to profile folder of EA, which is shown in Figure 5.1, as part of the DSML definition. This algorithm describes that the transformation of a DSML profile is performed using an independent EA plugin, the GUI of which is integrated with the EA modelling tool. Upon launching the GUI, the exported XML from MagicDraw is selected and parsed for conversion. The conversion part of the algorithm searches for all *packagedElements*, which essentially contain the information about the DSML elements. The *DSMLCustomizations* store information pertaining to the restrictions of the language and are considered during the DSML profile

transformation at a later stage. The information extracted from *packagedElements* and *DSMLCustomizations* are stored into two individual lists. The first list contains all the DSML elements created within the MDG file and are provided to EA. The second list contains all the restrictions to the elements created using the first list. These restrictions include allowing only a specific relationship between two DSML elements or contains information about the styling of the element such as its shape or colour.

Algorithm 3 Import MagicDraw XML file to EA

```
1: start the EA GUI
2: select MagicDraw XML file
3: while MagicDraw XML has XML nodes do
4:     get packagedElement or DSMLCustomization
5:     store elements in separate in lists
6: for each element in packagedElements do
7:     create MDG profile element in EA XML
8: for each element in DSMLCustomizations do
9:     if customization is constraint then
10:        find MDG profile element
11:        add metarelationship
12:        add metaconstraint
13: store the XML file into the EA MDG folder
```

5.6 Discussion

This chapter discusses a bidirectional exchange mechanism for a DSML and its constructs specifically between the commercial modelling tools of EA and MagicDraw. Every modelling tool has its own challenges in modelling complex systems and require extensive functionalities for language development. Therefore, considerations for interlinking additional functionalities from other modelling tools such as import or export of modelling information is often neglected. The example used in this chapter illustrates the use of both UML and domain-specific constructs that is generally valid in both the modelling tools within the graphical modelling technological space. However, the exchange mechanism described in this chapter are also generalisable to other modelling tools to ensure a foolproof and seamless interoperability of DSMLs. Efforts to solve this challenge have been made by designing commonly interchangeable standards such as with XMI formats. Despite such efforts, achieving true interoperability of DSML elements between modelling tools requires additional efforts as language definitions across modelling tools are often not equivalent. The syntax, both abstract and concrete, are also not entirely equivalent and thus it is not possible to reuse well-formedness rules, transformations, and

symbol tables to derive a bidirectional exchange of DSML constructs. Therefore, there are always additional steps required in order to successfully exchange domain-specific constructs. This means that tools must support the combined export of both domain-specific and UML information in a standard form to better allow building versions or families of such DSMLs. The concepts described in this chapter allow language engineers to develop plugin mechanisms ad-hoc that are combined with a modelling tool. Such plugins reuses XML formats to exchange UML constructs and additionally includes a detailed mapping and translation of domain-specific constructs across two modelling tools. Thus, language engineers who primarily work with UML-profile based modelling tools and customise UML metamodels to develop DSMLs are benefited from the concepts presented in this chapter.

While this chapter presents a bidirectional exchange mechanism, there are certain limitations to the mentioned approach. The approach has been validated in two commercially used modelling tools in the graphical space, that makes the study vendor-locked. The tools used in this chapter provide APIs that allow a DSML and its elements and relations to be exported through suitable data formats. Thus, modelling tools must be able to provide for a mechanism to export data so that ill-defined syntax and semantics may not be a limiting factor in the import and export process. Further, a complete interoperability may often not be necessary for teams managing smaller projects, therefore a reduced import and export process that handles only specific language constructs is often sufficient. For textual language integration [DJRS22] into graphical modelling tools, there must be support for allowing the set of syntactic sentences as well as the validation of well-formedness in the respective contexts. Another limitation of the described approach is that different stereotypes may not be mapped to a specific concept, that reduces the consistency of defining language constructs across different DSML building blocks. In this chapter, we discussed the exchange mechanism for a single DSML project, therefore future work must take a look at establishing common definitions across multiple projects as well as DSML building blocks. The translation of methods and UXD aspects across modelling tools have not been explicitly explored in this chapter as the exchange mechanism focussed on specific parts of the language, its syntactic definition. Finally, this exchange mechanism does not consider privacy and security of data across modelling tools, as the mechanism has been defined at a metamodel level.

As systems become more complex, challenges for language engineers in exchanging domain-specific constructs between modelling framework, ecosystems, and tools that support language development also increase. While standard file interchange formats, such as XMI, exist and works well for exchanging UML constructs such as with UML models with other tools, it often does not cover all aspects of a DSML. Different modelling tools generate different formats of XML files for domain-specific constructs that eventually prevents a single effective solution of language exchange between multiple modelling environments. Therefore, the exchange mechanism described achieves the bidirectional exchange of DSML elements between EA and MagicDraw. Here, both

UML as well as DSML constructs are interchanged within instances of the individual modelling tools along with other modelling tools. Two custom Java-based application interfaces, as EA add-in and MagicDraw plugin, are created that enable the import and export of domain-specific information across the tools. This ensures that domain-specific information is not lost during transfer of a DSML across other modelling projects and various stakeholders. In addition, cross-functional teams across different organisations reuse the same language definitions for use in different domains, thus eliminating the need to build every part of the DSML from scratch. Although in this chapter we elaborate the DSML exchange mechanism process, further work is still needed to validate the implementation across other modelling tools. One threat to validity for this study is being in a vendor-locked scenario, since the implementation is described using two modelling tools. However, any modelling tool that supports language development and the ability to execute external GPL code such as in Java, use this mechanism. Ultimately, a seamless DSML exchange mechanism between two commercially established modelling tools is enabled by detailing the process needed to exchange DSML constructs between various modelling environments that enable graphical modelling in particular.

5.7 Related Work

Solutions to designing complex systems with DSMLs often come with problems across business lines and organisations that require a constant upkeep of such DSMLs. It is therefore crucial to build modular and reusable DSML elements that are seamlessly interchanged between modelling tools without the need to make significant adjustments to the language itself. Most other related works, focus on the transcompilation or metamodel translation between language workbenches through symbol names [DJK+19], whereas the concepts described in this chapter facilitates bridging the reuse of DSML constructs across graphical modelling tools through unique element identifiers such as the *xmi:id* described in Listing 5.1. However, the *name* attribute is used to reconstruct the corresponding DSML element in the resulting DSML. Standard file formats such as with XMI or XML [XML23, Mod23a] have emerged as solutions to address this challenge. On the other hand, with every MBSE approach [BCL+21, CCF+15], it becomes a greater challenge to engineer [CBCR15] and exchange [HRW18] domain-specific constructs across a variety of domains. This challenge is also extended to reusing standard file formats that is eventually used to generate DSML elements that users model with using different modelling tools or language workbenches [HR17, DJRS22, EVDSV+15, Mer10].

Even with techniques that provide a common format to exchange models between model-driven software and systems development tools, the challenges with model interchange still exists [Rum16]. This is partly because of different modelling tools storing the DSML concepts such as UML or the SysML using different file formats and data structures [DJM+19]. OMG, in this regard, has come up with a standard way to exchanging

models between different tools using XMI file formats [XMI23]. However, the export of domain-specific elements for similar models [ZNG15] requires a clear specification in terms of the semantics of these exchange formats [GRR09, MRR11]. For example, if the UML stereotypes are modelled with the tool MagicDraw and later exported as XML, the elements that are exported is configured with the metaclass *uml:Stereotype*. In contrast, the same model generates a UML metaclass *uml:Class* when it is exported from EA.

Previously in Chapter 4, we took a detailed look at how languages are composed in language workbenches and graphical modelling tools and discussed the concepts for extending a language through novel syntax, embedding an embedded language into a host language, and aggregating a language by referencing to model elements using symbol tables [BMR22]. The software components, its artefacts, and the models undergo transformations [TAB+21, BEH+20] that allow a language to be inherited, extended, embedded, or aggregated [EGR12, MCGDL12]. In contrast, the exchange mechanism described in this chapter provides a lightweight customisation capability without the need for providing direct interfaces between modelling tools and modifying existing language infrastructure. Globalising DSMLs is achieved with the use of compositional mechanisms [CBCR15], but often require additional customisations [DJRS22] in their respective technological spaces [EvdSV+13]. Templates to map different metamodels during model transformations [SCGL11] or multi-level modelling paradigms [LGC14] have also been studied for reusing models across language workbenches. However, the concepts presented in this chapter primarily studies the applicability of transforming DSMLs across UML-profile based graphical modelling tools.

Previous work bridges the gap in different technological spaces that focus on translating languages [BJRW18, DJK+19], providing interoperability of models and language elements [DCL13, BBC+10], and for improving modelling language variability [GR11]. A study on model-level integration of the OCL standard library [BD07] through pivot models note the limitation of a common format for the library when OCL is integrated with UML profiles for defining DSLs. In contrast, this chapter discusses DSMLs extended via UML-profile based modelling tools. A model-bus architecture [BGS05] allows the services of different modelling tools to be connected using a functional description metamodel by establishing concrete connections. However, implementing model transformations such as with model-bus in industrial applications requires extensive efforts for language engineers that work under rather stricter project resources [KBC+19]. Language interoperability and model synchronisations between modelling tools is discussed in [BCC+10] but the application of such concepts in an industrial context is still missing. Another example that provides model exchange between stakeholders in a single project has been evaluated with LemonTree[1], but their approach considers only UML constructs and do not consider domain-specific constructs. In another work, the interoperability in complex adaptive enterprise systems is only partially researched [WGN16] in regards

[1] https://www.lieberlieber.com/lemontree/

to domain-specific contructs. Further, solutions for extending DSLs, that are embedded in language development tools using tagging languages [GLRR15] or tool integration frameworks for multi-paradigm modelling [MDLV12] are present in the literature. Studies [Ozk19, Ker14] also show that the two commercially modelling tools detailed in this chapter, EA and MagicDraw, provide ample customisations for exchanging both UML and DSML constructs.

Chapter 6

Design Considerations for High-Quality User Experience in Industrial DSMLs

In this chapter, design guidelines for improving the overall modelling experience for industrial DSMLs is discussed. Previous chapters looked at composing DSMLs from the viewpoint of language components and their definitions, based on the syntax and semantics of a modelling language by reusing DSML building blocks. While, describing modular concepts for reusing language components is important in DSML engineering, the focus on users and their experience with using DSMLs is equally important. The guidelines presented in this chapter are aimed for language engineers in designing state-of-the-art DSMLs for practitioners that improves their overall user experience in their modelling environments. Some results of this chapter have been published in [GJRR22a]. Therefore, passages from the paper may have been quoted verbatim in this chapter.

As domains become more heterogeneous and complex with constantly evolving domain concepts, practitioners in the industry often face challenges as part of usability considerations, specially for graphical DSMLs. Guidelines are still missing that language engineers must consider to improve the overall user experience (UX) for all stakeholders involved in a modelling project. UX is a very subjective topic as, in general, users are often influenced by a variety of factors. Because of this vastness in topics surrounding UX, proposing guidelines and definitions of UX is often complicated, excessively generic, and most times tied to a specific technological space. This particular challenge is solved by leveraging existing design principles and many standards of human-centred design. With a combination of such designs, this chapter proposes definitions and guidelines that are specifically suited for improving UX and user experience design (UXD) for industrial graphical DSMLs. These important aspects of UXD are categorised, allowing language engineers to design better and high quality graphical DSMLs. The overall aim for such DSMLs is to invoke positive emotions among practitioners during their modelling. To this end, the proposed UXD guidelines are detailed in way that they are rationalised in supporting language engineers build better usable DSMLs and are more widely accepted by practitioners.

Introducing modelling in the early stages of a project in the systems engineering domains comes with its challenges. The overall DSML engineering requires develop-

ing concepts, methods, along with tools that allows practitioners to effectively employ models rather than traditional documents. Accordingly, as GPLs cannot suffice the design of system models [PB19], there is slow and constant shift towards evolving MBSE techniques [FR07]. This is largely based on the fact that GPLs cannot really provide complete solutions for domain-experts in contributing to solutions of their domains. DSMLs are therefore being developed for supporting such domain-specific abstractions [CBCR15, GKR+21], primarily in the textual, graphical, or projectional technological spaces [DCB+15, Bet16, Tol06]. As complexities in engineering projects increase, so does the natural complexity in developing DSMLs for such domains, which leads to users consequently often struggling to use such DSMLs effectively.

Assisting practitioners in their modelling serves a beneficial purpose in conveying key aspects of a domain that leads to better decisions in the engineering of their systems. While a modelling tool is only a part of the entire lifecycle of the modelling process, a solid foundation needs to be established in combination with an appropriate tooling mechanism [GKR+21]. This integrated aspect is especially necessary in promoting modelling in small and medium enterprises whilst also considering the skill level of novice and experienced users [Reg18]. As the complexity in designing languages, namely with the syntax and the semantics of the language, increases [CBCR15, CGR09], there is also a growing need for a good UX. However, there is an assumed notion that all practitioners are modelling experts and understand everything with respect to the DSML that language engineers do not consider. This is often not true since there is often a call for guidance for novice users who want an introduction to introduce graphical modelling tools but lack the desired knowledge due to their late involvement in the projects. Some of these challenges in graphical DSMLs consist of irregular visual notations of standard domain aspects, endless searches through language and model elements in the DSML, burdening users with unneeded tools and their functionalities [WHR+13], the unavailability of predefined models, and improper documentation for various modelling constructs.

In the literature, several definitions of UX and usability heuristics have emerged [Moo09, Has08]. These are highly specific to the respective technological spaces and are very generic in terms of the concepts covered under each definition. This is especially true because UX is not only a vast topic, it is also rather subjective and therefore the taste of users depend not only on the domains in consideration but is also heavily shaped by the most recently observed trends in the ubiquitous software and systems domain (Chapter 2). While UX and business logic greatly depend on each other, growing complexities in the project means often they are not considered or are lost in translation. Therefore, no single solution exists, however industry standards, design guidelines [KKP+09, CMP18, Voe09], and UXD aspects needs to be considered by language engineers during graphical DSML development. A mindset change is certainly required to achieve this, as better UX often translates to successful modelling. Therefore, this chapter aims to provide the necessary guidance for a more holistic modelling experience

for practitioners in various domains by demonstrating the same in MagicDraw.

In the following, Section 6.1 discusses the definitions of UX and UXD for graphical DSMLs in MagicDraw that is generalisable in modelling tools. Section 6.3 discusses the human-centred design approaches by categorising them into: visual design, information architecture, interaction design, and usability heuristics for consideration during DSML development. The categorisation of the UXD aspects is discussed with a running example of a feature model in MagicDraw DSML in Section 6.2 and later evaluated in Section 6.5. Finally, we discuss the scope of the UX in modelling approaches (Section 6.4), discuss the generalisation of the guidelines to other modelling tools (Section 6.6), and related approaches in Section 6.7.

6.1 Defining User Experience in MagicDraw

This section defines UX and UXD in MagicDraw and is adapted from [GJRR22a]. While the definitions have been proposed primarily in MagicDraw, they are generalisable to other graphical DSML tools as modelling tools serve the same foundational purpose: language development. User Experience (UX) in MagicDraw is defined as follows:

Definition 8 (User Experience in MagicDraw)**.** *User experience (UX) for graphical DSMLs in MagicDraw is an instantaneous intuitive feeling (positive or negative) of a user (modeller or practitioner) while interacting with the defined constructs of the graphical DSML and the accompanying graphical modelling tool, MagicDraw.*

In other words, UX is mainly described as an intuitive feeling, often non-rational, for practitioners during modelling. The goal here is to describe the concepts of a DSML such that a *good* UX satisfies the expectations of practitioners in relatively easy, simple terms and impressions, while also minimising the number of clicks or interactions needed between the modelling language and the modelling tool. The interactions are defined as the abilities of both the systems and the users to constantly influence each other so that the users reach their final modelling goals. A good UX also reflects the standards described in the literature and and keeps the user satisfied and ensures users easily navigate through the constructs of a DSML. Therefore, any positive feelings invoked during these interactions with the DSML or the modelling tool almost certainly is considered a good UX. A good UX however must not be considered synonymous with a large of number of DSML functionalities, as often with growing DSMLs, functionalities are more important because the development of the syntax and semantics of a DSML is often prioritised in time and resource constrained projects. On the other hand, a *bad* UX tends to invoke feelings of negativity, incomprehensibility, and confusion leaving DSML users unsatisfied in using such DSMLs that naturally affects all involved stakeholders. However, in reality, interacting with the current models maybe a mixture of positive and negative feelings, therefore it is important to consider design decisions that reduce this gap.

Based on the broader definition of UXD described in Section 2.2.6, we extend the UXD definition to the MagicDraw ecosystem and adapted from [GJRR22a] as follows:

Definition 9 (User Experience Design in MagicDraw)**.** *User experience design (UXD) is any design decision taken by a language engineer during the development of a graphical DSML in MagicDraw, that ultimately fosters a good UX for a user or a practitioner.*

These design decisions that influence the usability of any DSML is realised and implemented during DSML development by language engineers by involving all stakeholders during the DSML engineering of a project. Different stakeholders means different opinions which means different interpretations of UX, however, ultimately the practitioner will use a DSML to build their models, therefore efforts must be made to ensure a satisfiable UX is provided to such practitioners. In this regard, design decisions must follow the principles of human-centred design as is listed under ISO 9241-210 [ISO10] and influence the UX towards a more positive effect. This chapter describes categories for these UXD aspects in MagicDraw and are based on certain elements of UX existing in the literature [Gar10]. While the list of UXD aspects are non-exhaustive, it must be noted that they provide the general foundation required for a good UX in graphical DSMLs that is independent of the modelling environments. The design decisions provide the benefit that they cover all aspects of a DSML for providing a good UX and are based on requirements found in actual industrial projects.

6.2 Running Example

To illustrate the different design decisions and guidelines that help improve the UX, we use the following running industrial example of a feature model [BEK+19] DSML consisting of language elements, product lines, and products that has been developed for Siemens Healthineers. The DSML has been been developed using the modelling tool MagicDraw, therefore the concepts of UXD are described primarily in the MagicDraw ecosystem. In this example, a feature model DSML building block is used that has also been used in other DSMLs (Chapter 8). Figure 6.1 shows an example of a feature model modelled by a user in MagicDraw. This feature model consists of different product lines and products, features such as mandatory, optional, and Xor, and a custom UI. It lists the design decisions that are defined by the language engineers in the various parts of the visible model. In the figure, the various building blocks (discussed later in Chapter 8) that are relevant for use by the modeller are shown on the left-hand side under project template. In doing so, the relevant models and the model elements remain in their individual building blocks and eventually help user segregate the different models. In the next section (Section 6.3), we discuss the categorisation of these UXD aspects that have been annotated in the figure. In particular, different parts of the figure are annotated with design decisions and an accompanying identifier that are referenced throughout this chapter.

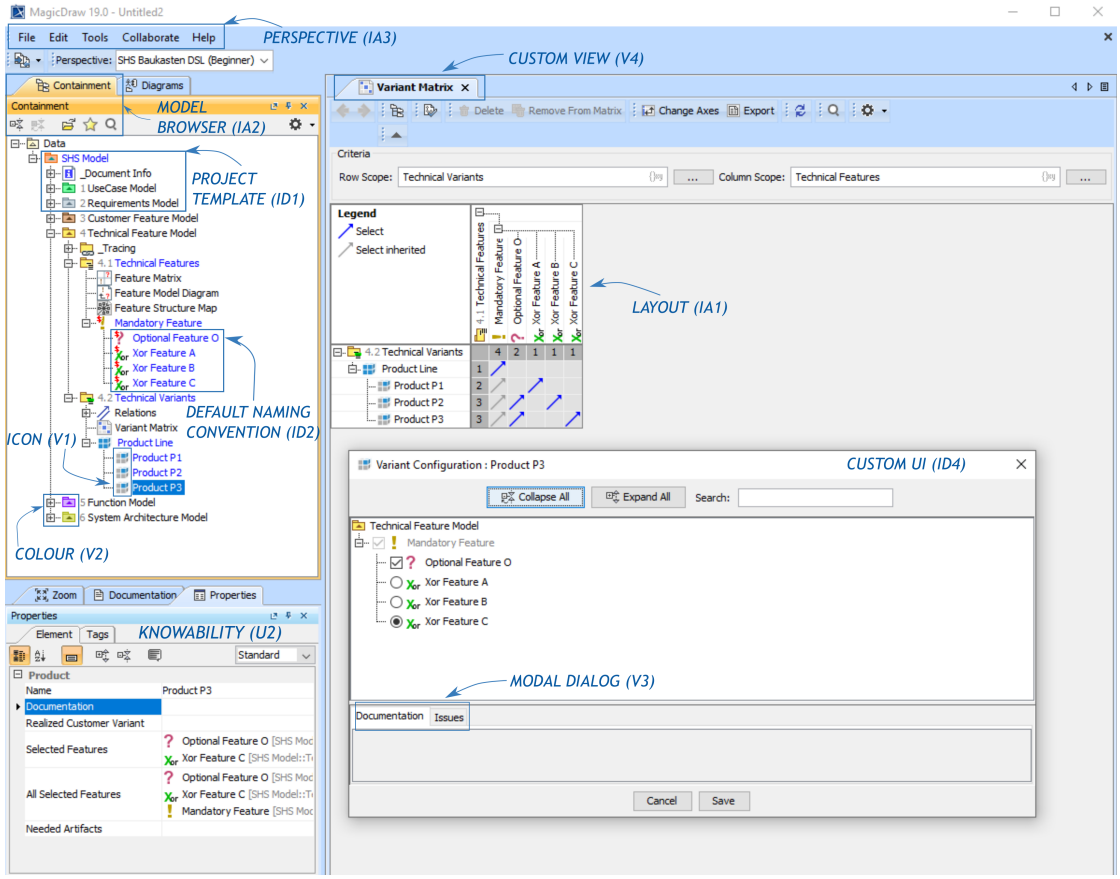


Figure 6.1: Annotations of design decisions in MagicDraw for a feature model consisting of (1) product lines and products, (2) mandatory and optional features, and (3) a UI for variant configuration. The figure shows enhanced aesthetics of models, the structuring and organisation of model elements, the interactive aspects focussing on the cognitive dimensions, and the various usability aspects that complement the design decisions. Figure adapted from [GJRR22a].

6.3 Categorisation of UXD in MagicDraw

In this section, we discuss the categorisation (Figure 6.2) of the individual design decisions [Tid10] for graphical DSMLs in MagicDraw. These design decisions address some of the most important aspects that language engineers must consider in addition to building the syntax and semantics of the language itself. Often, these UXD aspects will complement the syntax or the semantics of the language, which only improves its usability.

Each design decision is provided with a rationale that elaborates the reasoning behind the benefits that the decision brings about. The design decisions and rationales are based on feedback from practitioners and domain experts in various industrial projects studied during the course of this thesis. Further, case studies in Chapter 8 detail the implementation of these design decisions in actual modelling projects. In the following, the design decisions are labelled with an identifier to distinguish them based on their respective categories.

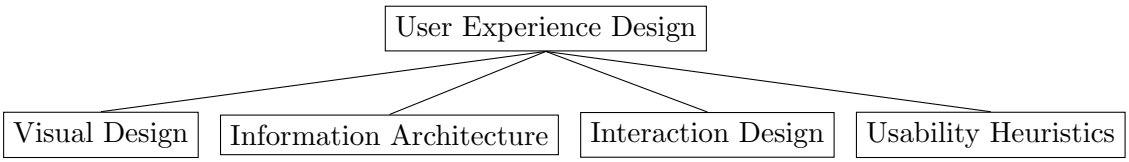


Figure 6.2: Categories defined for the various design decisions that must be considered for a good UX in DSMLs.

6.3.1 Visual Design

Visual designs are any design decisions that further enhance the aesthetics or the look and feel of models and model elements and indicate ways, including visual notations, in which they are presented to the modellers [Moo09]. They are closely linked to visual notations that have been extensively defined and used in the software engineering world. In the modelling sense, such design decisions directly affect the graphical concrete syntax of the DSML. This means that, as an example, model elements are further configured in various forms, such as icons, colours, appearance, dialogs, as well as their respective properties such as shape, size, and opacity. While the language itself describes the necessary graphical syntax needed for successfully model a modelling scenario, visual designs are defined by language engineers to enhance the various language elements in the graphical concrete syntax to effectively represent the heterogeneous domains that a practitioner is involved in. To this end, the following visual designs, listed in Figure 6.3, convey various DSML constructs in a more visual and appealing manner.

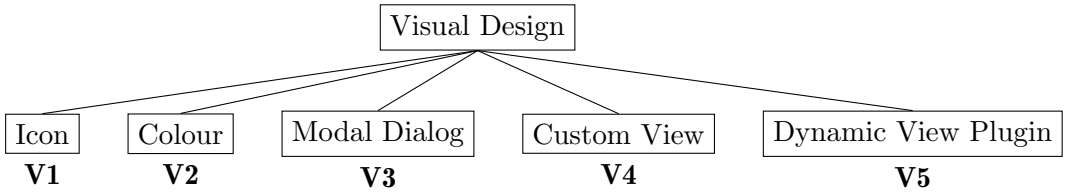


Figure 6.3: Visual design decisions that must be considered for a good UX in DSMLs.

Icon (V1).

An icon is an extra graphical element that is displayed for a model element when it is selected. These icons are configured to be displayed automatically on a graphical modelling canvas of a modelling tool. An example of icons is shown in Figure 6.4.

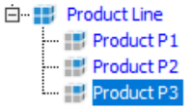


Figure 6.4: Different icons for representing different model elements.

Rationale: Icons are beneficial in differentiating various model elements of a DSML as they are designed to convey real-world representations of a specific abstraction that carries some kind of meaning to a user. For example, a mandatory feature in a feature model is represented with the icon ! , while an optional feature in a feature model is represented with the icon ? . Although it must be noted that designing icons is considered a challenge for language engineers, as they may often not possess the relevant design skills.

Colour (V2).

Applying a colouring scheme to a model element in a DSML enhances the appearance of a model element through a specific colour that generally tends to invoke a reaction from a user. An example of different colours for distinct model elements is shown in Figure 6.5.

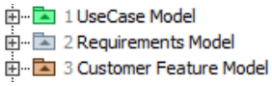


Figure 6.5: Different colours used for representing different model elements.

Rationale: Colouring schemes are meant to invoke reactions from users in a positive or negative way. For example, the usage of red colour generally tends to invoke a feeling that something is not right with the currently observed model and that it may contain an error or a warning. An example of this is using a red coloured question mark for detailing some error within a package containing models . This leads to an increase in the user attention for the current models [BN07] which subsequently differentiates it from other model elements that are *well-formed*.

Modal Dialog (V3).

Modal dialog is a graphical control element which shows information to users that help them make appropriate modelling decisions. For example, a modal dialog is displayed to modellers that shows them a list of issues in a current model. They are also used to show static documentation for a model that ensures users are informed about the currently selected models. An example of a modal dialog showing issues in the model is shown in Figure 6.6.

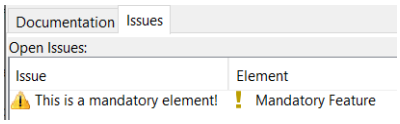


Figure 6.6: A modal dialog listing issues.

Rationale: Modal dialogs are used to channel relevant information to a user when a user's immediate attention is required. This means that such dialogs often interrupts a user's workflow, which is counterproductive. On the other hand, delivering such information is extremely useful in correcting various issues with the models that would be hard to find when the models get more complex.

Custom View (V4).

A custom view is a kind of visual representation of the existing textual information of certain models. These kinds of information could be displayed in the form of matrices, tables, UML, or free-form diagrams. Such a view is directly integrated within the modelling tool itself, thereby displaying users with a way to better visualise the textually stored model content. An advantage of such a view is that certain actions on the models are directly performed such as establishing relationships between two different model elements from two different DSML building blocks. An example of a custom view displaying an impact map is shown in Figure 6.7.

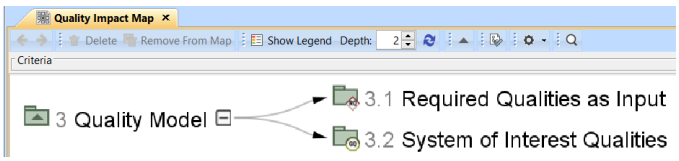


Figure 6.7: An example of a custom view that shows an impact map.

Rationale: Custom views show textual model information visually using specific diagrams that internally store the model information in a particular format or data structure. Each of these diagrams serve a very particular purpose. For example, matrices are

used for optimisation problems, such as solving linear problems of relating one model element to another. Tables on the other hand show static model information for faster readability across the rows and columns. Therefore, language engineers must consider using suitable custom views based on the kind of information that is desired to be seen as per the language definition.

Dynamic View Plugin (V5).

A dynamic view plugin in MagicDraw is a GPL, e.g., Java, based plugin that enables dynamic filtering and display of model specific information directly on UML diagrams or any DSML specific custom views. These plugins are used to show additional items such as legends or annotations that help identify various parts of a model and allows filtering the model information displayed on the diagram. An example of a dynamic view plugin for filtering products belonging to a specific product line is shown in Figure 6.8.

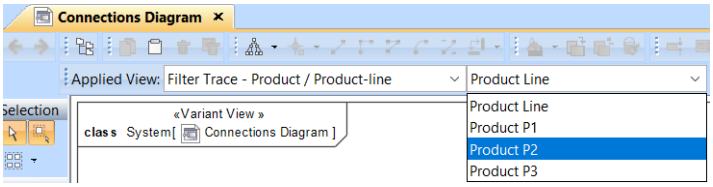


Figure 6.8: A dynamic view plugin that filters products in a product line.

Rationale: Dynamic view plugins assist users in focussing especially on the model information. As an example, such a plugin is used to enable or disable the power supply unit models in a complex power system. Further, various connections or ports that exist in the model are also filtered for display to allow the users a more focussed view of their models. Legends and annotations are beneficial in quickly displaying and removing model information for different custom views that are based on certain filters. They are enabled for toggling which allows users to quickly check their models in one state or the other. Dynamic view plugins work effectively with custom views, as such views are specially tuned to improving the experience of a user in understanding only specific parts of the system without the additional noise around such systems.

6.3.2 Information Architecture

Information architecture is the practice of organising and structuring the constructs of a graphical DSML such that the overall architecture of the DSML is improved in easily finding such constructs that users immediately use [dLSPF16]. In other words, this solves the problem of spending tedious amounts of time in searching and organising models. This is especially true in the case when heterogeneous systems grow in complexity, which means complexity in modelling also increases. Improving the overall information

architecture of a DSML is important because not every modeller maybe a domain expert and even novice users must be able to model with ease. As domains evolve and more and more concepts and functionalities are added to a DSML and the modelling tools, there is a growing need to consistently alleviate the concerns of users as to how a DSML is presented, how the constructs of the DSML and the functionalities of the modelling tool is preserved, organised, and structured. To this end, we identified the following information architecture design decisions along with stakeholders of different Siemens projects, listed in Figure 6.9, as important decisions to organise and structure DSML constructs in a better way.

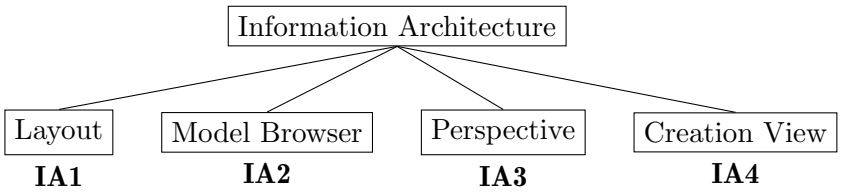


Figure 6.9: Information architecture design decisions that must be considered for a good UX in DSMLs.

Layout (IA1).

The layout determines which model elements are positioned where according to the context of use of a DSML and their use on custom views and domain-specific diagrams. In general, the placing of model elements on a custom diagram may not seem substantial, but impacts the way a modeller thinks. This is because some aspects of where notations are configured in software engineering products have become ingrained in engineers and are just convenient and natural to consider [Moo09]. An example of a layout involving the positioning of input ports on the left side and output ports on the right side of elements is shown in Figure 6.10.

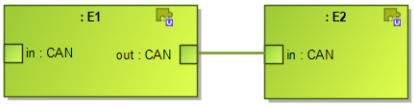


Figure 6.10: Layout involving the positioning of input (left side) and output (right side) ports.

Rationale: By placing model elements on specific areas of custom views or DSML diagrams, modellers get a better overview of the overall structure of their models in complex modelling scenarios. For example, it is just natural to position input ports on the left side of a model element, and the output ports to the right side of a model element

as the general design principles in software engineering denotes inputs and outputs in the same way. Of course, the reverse is also true and is possible in certain scenarios, therefore language engineers must consider the requirements of the stakeholders and design such layouts accordingly that is best suited for their modellers.

Model Browser (IA2).

A model browser in MagicDraw is a visual representation of the list of elements that exist in the modelling tool and is beneficial in displaying the hierarchy of models and model elements in a DSML project. In other words, a model browser is considered a hierarchical navigation browsing mechanism that is used to manage the model data. In MagicDraw, the model browser is configured to display models belonging to either a single DSML project or models from different DSML building blocks at different levels of granularity. A model browser displaying different models is shown in Figure 6.11.

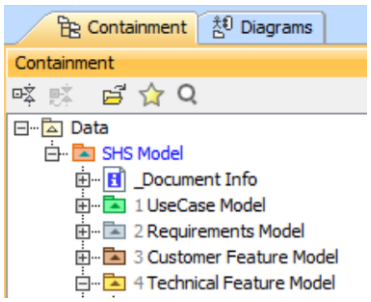


Figure 6.11: MagicDraw model browser shows the models in a project containment tree.

Rationale: Modelling involves working with a lot of different, yet intersecting, domain concepts. In this regard, navigating, finding, or arranging model elements in complex DSML scenarios is often very challenging and consumes a lot of effort and time of the modellers. A model browser is designed to solve this problem by rearranging and providing model elements using a sound hierarchical structure that ultimately helps in organising these individual models. A cross cutting concept is observed for model browsers with visual designs as they are able to display models with the relevant icons (Section 6.3.1). Further, the model browser prevents models to be mixed up during modelling, by allowing common types of elements to be grouped together under a common construct.

Perspective (IA3).

Perspectives, as the name suggests, are used to display only those set of modelling language constructs or modelling tool functionalities that are required by a certain kind of user. This means novice users must be presented with lesser number of complex DSML

constructs, while advanced users may be presented with more number of such constructs. By having multiple perspectives, cluttering of irrelevant information is avoided. For example, a functional view must only discuss the functional composition aspects, whereas a deployment view must discuss only the distributed deployment of a system. Perspectives are considered a kind of view and is different than custom views, where perspectives are used to change the existing display of DSML constructs or tool functionalities. An example of different perspectives that users see is shown in Figure 6.12.

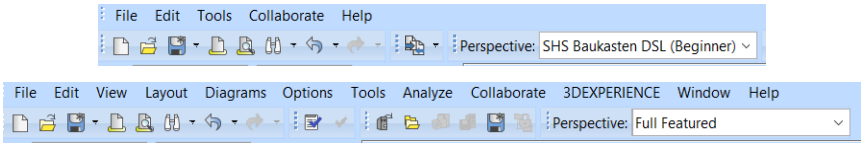


Figure 6.12: Perspectives that show either (top) a reduced set of functionalities, or (below) a detailed set of functionalities.

Rationale: Novice modellers that are not well trained or experienced with a certain DSML often lack the know-how in how to start their modelling. Such users must not be subject to an extensive list of domain concepts that will overwhelm them. Instead, limited constructs of the DSML or tool functionalities help them better onboard them in their modelling. As they get more experienced, therefore reaching advanced stages in their modelling, the same users tend to use more advanced concepts that are present as part of the DSML. Perspectives help such users in being presented with only relevant concepts at different stages of their modelling. As an example, a perspective for a system architect may be significantly different for a system modeller.

Creation View (IA4).

A creation view is an additional window, pane, or a view that shows logically grouped language elements, UML diagrams, and custom views that are created on the model browser. Often in large DSMLs with a large number of intersecting domain concepts, many model elements are created to describe a certain domain. A creation view helps organise these model elements into their respective subcategories to ensure that there is a clear distinction between domain concepts that belong to one subsystem and those that belong to another subsystem. As an example in Figure 6.13, only product lines and products are created as part of a creation view.

Rationale: DSML elements that are part of the same logical group must ideally not be part of other logical groups. This is necessary to prevent inconsistencies in structuring and organising the constructs of a DSML as it often leads to disorder and mixing of domain concepts from different subsystems. One such example is that the model elements and custom views of the functional and non-functional requirements must only exclusively be part of a logical grouping that involves a requirements specification, meaning

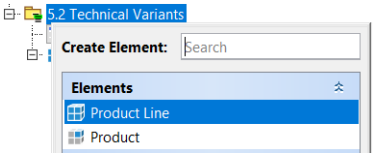


Figure 6.13: An example of a creation view for product lines and products.

a DSML building block for modelling requirements in a project must consist of creation views that contain the concepts specific for requirements only, and not feature models for example.

6.3.3 Interaction Design

Interaction designs are design decisions that assist users in interacting effectively with the constructs of a graphical DSML with a focus on the cognitive aspects [Nie94, BBC+01]. These cognitive dimensions are aimed at a kind of analysis for discussing the characteristics of a system and for providing a rather lightweight approach in improving the overall usability of a system on a design level without considering all the syntactical details of a DSML. These designs subsume characteristics such as auto-completing model names, automatically instantiating models in a DSML project, and transforming models into other formalisms [KMS+09]. Therefore, largely these design decisions focus on the behaviour of the DSML constructs and not the actual implementation of DSML. To this end, we identified the following information architecture design decisions along with stakeholders of different Siemens projects, listed in Figure 6.14, as important decisions in achieving better cognitive support for modellers.

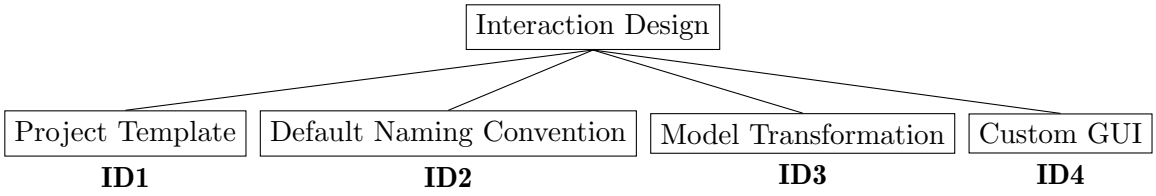


Figure 6.14: Interaction design decisions that must be considered for a good UX in DSMLs.

Project Template (ID1).

A project template is a customised predefined DSML project pattern that is configured at the start of a new DSML project thereby serving as a starting point for a modeller. Such a kind of automatic instantiation of a project template with predefined model

elements allow users to easily understand its conformity with the overall architecture of the domain in consideration. As it allows a predefined set of models to be created, information architecture designs such as layout of models, perspectives, and creation views are configured as part of this start up configuration. An example of a predefined project pattern for Siemens Healthineers DSML is shown in Figure 6.15.

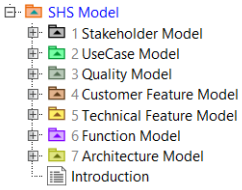


Figure 6.15: A predefined project pattern containing models for the Siemens Healthineers DSML.

Rationale: At the start of every DSML project there is often an uncertainty for modellers in how to create a robust, well-organised model that represents effectively a domain. Such modellers may also lack a kind of know-how on how to simply start in their modelling, meaning which domain concepts to generally consider. Therefore these project templates allow users to automatically instantiate a model, thereby making the effort needed to create a model for this domain a bit simpler. One example is an automatic creation of a basic traffic light model with the states and the transitions already configured for the respective state machine [GKR+21].

Default Naming Convention (ID2).

Configuring a default naming convention within a DSML allows default names or numbers to be automatically assigned to the respective model elements. Such an automatic naming convention is used to automatically complete the naming of a model element based on the name convention that was configured during the DSML development. An example when default naming conventions are used for product lines and products is shown in Figure 6.16.

Figure 6.16: Names and identifiers are assigned automatically to product lines and products.

Rationale: Assigning relevant domain specific names, labels, or numbering model elements along with their names helps distinguish between various model elements. This is particularly beneficial for preventing naming conflicts, that are also detected by context

conditions, but simply is an overhead. For many complex systems, the non configuration of such naming convention cascades further down the chain and ultimately adds unnecessary times needed to debug the system or the models. For example, configuring a mandatory feature as "Mandatory Feature x", with x being a combination of the current model name and an automatic incremental number, allows not only an automatic completion of the name, but also a naming convention that is relevant to the domain of feature models.

Model Transformation (ID3).

A model transformation is described as a transformation of a model into another formalism [HRW15]. As requirements for different systems and software aspects change, the corresponding models for such systems must also change according to certain kinds of refactoring, evolution, and refinement. Model transformations allow model-based evolution, model refinements preserve the existing models but adds various informational details, and refactoring improves the overall architecture of the models while still preserving its behaviours. An example for transforming a model element port into another metaclass is shown in Figure 6.17.

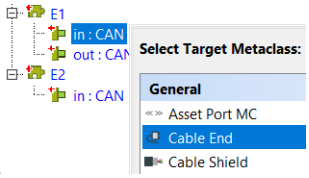


Figure 6.17: Transformation example of a model element through new metaclass selection.

Rationale: Model transformations are beneficial in continuously evolving models in a system. This means it is important to understand the semantic differences between the refactoring steps. Further, as modelling tools constantly adapt to newer technologies, a more sound transformation method is needed to transform the concepts of a model into other formalisms during design time. One such example of transforming a model from an optional feature to a mandatory feature in a feature model based on certain changed requirements.

Custom Graphical User Interface (ID4).

A custom GUI is programmed using a combination of frontend frameworks, such as Java Swing, and a GPL that is supported by the modelling tool. These GUIs is used to access model elements and also used to edit properties of specific model elements of a DSML. Such GUIs help configure and enhance functionalities that are not directly usable on the

modelling tool. A modal dialog, for example, is configured using a GUI. Designing such a GUI is also beneficial in retrieving current model information which is discussed as part of a recommendations engine in Chapter 7. An example a custom GUI for configuring a product line is shown in Figure 6.18.

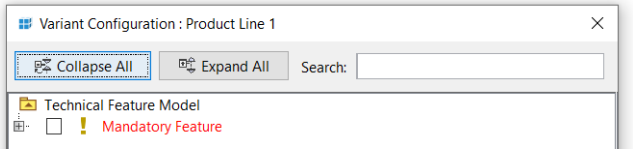


Figure 6.18: A custom GUI for product line configuration.

Rationale: Modelling tools may or may not provide all functionalities to every kind of user. This hinders modellers in using certain tool functionalities that would be necessary for them in achieving a specific modelling scenario. A custom GUI enhances the existing functionalities of a modelling tool such that the capabilities are greatly enhanced. An example of a custom GUI could be to design a configuration window in a model which allows various features in a feature model to be selected or deselected.

6.3.4 Usability Heuristics

While visual designs, information architecture, and interaction designs discussed design decisions in detail, usability heuristics are described in a way that is provides the necessary guidelines for language engineers to consider when developing graphical DSMLs, so that it is ultimately the modellers who are benefited in their modelling. These usability heuristics are defined in a way it gives modellers a greater sense of effectiveness and satisfaction in using their DSMLs [PRBCZ17, PZBdBC18]. This means a good UX of a DSML is generally also dependent on good usability heuristics in a specified context of use. It must be noted here that while these usability heuristics are not strictly considered design decisions, they naturally complement the design decisions that were earlier described. Today, many graphical modelling tools have an underlying API that help support additional functionalities of a DSML, and therefore describing usability taxonomy attributes parallel to UX studies in APIs [MRAR20] is considered important. While many such usability heuristics are considered in general for modelling languages, we consider the following "ilities", listed in Figure 6.19, and based on the taxonomy proposed by [ARVGMRMB09]. Based on various discussions between language engineers, practitioners, and domain experts on different Siemens projects, we have focussed on the following usability heuristics, as we believe these cover the main aspects of general usability in relation to graphical modelling.

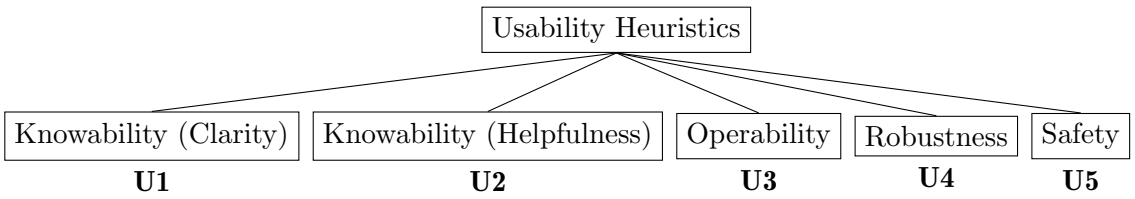


Figure 6.19: Usability heuristics that must be considered for a good UX in DSMLs.

Knowability (Clarity) (U1).

Knowability is a usability heuristics that language engineers must consider to make the constructs of the DSML self-explanatory and easy to understand. In other words, disambiguation between various constructs in a single domain must be avoided.

Rationale: Clarity is a quality of a DSML that makes it easy to understand and comprehend. This is often a challenge modellers face during modelling complex scenarios either because the tool is inadequate or the DSML has not been configured in a way that it easily describes the domain-specific concepts. Therefore, a clarity in terms of model names, model types, structure of the models, logical grouping of such common model elements, easily comprehensible modelling constructs, position of the models on a graphical modelling canvas, and the ability to design model elements for a specific purpose, is crucial to improving the readability of a DSML.

Knowability (Helpfulness) (U2).

The helpfulness of a DSML is to describe the DSML constructs in a way that it is able to provide helpful annotations, documentations, and descriptions of model elements such that users have an overview of the DSML constructs. It would also be helpful to identify which elements have evolved beyond the point of use and must be considered deprecated elements.

Rationale: The lack of documentation and description of models and model elements within a DSML leads to a reduce in productivity for modellers as they have to often search through handbooks and other training materials that contain such model information. Such users find it tedious to comprehend the meaning and usage of DSML constructs by themselves. Therefore, providing ample documentation for models with sufficient examples for how they are used reduces the effort that is needed to model extremely complex modelling scenarios and restricts different kinds of mistakes during modelling.

Operability (U3).

Operability ensures that the graphical DSMLs provide the desired domain-specific functionalities and that DSML engineering must ensure that mechanisms for DSML con-

structs to be reusable, extensible, and composable, such as with DSML building blocks, are provided.

Rationale: Modellers must be presented with constructs that are not only relevant to their domains, but also is capable of performing all the desired functionalities of the DSML. This means model elements must be universally recognisable and reusable either through visual designs, or through other means such as language composition that help support model transformations.

Robustness (U4).

A graphical DSML must be foolproof, meaning the DSML in combination with the modelling tool it is built in, must not contain any inconsistencies such as bugs, errors, or vulnerabilities that could potentially compromise the system or its subsystems. Therefore, robustness is a property of both the DSML and the accompanying modelling tool.

Rationale: DSMLs must be subject to thorough checks after it has been engineered to detect for any runtime errors. This ensures that the DSML does not encounter any significant errors that hinders the usage of the DSML. The constructs of the DSML must be error free and checked for potential vulnerability leaks.

Safety (U5).

This heuristic defines that the graphical DSML must not actually compromise the data or the assets that belong to a user. It is important the the DSML is considered safe and that there is no violation of protection rights or any kind of legal issues.

Rationale: The data belonging to a user must not be directly or indirectly exposed to third-party entities. This means that the licenses must be up to date, legal actions are already taken care of during deployment of DSMLs, and the personal information of users and their data protected according to their local governments.

6.4 Scope of UXD

The list of design decisions and usability heuristics described in Section 6.3 is certainly not exhaustive but aids in describing a good UX for a DSML. There must be flexibility in consideration for new design decisions that language engineers must consider to enhance their DSMLs. This is because UX is consistently changing based on user's opinions and the overall topic of UXD is very wide and covers many scenarios. Additionally, combining quality management standards such as ISO standards along with the ergonomics of human-system interaction and usability considerations provides best practices to develop better graphical DSMLs. Often projects follow a very tight development schedule so most of the focus is on developing the DSML and its functional aspects such as its syntax. There is often constraints in resources and budget that introduces

trade-offs that language engineers must carefully consider. This means while consideration is made for modifying the aesthetics of a modelling language, design decisions that improve the general usability of graphical DSMLs must be considered. The following three questions must be answered closely by language engineers and project stakeholders during graphical DSML development to address usability concerns:

- **UXD-Q1**: Can a specific design decision satisfy a user's modelling needs and help them reach their goals?
- **UXD-Q2**: Does a design decision cause any potential disputes between the DSML constructs and with the functionalities of the modelling tool in consideration?
- **UXD-Q3**: Can the design decision be considered for a particular domain aspect, is non-subjective, and is related to a domain system or subsystem?

These answers help language engineers understand the extend to which design decisions must be considered to aid in a better UX and are described in more detail with a specific industrial example in Section 6.2. Often it is a mindset change that language engineers must undergo, for thinking more like a user is beneficial in designing DSMLs that users will base their successful models on. In general, those design decisions that are not relevant to a domain, compromises the quality of a DSML, and is not aligned with all stakeholders must not be considered for implementation in a DSML.

6.5 Evaluation of the Industrial Example

This section illustrates the implications of the described design decisions and guidelines using the industrial example of a feature model [BEK+19] DSML described in Section 6.2. The enhancing capabilities of MagicDraw through customisations allow design decisions such as icon (V1), colour (V2), default name (ID2), and creation views (IA4) to be directly integrated into the language definition, meaning it directly affects the concrete syntax. The MagicDraw Open Java API is used to configure the modal dialogs (V3) and the custom views (V4) are configured using the customisations provided by MagicDraw. The interaction designs (Section 6.3.3) and the information architecture (Section 6.3.2) designs are also directly configured using MagicDraw's settings. To this end, various plugins are created for a DSML such as the dynamic view plugin (V5) that is also programmed using Java. These plugins are also capable of transforming models (ID3) into other formalisms. Finally, a custom GUI (ID4) is programmed using Java Swing and is incorporated as part of the feature model DSML.

We show the substructures of Figure 6.1 and evaluate the example in this section. Figure 6.20 shows the model browser (IA2) as a functionality that MagicDraw provides by default. The model browser is responsible for hierarchically organising and structuring the feature model DSML constructs so that a modeller easily navigates through

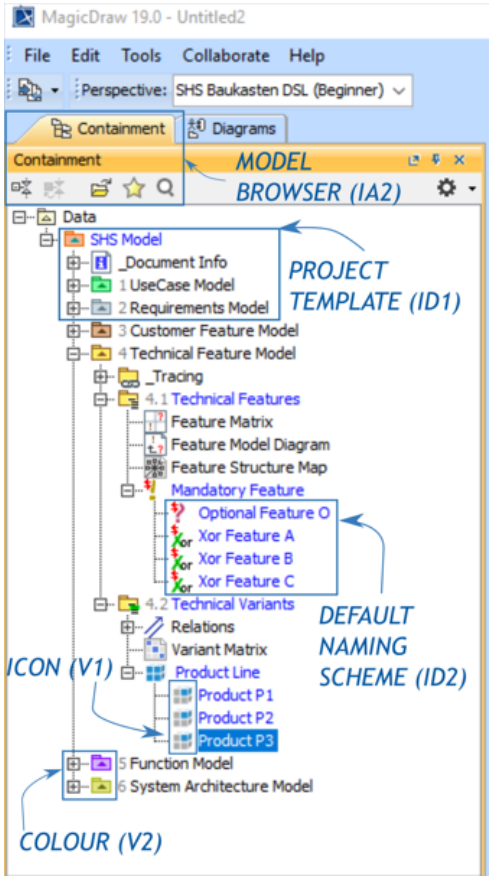


Figure 6.20: A part of the feature model showing design decisions such as icons, colours, default naming convention, project template, and a model browser.

the various models and find their respective models. The project template is the predefined feature model that is automatically created (ID1) within the model browser upon instantiation of a new feature model project. As the relevant models remain in their individual building blocks, this takes a step in addressing question **UXD-Q1**. In this predefined project template, a default naming scheme (ID2) is configured which automatically assigns names and numbers to the different models, such as "2 Requirements Model" and "4 Technical Feature Model". This is beneficial specially because modellers may often forget to assign relevant naming identifiers. Further, they are also guided in a way to create models sequentially (U1) to avoid immediate conflicts with other DSML parts. The individual models are also configured with appropriate icons (V1) and colours (V2) for different features in the feature model to provide the necessary distinction to other models. Language engineers must consider designing appropriate icons and colours

to such model elements so that it better represents their domains. As an example, here the "Product Line" shows four blue cubes whereas the "Product P1" (product) shows only one blue cube and three white cubes, thereby differentiating between a product line and a product. In a similar way, the design differences between the icons of different feature types, such as mandatory feature (with an exclamation mark) ! and an optional feature (with a question mark) ? shows the distinction between these features in a feature model. The designs for the icons must be designed similar to existing feature models that are currently in use that fully support product line engineering such as pure::variant [Beu08] for consistency. As such icons may not be known to modellers, the models themselves must also be accompanied with sufficient documentation (U2) shown in Figure 6.21.

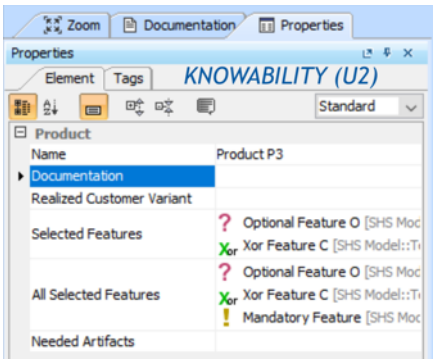


Figure 6.21: Properties of a product showing documentation and the related features.

A perspective (IA3) for a beginner in this DSML (SHS Baukasten) was shown in Figure 6.1. This can be later changed to advanced if the current user is more knowledgeable in modelling with the DSMLs and needs more functionalities for modelling complex scenarios. Here, language engineers appropriately configure the functionalities to restrict or allow modelling constructs or MagicDraw functionalities for different kinds of users. In the example, since a beginner perspective is chosen, only certain basic toolbar options in MagicDraw is made visible to the user. Advanced perspective users configure functionalities such as cloud collaboration for migrating projects, validating advanced context conditions, and performing a merge of two different projects. These perspectives avoid creating any conflicts between the feature model DSML constructs or the existing MagicDraw functionalities by separating the concerns of the tool and the concepts provided by the DSML, thereby taking a step towards addressing **UXD-Q2**.

As part of a custom view (V4), a variant matrix is displayed in Figure 6.22. This shows the various product lines and products and its relations to the corresponding features that were configured. Here, a matrix definition allows the support for an architecture design (IA1), which shows the layouts and the positioning of the various technical variants (rows of the matrix) and technical features (columns of the matrix). The legend infor-

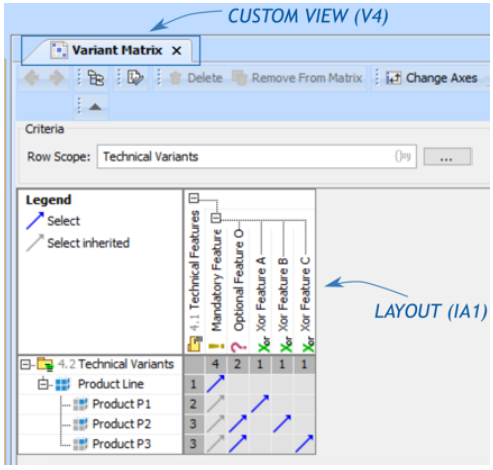


Figure 6.22: Custom view and layout of the feature model.

mation indicates the description of the relations between the variants and the features and is configured using the dynamic view plugin (V5). By default, this is not possible in MagicDraw since custom views such as matrices and tables do not allow information pertaining to legends. The configured variant matrix is also interactive, meaning users immediately select an empty cell and configure a relationship, or select an existing relation on a cell and quickly remove it. As the matrix is configured for SPLE and it enables concepts for feature modelling, it therefore takes a step in addressing **UXD-Q3**.

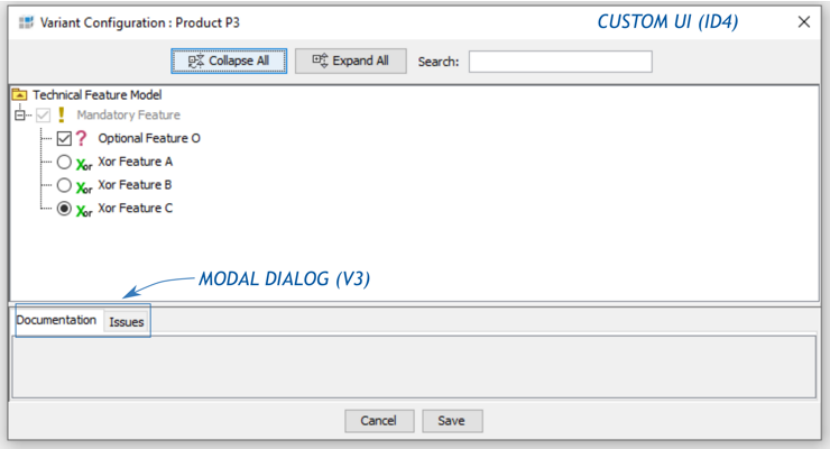


Figure 6.23: Custom GUI and a modal dialog for product line configuration.

Figure 6.23 shows a dedicated variant configuration GUI (ID4) that is displayed to users allowing them to select the proper configuration for their product lines and prod-

ucts. Depending on user selection, relations between the variants and the features are automatically adjusted. Further, the GUI is configured to allow the collapsing and expanding of the various features on this custom GUI, and also includes a search bar to quickly search features when the model gets complex or the list of features are very long. A modal dialog (V3) consisting of the tabs for documentation and issues is directly integrated on the variant configuration (ID4) for listing additional information about the variants or the selected features or a combination of both. The usability heuristics are not specifically displayed on the figure but is an integral part of the DSML. They ensure that the DSML constructs remain self-explanatory (U1) through relevant an helpful documentation or notes (U2). Finally, language engineers must consider validation and verification of the entire DSML such as checking the Java code for the dynamic view plugins that they ensure robustness (U4) and safety (U5) of the DSML, the modelling tool, and also the data and assets of the modellers.

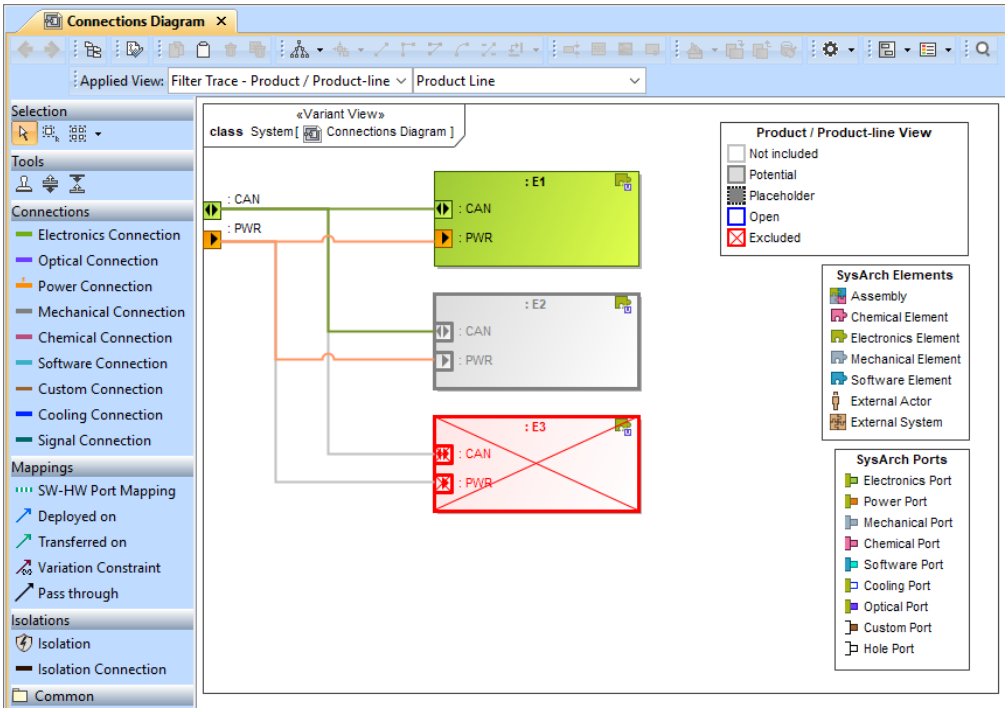


Figure 6.24: An example of a custom view (V4) with a dynamic view plugin (V5), for filtering product lines and products. Element E2 is marked as potential, meaning it is optional for products, but undecided on the current product line. Element E3 is marked as excluded with a red cross, hence the respective connections are greyed out. Figure taken from [GJRR22a].

Figure 6.24 shows an example of a custom view (V4) for a feature model product

line that is integrated with the dynamic view plugin (V5). In this view, the "Variant View" allows a user to configure various system elements, ports, connections, and their mappings. The structuring of the model elements (IA1) inside this view is configured by language engineers. This means that the placement and layout of the input ports from the left side of the view to the elements in the middle of the view is preconfigured, to provide a better first impression of the system or the subsystem. However, this is not a strict restriction and during design time, modellers are free to move around and restructure the model elements according to their modelling needs. The applied view in this case is changed dynamically upon selection of a product line or product for a feature model at the top of the diagram. In this example, selecting the product line means the respective "Connections Diagram" (V4) is updated with the appropriate elements which shows the electronic element E1 is positioned (IA1) with the use of a custom GUI (ID4). This means the electronics (green) and power (orange) connections to E1 (V2) is also made visible to the product line. The element E2 is marked as potential as it may be later configured for a product in the product line, therefore the element is greyed out. The element E3 is marked as excluded (greyed (V2) and marked with a red cross (V1)) such that it cannot be included in any subsequent products in the product line. This kind of configuration is helpful when a low cost variant of a product with a fewer number of features are required to be built.

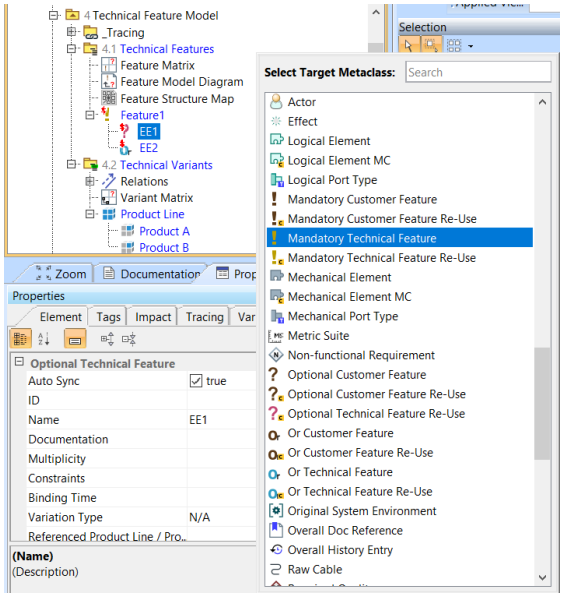


Figure 6.25: An example of a model transformation (ID3) for refactoring an optional feature to a mandatory feature. Figure taken from [GJRR22a].

Refactoring of models, shown in Figure 6.25, by modifying the underlying stereotype

of a model leads to a way of model transformations (ID3). To support such refactoring a modal dialog (V3) is created by language engineers. The figure shows such kind of a model transformation for refactoring an already defined optional feature in a feature model to a mandatory feature by selecting its appropriate target metaclass. The example shows that a model transformation is achieved during design time, however extra care must be taken by language engineers and the modellers so that the risk of losing incompatible properties during such transformations is minimised, meaning a DSML must not ideally allow refactoring a mandatory feature to a product, as their usage is completely different. Such risks are informed to users using an additional modal dialog (V3) when a model transformation is performed.

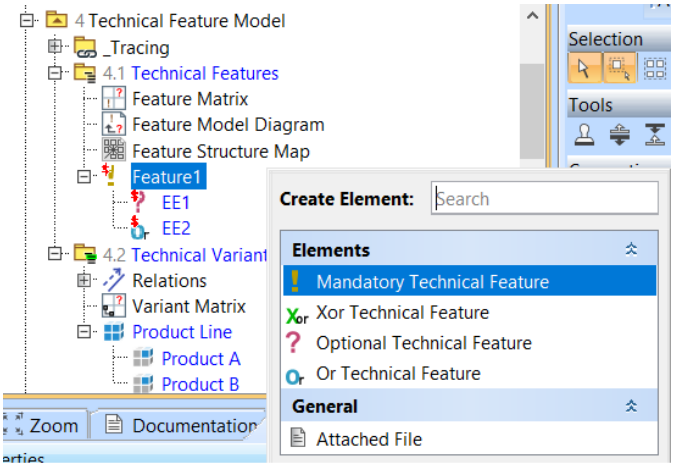


Figure 6.26: An example illustrating the creation view (IA4) where the creation of a model element inside a Technical Features package allows only a mandatory, or, Xor, or an optional feature to be created. Figure taken from [GJRR22a].

Figure 6.26 shows an example of a creation view (IA4) that lists the elements that are created as part of a feature model, namely mandatory, optional, or, and Xor. This view presents modellers with a logically grouped and restricted set of features based on the configuration of a DSML. As shown in the figure, the four defined features are only created under the "4.1 Technical Features" package and is not permitted to be created inside any other package. Therefore, creating such features in the "4.2 Technical Variants" package is not allowed. Similarly, only products and product lines (variants) are created under the "4.2 Technical Variants" package and not inside a "4.1 Technical Features" package. This prevents any disorder and mix of elements and allows separating the concerns of individual subsystems and their domain concepts. Here, a default naming scheme (ID2) is also used to automatically assign names and labels to features such as "EE1" or "EE2", or for variants such as "Product A" or "Product B". In reality, however, it is recommended to automatically assign domain appropriate names, such as "Collimator

v1", "Collimator v2", and so on. This serves as a guide to modellers in developing features or variants in a sequential order with the flexibility to modifying the names later during modelling.

6.6 Discussion

This chapter presented definitions for describing a good UX and UXD aspects that language engineers must consider for developing graphical DSMLs. A categorised list of design decisions was detailed for improving the overall UX for practitioners. There exists many development techniques for building DSMLs but often there is little consideration for integrating good UX aspects. As such DSMLs get more complex and harder to use, providing a combination of a methodology along with a modelling tool greatly benefits a user. To this extent, a good UX is integrated by involving all key stakeholders for a DSML project. Here, a kind of an iterative feedback and communication loop must be set up such that modellers, domain experts, software and systems architects, as well as language engineers easily understand all aspects of the specific domain.

Language engineers primarily develop DSMLs and are not experts in UX or even software development. This means they often need to go through additional trainings to get more knowledge of these technologies. However, design decisions that are ultimately taken by language engineers often result in improving the UX since they talk often with the stakeholders. By categorising the design decisions, language engineers that possess different skill sets could achieve a separation of concerns in improving DSMLs. Visual designs help improve the look and feel of the models. Models must be designed to be aesthetically similar to their real world counterparts so that they could easily be understood. Therefore, icons, colours, modal dialogs, and custom views are configured to make the models feel closer to real world abstractions. Information architecture designs help in organising and structuring the constructs of a DSML in a way that improves their findability. Interaction designs assist users in making effective interactions with the constructs of the DSML with a special focus on the cognitive aspects. These design decisions not only assist practitioners in building models from scratch but also provides them with a kind of guidance that is necessary especially for novice modellers. Usability heuristics are guidelines that help language engineers in developing graphical DSMLs for improving the satisfaction of use for DSML users. For these heuristics, the most important taxonomy attributes to consider are knowability (clarity and helpfulness), operability, robustness, and safety of the data of users.

The design decisions presented in this chapter are configured using the customisation capabilities MagicDraw offers, hence the choice of tool. The constructs of the DSML are created as per the requirements of the stakeholders for the projects and the language components earlier discussed (Chapter 4) are easily created and bundled in the DSMLs. Additional enhancements that are not offered by MagicDraw are achieved by deploying

various plugins written in Java. The components are brought together to compose into a single DSML which is used by practitioners. A large variety of design decisions are therefore created that serve various purposes to improve the UX for a DSML. As building blocks represent heterogeneous aspects of various application domains, language components and their design decisions foster a high level of reusability for various constructs of a DSML.

While the design guidelines have been presented mainly for industrial graphical DSMLs, they naturally apply to all kinds of graphical DSMLs, including in research. The chapter focusses specially on industrial languages as they have a greater need for providing good UX to practitioners. The categories of design decisions discussed in this chapter are essential for practical applications such as in the industry while DSMLs in research mostly represent PoCs. To this end, the presented guidelines are rather suitable in industrial contexts. While specific guidelines for textual DSMLs, such as defining custom GUI for generating ASTs from textual grammars is possible, it needs more research. These guidelines are also applicable to other graphical modelling frameworks as they all support graphical DSML development.

Describing a single source of truth for improving the UX is challenging as UX is a relatively subjective topic and practitioners often have a mixed understanding of their domain representations. Research in UX often exceeds traditionally used usability heuristics which is a reason why UX experts often struggle to assign behaviour to DSML constructs. Some proposed definitions of UX are reused for graphical DSMLs but it is too generic or restricted to a specific modelling tool. It is therefore aimed through this thesis that discussion for improving UX in graphical DSMLs will certainly gather more traction in the modelling community. Language engineers must work hand in hand with UX experts along with all the other stakeholders to better describe a DSML. They must also be considered for training in the area of UX for better development of graphical DSMLs. The list of design decisions are certainly not meant to be exhaustive, but simply provide a starting point for consideration to effectively apply UX techniques in graphical modelling. Therefore, research for other modelling tools is considered a future work. This would certain avoid risks introduced in this chapter as being vendor-locked to a MagicDraw implementation. While the design decisions elaborated in this chapter are seemingly particular to MagicDraw, describing them at different abstraction levels as well as with MagicDraw specific terminology helps categorise them into commonalities that must be considered in other modelling tools. Further, such research helps shape, refine, and improve design guidelines that are constantly changing in the ever adapting modelling community. The scope of UX and UXD aspects must be considered by language engineers to probe the trade-offs in relation to project resources and costs. A key benefit in developing and researching industrial DSMLs is that guidelines are proposed and adjusted according to current needs. This chapter therefore presents a first reference point in fostering discussions towards defining better UXD aspects for industrial DSMLs that is ultimately independent of any graphical modelling tool.

6.7 Related Work

Usability driven development of DSLs [BAG18, BNS21] has focussed on evaluating usability [PZBdBC18] during a DSL development process to solve usability challenges, but lacks an overall implementation in complex industrial scenarios. The guidelines presented in this chapter is a result of work along with researchers over the years on a variety of industrial domains. General design guidelines for DSLs and DSMLs exist [KKP+09, BDH94, Fra13], but they apply mostly to DSLs whereas the guidelines presented in this chapter focus on industrial graphical DSMLs intended for practitioners. All stages in a DSML development lifecycle is generally not considered during experimental usability evaluations [KTK09]. However, the guidelines presented in this chapter explicitly mentions that language engineers must involve all relevant stakeholders of a project early on to avoid any dissatisfaction later on (also described previously in Figure 3.4). Other techniques such as collaboration and crowdsourcing for designing graphical notations of DSLs have been proposed [BCCIM17, IC16], which permits a greater flexibility and subjectivity as it is aimed to a large group of users for validating and accepting DSLs. The description of the interaction designs in this chapter are based loosely on the cognitive dimensions of the notations framework (CDF) [Gre89, BBC+01]. This allows integrating various cognitive aspects such as *hidden dependencies*, *diffuseness*, and *viscosity*. The principles of human-centred design defined in ISO 9241-210 [ISO10] is broadly relevant to any graphical DSML development process, and evaluations against usability using these approaches has also been studied [PRBCZ17]. The chapter presents certain DSML usability heuristics that are proposed by [MRAR20] which are applicable and important in industrial DSMLs. [ABC+17] discuss the various challenges and a general direction in which UX for model-driven engineering approaches is headed, but do not discuss specifics in relation to the general usability and improving UX in industrial graphical DSMLs that is presented in this thesis.

Chapter 7

Model-Aware Recommendations for Industrial DSML Users

In this chapter, the integration of a DSML, and its building blocks, with methods that assist users in their modelling to further improve UX for DSML practitioners is discussed. Previous chapters elaborated composing DSMLs from the viewpoint of language components and their definitions, with a focus on the syntax and the semantics of a modelling language. This chapter presents concepts that details the necessary missing links between a language and the methods that enable users to take the next step in their modelling. The concepts presented in this chapter build on the UX guidelines that were proposed in Chapter 6. As models grow in complexity, practitioners using DSMLs encounter numerous challenges with the lack of effective methods, guidance, and support needed to improve their modelling situation. As languages are built with the sole purpose of providing domain-specific constructs to design models, providing a rather static source of information often leads to endless hunt for modelling information that is relevant to users. This is because a combination of a modelling language along with methods to effectively use the language in a single modelling environment is missing. This chapter therefore provides several methods, that range from providing general training material for the models, frequently encountered problems and their solutions in that domain, actively recommending users to consider the next steps in their modelling, and providing prescriptive process models that improve the overall modelling situation of these users. A set of dynamically changing recommendations is presented to users assisted by configurable general rules that are checked against the current and past models related to the DSML. Some results of this chapter have been published in [GJRR23]. Therefore, passages from the paper may have been quoted verbatim in this chapter.

The goal of a DSML is to move away from traditional documents to an intensive use of models. To solve this challenge, MBSE techniques have been constantly applied over the years both in the academia and the industry [FR07]. While GPLs are not suited to solving domain-specific aspects for systems modelling [PB19, CBCR15], they are used for developing a more complete DSML definition. This is achieved by introducing context conditions written in GPLs such as Java that check the well-formedness of the language. DSMLs are heterogeneously designed and be represented in either a textual, graphical,

or a projectional form [DCB+15, Bet16, Cam14]. Within each such technological space, there exists challenges in how to effectively use the DSML with the provided language workbench or a modelling tool that is suited to both novice and advanced users. Therefore, providing guidance and support to such users is necessary for them to achieve their modelling goals.

In this chapter, Section 7.1 discusses the motivation for developing a guidance infrastructure before Section 7.2 takes a look at the requirements for such a recommendation system. Section 7.3 provides an architectural overview of the integration infrastructure needed to provide various methods to support users. Section 7.4 discusses the implementation of the methodology using MagicDraw and various techniques required to illustrate model-aware recommendations checked against configurable general rules. Section 7.5 describes the applicability of the implementation using a real industrial use case example of a function context model designed by Siemens Healthineers that models functions related to an X-ray collimator. Finally, Section 7.6 discusses the overall infrastructure and implementation benefits and limitations, while Section 7.7 discusses related approaches.

7.1 Motivation

As discussed previously in this thesis, modelling helps in making key decisions at various stages of a software or systems engineering process. DSMLs must be designed in a way that it represents important parts of the domain in consideration. Therefore, the language engineering process must consider both the definition of a complete modelling language as well as methods and concepts that assist users for using the constructs of such a DSML. This is especially important in an industrial environment, as the integration of a DSML, method, and an appropriate tooling support leads to better modelling experience for users [GKR+21]. Small and medium enterprises are often faced with resource constraints [Reg18], for which an integrated guidance mechanism is particularly helpful. Providing such a modelling direction to users help them create not only effective models, but also allow language engineers to develop complex DSMLs with more variability [CGR09]. While guides and documents do exist in some form, such as handbooks or large sets of documents, they are often quickly outdated and do not consider modelling aspects that are very quickly changing in the complex world of modelling. This is true for novice users who often lack the know-how to model with the provided DSML when they are introduced to the modelling language or the corresponding tooling environment. Some of the challenges of methods that guide users in their modelling are:

1. providing a sufficient level of overview that describes a brief summary of the currently designed models;

2. providing training material of various modelling constructs that includes extensive and relevant documentation [Hon13];

3. displaying frequently asked questions by modellers in the recent past for providing quick solutions to model-specific questions;
4. suggesting useful advice in the form of recommendations, such as a model diagram or a model element [ARKS19]; and
5. identifying and suggesting various tasks, activities, or processes using process models that are useful in describing a particular course of action for modelling a specific scenario [Fra10] to enhance a modeller's experience.

Mechanisms to deploy concepts and methods that run in tandem with a DSML have been proposed in previous studies [NFT+10, Roq16]. However, they are either specific to a technological space, or simply too generic to be adapted to other environments. It is also very challenging, and nearly impossible, that language engineers consider every single aspect of a modelling language, that includes rules, standards, or guidelines for a DSML as they often prioritise implementing the syntax and semantics for the DSML. It would therefore be more beneficial if users are actively assisted directly in their modelling environment, rather than search for the corresponding information in a passive source of information. This chapter provides a framework detailing the capture of such methods, and how active model-aware recommendations are made to users. It builds on the concept that the integration of a DSML and a method along with a modelling tool are necessary to alleviate users in a more holistic modelling experience for various domains.

7.2 Requirements for the Recommendation System

To define a recommendation system that captures different methods for providing active model-aware suggestions and techniques to DSML users, language engineers must first define requirements for such a system. In particular, such a recommendation system must be able to provide detailed descriptions of the various models that have been developed. These descriptions must include brief summaries, statements, and any information that is often described in guides, documents, or operator manuals in detail. A recommendation system must be able to collect this information and summarise the descriptions into a concise text that is helpful for DSML users and therefore requires aspects of location, components, and infrastructural details to be initially defined by language engineers. The recommendation system proposed in this chapter supports the following kinds of requirements:

1. The recommender system must (not) be deployed at a specific location in the industrial modelling environment,
2. A location requires that a certain number of components belonging to the recommender system be deployed there,

3. A component requires a certain set of (sub-) components potentially exist in a similar location,
4. Two or more distinct components may not necessarily be deployed at the same infrastructure level,
5. An infrastructure level requires that a certain set of components logically belong to serve a particular purpose for the recommendation system.

By defining these requirements, language engineers can eliminate the need to explicitly express requirements that apply to individual parts of the modelling environment, but rather define requirements that apply to the overall modelling environment. This means by deploying the recommendation system at a particular location, language engineers can reduce the efforts needed to develop and maintain different components. The recommendation system can be suitably deployed immediately at a specific location of the modelling environment if it fulfils these requirements. On the contrary, if the requirements cannot be fulfilled, language engineers must propose a set of modifications that help in the deployment of the recommendation system through, e.g., relaxation of requirements, addition of further components, or infrastructure levels.

These requirements enable language engineer to provide a recommendation system that includes providing ample training material pertaining to models that ensures users do not have to move away from their modelling environments for fetching specific model information. Such training materials could also reside at specific locations in the modelling environment, therefore the language engineer can deploy code to generate information that is generated via large-language models (LLMs) [BKK+23] such as generative pre-trained transformers (GPTs). The decision to provide such information is ultimately left to the language engineers and the stakeholders of the DSML projects. A rule engine must be embedded as a component in the recommendation system that provides for models to be checked against various rules for generation specific recommendations. These rules are defined by language engineers and the stakeholders of the respective DSML projects. Such rules are configured and deployed at specific locations in the infrastructure levels to ensure that rule checks are not completely disjointed from the overall recommendation system. The process of identifying these requirements involved the discussion of various DSML stakeholders in different industrial projects across Siemens, discussed later in Chapter 8. These stakeholders include language engineers who develop the DSML and its building blocks, domain experts in their respective domains such as healthcare, and practitioners with varying levels of experience in modelling from 3-15 years. Further, as these projects involved research discussions to provide state-of-the-art recommendation system, we intensively discussed these requirements with researchers from the academia as well. Overall, these requirements ensure that the technical deployment can either be immediately deployed, or need certain modifications that do not deviate drastically from the modelling environments.

7.3 Architectural Overview of a Recommendation System

This section describes the overall high-level architectural view of the infrastructure built to realise the concepts of providing methods and recommendations to DSML users. Here, an overview of the software architecture of the infrastructure, the MagicDraw plugin, and the corresponding database configuration used for supporting the plugin are discussed.

7.3.1 Infrastructure Overview

The classical three-tiered architecture [ABM96] used to demonstrate the applicability of providing user-centric model-aware recommendations for users is shown in Figure 7.1. Users interact directly with the presentation tier that consists of the GUI. This is also known as the communication layer, as the user is able to interact with the relevant information that is displayed back to them. The GUI (described earlier in the UX guidelines as the **ID4** interaction design in Section 6.3.3) is implemented using a simple Java Swing interface that is developed to gather and display information from the tiers below this communication layer. The tier below the presentation tier contains the logic that is needed to collect and process the information from the GUI and it makes the corresponding requests to the data tier using a set of general and customisable rules. The data that is processed and calculated is returned back as recommendations and methods for a particular model data that is incoming from the GUI. The lowest tier of the architecture is the data tier, where all the information and its relevant links are stored and managed. Training materials, documentation, and specific recommendation related to the models are stored in this data tier. The application described in this chapter communicates between the logic tier and the data tier using API calls. To store the data related to the recommendation application, a MongoDB [BGBV16] database is used. MongoDB is a NoSQL database that stores data in the form of JavaScript Object Notation (JSON) documents [PRS+16], does not require a schema, is distributed, and therefore provides the desired scalability, availability, and reliability of the stored data [MPHH10]. This database is configured to be stored either locally or is configured in a remote server, meaning language engineers effectively manage the stored data and metadata without making any significant changes to a DSML itself. The overall architecture of the infrastructure is presented in a way that the GUI, or the presentation tier cannot directly invoke or communicate with the MongoDB data tier.

7.3.2 Presentation and Logic Tier

Previously in Section 3.4, we took a look at the capabilities that MagicDraw provides for enhancing the capabilities of a DSML with the introduction of customisations on various DSML constructs. While modelling workbenches, by default, provide an extensive set of model-aware validation rules [EvdSV+13], they do not provide DSML-centric recommendations detailed in this chapter. A reason is the nature of DSMLs being highly tailored

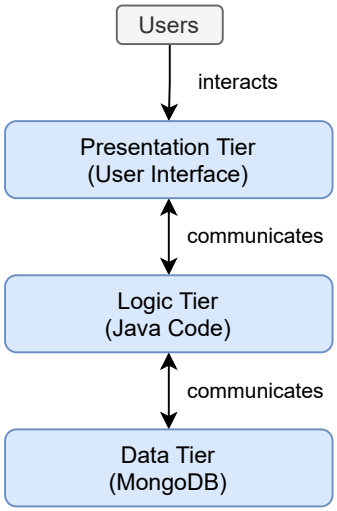


Figure 7.1: The three-tiered architecture for generating model-aware recommendations for users. The presentation tier is the UI created using Java Swing, the logic tier is where the business logic and the set of rules are configured using Java code, and the data tier is the database where the data related to the user-centric recommendations is stored and managed in a MongoDB database. Figure taken from [GJRR23].

for a specific domain where a particular modelling scenario cannot be predetermined at a general-purpose level. In the presentation tier, the MagicDraw's Open Java API is used to develop additional functionalities in the form of customisation capabilities, to provide model-aware information at various stages of the current modelling situation of the users. As part of this integration infrastructure, a MagicDraw DSML plugin file is created that is archived into a single `.zip` file and is installed on a MagicDraw instance. This archived file consists of internal plugin files that provide the additional functionalities for a DSML. The customised GUI that is created to enable model-aware recommendations for users is present as part of one of the described plugin files. In this application, a WindowBuilder Eclipse plugin [RWC11] additionally helps in easily designing GUI parts as it is an easy-to-use bi-directional Java GUI designer, and the plugin development also happens in the same IDE. Here, the plugin is configured with Java code that sends a request to the data tier through this logic tier, and retrieves the relevant recommendation data from a database using the corresponding API calls. Recommendation data include information that:

1. provides an overview of the currently designed models;
2. provides training material including ample documentation of models [Hon13];
3. displays frequently encountered past questions and their solutions by modellers ;

4. provides specific but useful advise for a model diagram or a model element [ARKS19]; and
5. lists tasks, activities, or processes describing a course of action to model a particular scenario [Fra10] for enhancing a modeller's experience.

The logic tier validates all the recommendation rules using the Java code that is configured within the plugin. In the logic tier, the MagicDraw plugin first identifies the currently open model diagram and its models. Then the plugin automatically connects to the database and checks rules against the model information. Once the information is retrieved from the database, the plugin processes it and beautifies it for display at the presentation tier. As a final step, the components of the UI are populated with this recommendations information.

7.3.3 Database Configuration

The data tier is configured to store and manage data using a MongoDB NoSQL database. This helps the overall infrastructure in allowing unstructured data to be highly available at all times, as it does not depend on the connection of tables, as compared to SQL databases. The high availability of data fosters scalability such that the database grows to large sizes without compromising the request and retrieval processes. This is particularly beneficial when the respective models grow more complex in nature, meaning the data and the corresponding recommendations also grow more complex and larger in size. The data is organised in the MongoDB database using document stores that stores data as documents, in the form of JSON documents. By storing documents in a JSON format, serialisation and deserialisation of these JSON documents into Java objects is done at application runtime in the logic tier. In this way, any language engineer separates the concerns of the actual logic with the data being managed and creates the Java classes based on the defined JSON document formats. The preconfigured MongoDB is referred to as database in the remainder of this thesis. If a database entry needs to be changed then it is easily modifiable and without any significant changes to the database configuration. This database is configured to run locally on a machine containing the same environment for the tool and the DSML, but the overall hosting of the database on an external server is considered more effectively manageable.

Structure.

The database is configured in a way that it consists of different *collections* storing different kinds of information that is relevant to the modelling situation. Every collection consists of a set of fields containing specific values. Each collection also is integrated with a *tags* field consisting of a list of keywords assigned to a specific model or a set of models that are designed using a DSML. These tags are used to identify the models for

which a recommendation is made. This means a high level of consistency is maintained between a DSML and the database, as only the tag values in the relevant database must be updated. As an example, a tag containing a list of values such as "Collimator-V1" and "Collimator-V2" refer to the same DSML element "Collimator" in variants of a DSML.

Table 7.1 lists down the most relevant collections and their respective fields configured in the database. As an example, DSML information that is stored externally in the form

Collection	**Fields**	**Description**
elementsInDiagram	diagramName: String diagramElements: Array	Stores a list of DSML diagram elements that are created as part of a DSML diagram.
faqs	question: String solution: String	Stores frequently asked questions and their possible solutions for a DSML model or model element.
hyperlinks	title: String link: String	Stores internal and external webpage links to training material, and links to videos for a DSML model or model element.
processModels	processName: String data: Base64 String	Stores prescriptive process models and processes as images encoded in base64 format.
recommendations	elementName: String recommendation: String	Stores recommendations for a DSML model or model element.
rules	rule: String recommendation: String	Stores preconfigured rules and their recommendations for DSML models or model elements.
textFromDocs	docName: String docDescription: String	Stores documentation related to a DSML model or model element.

Table 7.1: A list of MongoDB collections and fields we manage, to store training materials, links, recommendations, documentation, and process models for the models and model elements designed with the DSML. Table taken from [GJRR23].

of training documents or handbooks such as in Microsoft Word documents are tagged with the relevant model name or a name that identifies a model from the DSML. To load the information from these Microsoft Word documents, first the necessary content is extracted from the document, a collection *textFromDocs* in the database is created with the fields *docName*, storing the name of the Microsoft Word document, and a *docDe-*

scription, storing the contents of this Microsoft Word document. Another example of a collection is where data related to training videos is stored. These are stored under as links in the *hyperlinks* collection, which consists of a *title* field, that stores the video title, and a *link* field, that stores the hyperlink for the training video. Similarly, the *faqs* collection stores a list of frequently or commonly asked questions and lists down some of the solutions to those questions. The data in the database collections are maintained by domain experts since they suggest and provide changes necessary for the contents of the database, and language engineers are responsible for the functioning of the infrastructure. The information stored in these database collections provide domain-specific training materials, including training videos, general documentation that is extracted from handbooks or guides, process models and their relevant processes for the current models, frequently asked questions, and model-aware recommendations that help users understand and improve their current modelling situation. Building this dataset has been an agile process with information collected across a variety of domains and for specific topics within each of those domains. Each DSML project can, in theory, have a dedicated database that serves the purpose of providing domain-specific information which is not easily searchable and available otherwise.

7.4 Realisation of User-Centric Modelling Recommendations

To assist users in their modelling and provide them with hints and suggestions, the logic tier of the application must be aware of the currently modelled situation of a user. It must then perform a relevant search query to the database and check against a set of configured general rules to return any recommendations that exist in the database. Such a *model awareness* is used to characterise the kind of model currently developed and it also provides the necessary assumptions about the modelling situation for those modelling constructs. The MagicDraw plugin GUI described so far is built in Java Swing and the presentation of the user-centric model-aware recommendations is separated into three distinct parts. The GUI is responsible to observe the current model, analyse the state of the model, and provide recommendations for constructs, but not limited to, such as DSML diagrams, matrices, tables, and model elements that are part of such diagrams. The first part displays a general overview of the currently opened diagrams that is presented in a rather static data which is stored in the database. The second part of the plugin provides dynamically updating recommendations that are either specific to the entire diagram or only certain configured elements of the diagram. The third, and final part, of the GUI displays a set of prescriptive process models that are shown based on the individual models of the DSML diagram. Figure 7.2 shows an example of the UI with the *Overview*, *Recommendations*, and *Process Models* tabs shown to the user.

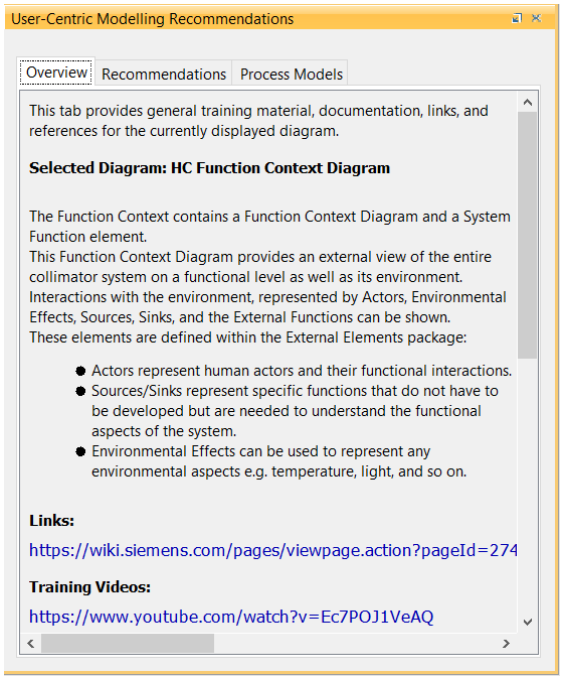


Figure 7.2: An example of the methods and recommendations on a MagicDraw UI with the *Overview*, *Recommendations*, and *Process Models* tabs. This figure shows a general overview for an open model diagram and provides the relevant training material, and hyperlinks that redirect to further documents or videos related to the function model. Figure taken from [GJRR23].

7.4.1 Part I: Overview

As part of the first tab that the user sees on the GUI of the plugin when they open a model diagram in MagicDraw is the *Overview* tab. As indicated by the name, this tab provides a complete overview of the currently opened diagram that is stored in the database in the form of documentation, guides, or handbook materials. The implementation of the *Overview* tab covers customised DSML diagrams, such as a feature model diagram supporting feature models, as well as any UML diagrams, such as a class diagram, sequence diagram, activity diagram, or a state machine diagram. The logic tier is able to process and identify the type of the diagram and sends the relevant query request to the concerned database collection. The request performs a text search on the *tags* field by comparing the values of the *tags* to the type of diagram. To illustrate this with an example, a search query includes the request where the type of diagram is a “Feature Model Diagram”. The corresponding searches are made in the database collection to check if the keywords “Feature Model” and “Feature Model Diagram” are present. If

there is a match, the JSON documents are returned from this database collection. These JSON documents are then processed by the logic tier, beautified into a HTML format, and finally sent to the presentation tier for display to the users. The following sections of the *Overview* tab are populated on the GUI:

General Overview.

This is the first section of the *Overview* tab that displays static model-specific information relevant for the currently opened diagram. The GUI displays a description of the diagram that is stored in the *textFromDocs* database collection as either training material, guide, or content from a relevant DSML handbook. Here, the HTML format helps in styling the data to make the display more presentable with icons, colours, item lists, and so on, so that it invokes positive reactions from the users.

Links.

The second section displays various links to internal organization webpages or even external websites that could potentially detail further information about the currently opened diagram in the form of hyperlinks. The list of hyperlinks are stored in the *hyperlinks* database collection and is retrieved by the logic tier. The hyperlinks are further configured with Java MouseListeners [Jav23] that enables redirection to a browser showing the relevant webpage upon mouse click.

Training Videos.

The third section of the *Overview* tab display various links to videos that contain training material relevant for the currently opened diagram. The logic tier is responsible for extracting information from the *hyperlinks* database collection that are marked as training videos. It is common in the industry to build complex DSMLs with intersecting domain concepts, therefore the creation of training videos to support the DSML concepts helps both novice and expert users. These training videos are mostly created once the DSML projects have been completed and deployed to the practitioners modelling environment, therefore there is a need to refer back to these videos once newer videos or updated videos are released by language engineers and/or domain experts. Such frequent updates to those training videos ensure that users are able to see the most up-to-date videos and that their links do not expire after a few months of DSML deployment.

Frequently Asked Questions.

The final section of the *Overview* tab displays a list of commonly and frequently asked questions by practitioners that are aided with viable solutions for the currently opened

diagram. Language engineers and modelling experts are often asked about specific modelling techniques and methods by practitioners, such as how to build a specific model or the usage of constructs of a DSML. These kinds of questions are collected over years of continuous DSML development for various projects and have been asked to domain experts. Such questions and their accompanying solutions are added into the *faqs* database collection and is shown to the practitioners in this section. This is either updated manually by a domain expert or also automatically queried from a wiki page. An advantage of storing such information in the database is that when solutions to these problems are modified or updated according to the latest modelling information, the necessary updates to endless pages of documentation is not required.

7.4.2 Part II: Recommendations

As part of the second tab that the user sees on the GUI of the plugin when they open a model diagram in MagicDraw is the *Recommendations* tab. In this tab, recommendations for the currently opened diagram that are based on certain configured general rules are displayed. While the recommendations are stored in the database and continuously updated, they are often displayed in a dynamic way on the GUI, as the current state of the models in the model diagram are continuously changing based on the modelling situation of users. The recommendations displayed in this tab are stored and managed in the database in the *recommendations* collection, while the *rules* collection stores the preconfigured rules based on which a specific recommendation is provided to the user. The recommendations provided as part of the application for the currently opened diagram are either for the individual model elements that are visible on the diagram, or a combination of multiple model elements that are currently part of the diagram. Similar to the *Overview* tab, the logic tier is able to identify the type of the diagram and performs the respective text query search on the *tags* fields for the collections identified by the preconfigured general rules. The JSON documents containing the recommendations, that are retrieved from the database, are also processed by the logic tier and beautified into an HTML format for finally displaying it to the users on the GUI.

Preconfigured General Rules.

The recommendations that are displayed as part of the *Recommendations* tab are provided by checking against a set of preconfigured general rules that are present in the database collections and validated by the logic tier. As part of the infrastructure, a simple rules engine [Fow10] is developed that is then eventually bundled together in the final DSML plugin archive file. This simple rules engine is developed within the MagicDraw plugin and the rules are eventually stored and managed in the *rules* database collection. A rule typically evaluates a particular condition, such as by using a logical or a relational operator. Other rules analyse the model diagram and provide recommendations

based on the state of the model elements. In the remainder of this section, each rule is discussed supported with a rationale that provides the necessary reasoning as to why this rule is required, and in addition, discussing the benefits of the individual rules. In the tool, each rule is then checked against the currently opened diagram, such that the plugin is able to identify the kind of models being currently used within the respective model diagram. In the following, the general rules for any kind of DSML diagram is discussed.

Rule 1 (R1): Provide specific recommendations for model elements that are listed as part of the legend items but missing as part of the model elements that are displayed on the currently opened diagram.

Rationale: Often, language engineers design model diagrams (UML or domain-specific) that are accompanied with legend items that list a set of model elements used to describe different styles and also logically group symbols within a diagram. Legend items are model elements used to define different styles and visually group symbols in a diagram for better model element visualisations. The reasoning to include such a rule is that as legend items often make for a good indicator of the visible and missing elements in a model diagram, it is beneficial in identifying which parts of the model diagram, specifically the model elements, are missing.

Rule 2 (R2): Provide general recommendations for model elements that are listed as part of the diagram elements toolbar but missing as part of model elements that are displayed on the currently opened diagram.

Rationale: The diagram elements toolbar in MagicDraw is configurable to an extent that it lists down the model elements that are created as part of the currently opened model diagram. They assist users in easily designing model elements on a model diagram. Often, users forget to utilise the full set of DSML functionalities that are offered in the modelling environment, and choose to model only a few elements based on their current modelling know-how. This rule encourages users to further verify if the models from the DSML that they are currently using are sufficient for their modelling situation or they need additional functionalities to better represent their modelling needs.

Rule 3 (R3): Provide recommendations for model elements that are currently visible on the currently opened diagram.

Rationale: Each domain consists of a wide variety of information based on feedback from users, domain experts, and modellers over the kind of specific values that a model

element must consist of and any specific properties that must be configured for a model element. Collecting, storing, and managing a dataset of such recommendations for such model elements is important in addressing issues related to guidance and suggesting users a way in their modelling. As users employ newer models and update their existing models, the recommendations changes accordingly, and is therefore dependent on the context of the system.

Rule 4 (R4): Provide specific recommendations for a model element or a linked model element that is based on logical conditions and relational operators over potentially several model elements in the currently opened diagram.

Rationale: This rule ensures recommendations are provided for a single model element on a diagram that are based on relational operators and logical conditions. Relational conditions such as <, >, <=, >=, ==, and != are checked against the value of the model elements. An example of a simple relational condition is that if a model element is configured to be of type energy, and the value is set to electrical, then a specific recommendation such as suggesting a recommended energy value in kilowatt-hours (kWh) is provided. In addition to relational operators, logical operations are also performed on a combination of model elements on the diagram. Logical conditions such as AND, OR, and NOT are therefore checked against a combination of model elements. Here, an example of a logical condition could be to check if a data sink exist in the diagram, but not a data source, meaning a specific recommendation related to the source-sink combination is provided.

Model-Aware Recommendations.

The *Recommendations* tab of the GUI shows recommendations and suggestions for guiding users in their modelling. These recommendations are a result of retrieving the corresponding data from the relevant database collections and are checked against the preconfigured general rules. The active and dynamically changing recommendations are a result of the analysis of the current models part of the active DSML project and is populated into two sections of this tab.

In the first section, recommendations that are specific to the currently opened DSML diagram are displayed to users. This is a result of the logic tier first analysing the currently open diagram, checking the type of the diagram, then checking the rules that are configured for this diagram, and finally retrieving a list of recommendations from the configured database collections. This means that any changes that the plugin detects during an active modelling situation are also analysed and considered and the recommendations are therefore accordingly adjusted and displayed to the users. Essentially, the application is therefore aware of the model constructs and provides the necessary recommendations that are based on the modelling stage that the user is currently in.

A primary advantage of providing recommendations to only the visible parts of the diagram is that the recommendations targets the user-centric view of a certain aspect in a larger system. For viewing recommendations for other parts of the system, a user would navigate to the respective model diagrams. Constructs that may never be used may also be recommended, for providing a more complete view of the current models in the system. A list of non-exhaustive recommendations that are specific to the currently open DSML diagram are listed as follows:

- The tab displays the missing parts of the diagram that includes model elements and other non-functional aspects such as legend items that describe the model elements.
- The tab suggests users to configure a certain model element, or a combination of multiple model elements, in the model diagram. For example, a function model must contain at least one function.
- The tab specifies which constructs of the model diagram or the corresponding models must be either modified or adapted. For example, a function's name in a function model must be unique.
- The tab also specifies and suggest any layout changes that are necessary for certain elements that are part of the diagram. For example, typically input ports must be positioned on the left side of a function.

The second section of the *Recommendations* tab displays recommendations that are specific to the model elements and are visible on the DSML diagram. The application analyses the model elements that have already been configured as part of the model diagram and provides dynamically changing recommendations that are based on the model element's properties or values. A non-exhaustive list of recommendations that are provided to users for the individual model elements configured in the DSML diagram is as follows:

- The tab asks users to configure or set particular properties for a given model element. For example, a function name must be meaningful and not simply "function1".
- The tab recommends users to check the type or kind of a model element and also suggests any additional information that maybe related to the respective type of model element. For example, a function must be type checked in a function model.
- The tab display recommendations for setting specific values for a certain model element that are based on historical feedback from other users or domain experts. For example, a function for calculating the energy of an electrical device must recommend a certain value in kWh.

- The recommendations also include asking users to check the *Overview* tab of the GUI to access training material, handbook information, or general guides that are specific to the model element.

7.4.3 Part III: Process Models

The *Process Models* is the third and final tab of the GUI that the user is shown as part of the MagicDraw plugin on opening of a DSML diagram. In this tab, a list of prescriptive process models in the form of activity diagrams are displayed. These process models are relevant to the currently opened diagram and describe the necessary processes, tasks, and activities that may be performed by the DSML user on the diagram and its constructs to progress in the desired modelling for this particular DSML diagram. The process models also details and inform users the current state of the modelling for these constructs of the diagram, as the logic tier checks the models of the currently opened diagram against any preconfigured rules. The *processModels* database collection stores and manages process models and the various processes as images, in a base64 encoded format [Jos06] which corresponds to the DSML diagram and is queried against the relevant tags. Here, a base64 encoding for the images is chosen, as the information stored in the database is mostly in text format and the size of the images are generally less than 10 MB. As part of displaying the process models on the GUI, the logic tier performs a search based on the type of diagram to the tags in the *processModels* database collection and retrieves a list of processes and process models. Here, the data is stored in the insertion order, therefore is also retrieved by default in the same order. The logic tier then transforms the information into a series of graphics using Java 2D™ API by decoding the base64 data. It then paints the graphics into specific colours according to any rules that are configured against this diagram, and prints a sequence of created 2D graphics as process models for display on the tab. This serves a primary benefit in the application as the logic tier adjusts the graphics according to the analysis of the current diagram and adjusts the process models if there are any subsequent changes based on preconfigured rules. These changes result in beautifying the process models by applying styles such as colours and icons. The styling options also help users identify easily which processes, tasks, or activities of the process models are complete, and marks the remainder as incomplete, thus improving the overall UX of the modelling situation.

7.5 Industrial Example

Motivation.

In this section, we take a look at the applicability of the infrastructure presented so far in this chapter in a real world context. Siemens Healthineers have continuously adopted MBSE techniques for modelling various modelling aspects related to medical devices,

requirements definition, product line engineering, and product structure modelling that involves bill of materials. They have used methods and tools to support their modellers in developing complex models. Further, some of the MBSE concepts have also been applied to DSMLs that are applicable in defining functional, logical, and technical views, for example, in the SPES [BBK+21] methodology. This is done to achieve truly modular separation and reuse of model elements based on various modelling artefacts. Here, an example of a model used to describe the aspects of function modelling in a system is discussed in more detail.

Problem Statement.

The example discussed in this section is based on the following medical case. In this case, a patient is brought into a medical clinic with a suspected leg fracture. Once the patient is admitted to the clinic, the treating clinician suggests a 2D X-ray of the patient's leg to diagnose the severity of the fracture. The clinician forwards the patient to the radiology department of the clinic or another hospital that has X-ray diagnostic infrastructure. As soon as the patient arrives in the radiology department, the patient is taken to the respective X-ray room by a medical technical assistant (MTA). Before the start of the diagnosis, the MTA has prepared the X-ray system and added the patient details already into their X-ray diagnosis database system. To successfully complete the diagnosis, the MTA positions the leg of the patient on the X-ray apparatus and starts the X-ray procedure. Finally, the 2D digital radiograph of the patient's leg is generated and stored in the system for further diagnosis.

Model.

To model an X-ray system, many physical elements in a technical view and their interfaces need to be set up. A detailed description of the Siemens Healthineers DSML used for modelling these elements is described later in Section 8.1. For now, let us just consider that a DSML supports modelling of elements belonging to a function model domain such as functional context, functions, actors, and other functional parts described earlier in Section 2.3.2. A very specific element of the X-ray system is the collimator, which is responsible for collimating the radiation beam from the X-ray machine onto the patient's leg. In a modelling sense, this part is further decomposed into a set of functions that needs to be performed so that the overall collimator works effectively.

Figure 7.3 shows a part of the example of a collimator described as a function context model, in the functional view, that has been modelled in MagicDraw. This figure shows a model browser (**IA2**) that DSML users use to model their system aspects based on the individual building blocks of the DSML.

In Figure 7.4, the left part of the figure shows the toolbar options that are used to model various functional aspects, such as setting up external connections between the

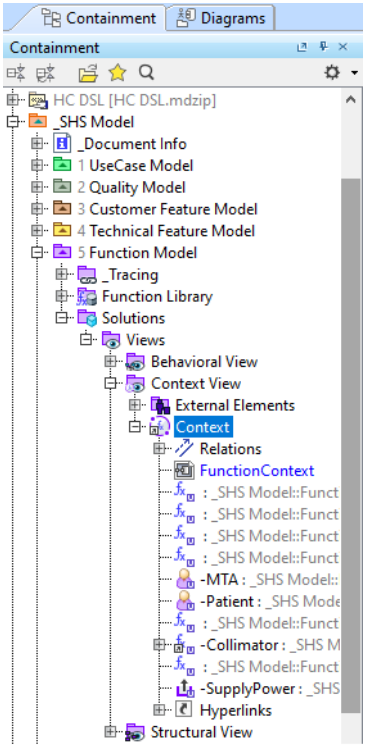
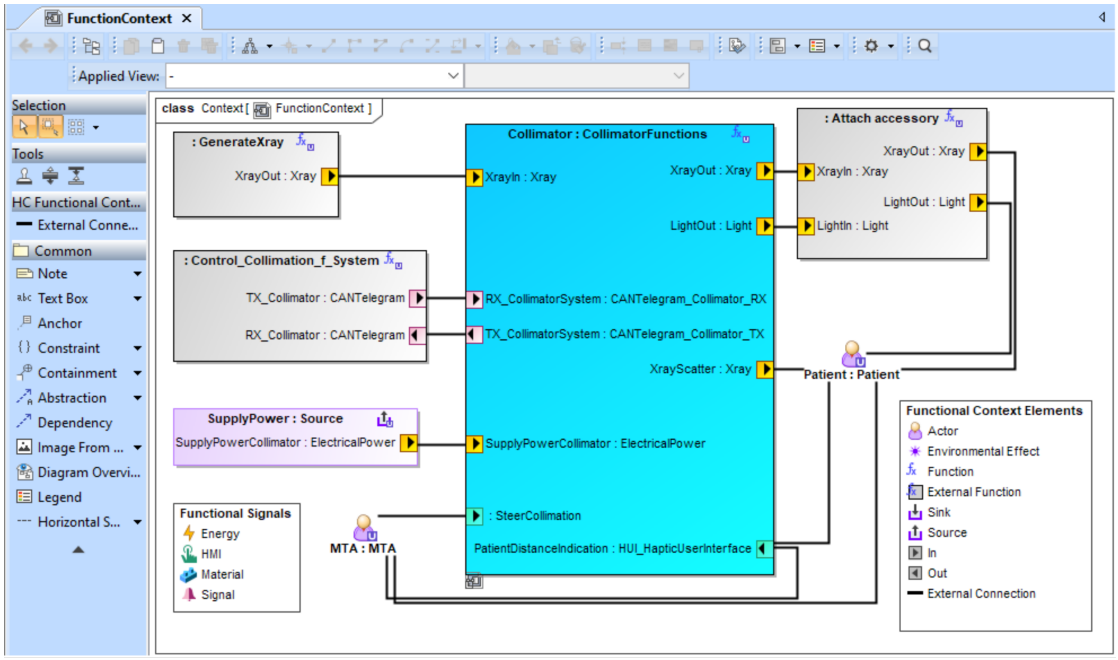


Figure 7.3: A part of the model example of an X-ray collimator function context diagram designed using the Siemens Healthineers DSML that shows the model browser (**IA2**) tree on the left consisting of the DSML constructs including the various building blocks of the DSML. Figure adapted from [GJRR23].

physical elements. The right part of the figure contains the actual function context model that provides the external functional view of the entire collimator, along with the various interactions such as actors, environmental effects, and other external functions that must be modelled to fully understand the collimator system. The model diagram describes the various decomposed parts of the collimator. The central part of the function context model shows the primary function, i.e., the collimator itself, that receives (Rx) inputs and transmits (Tx) certain outputs from smaller subsystems such as the collimator controller, or a power supply source, or additional physical apparatus that may be connected to the patient's leg. Actors, namely the MTA and the patient, are also modelled as part of the diagram as they perform certain interactions with the collimator. The input and output ports are configured with the respective functional signals listed in the legend items on the bottom left, with the colours of the ports signifying the type of functional signals. These legend items represent various input and outputs of the functional elements such as green for a human-machine interface, yellow for energy, blue for materials, and so on.



(1) Toolbar *(2) Function Context Model*

Figure 7.4: A model example of an X-ray collimator function context diagram designed using the Siemens Healthineers DSML that shows (1) the diagram toolbar in the left with configurable model elements such as external connections, and (2) the main function context model on the right that consists of the various functions, actors, sources, signals, their connections, and other functional DSML constructs. Figure adapted from [GJRR23].

Methods and Recommendations.

In order to build the collimator, various functions must be set up by the user by performing a sequence of steps. It is easier for a user equipped with knowledge of modelling X-ray system to model the parts of a collimator. However, novice modellers need initial guidance in starting their collimator modelling journey. Similarly, advanced modellers require the use of advanced functionalities for encountering more complex scenarios later during modelling. Such modellers often spend huge amounts of time and resources to endlessly search through passive information sources such as training materials, books, handbooks, or tutorials which are often overly generic. Therefore, the guidance infrastructure defined in this chapter provides users of all kinds with methods, concepts, and recommendations directly on the MagicDraw DSML itself which alleviates some collimator modelling concerns and encourages them to think in a specific direction or in a novel way.

The GUI consist of the *Overview*, the *Recommendations*, and the *Process Models* tabs. As described previously, Figure 7.2 displayed a general overview of the currently opened function context diagram. Information that is specific to this diagram is shown in this tab, but also general overview pertaining to the entire system are also shown here. A corresponding dataset consisting of domain-specific recommendations such as for medical devices like X-rays, or more specifically a collimator, are used that provides the necessary overview to individual modellers and their modelling needs. In the first part of the *Overview* tab, the logic tier analyses the type of the diagram, and retrieves a static documentation from the database that is related to this function context. Next, any additional hyperlinks are displayed as navigable links that ultimately redirects a user to a more relevant training material that could potentially describe in detail information about the function context, or just general interactions with the system interfaces. Here, links to training videos or videos that generally outline a method of modelling or the system under consideration, such as the collimator and its functions, are also listed so that the users easily navigate to the respective information.

Figure 7.5 shows information as part of the *Overview* tab and consists of a list of frequently asked questions and their possible solutions that may be of help to modellers. These are commonly asked questions to language engineers and domain-experts by modellers and practitioners who often use the DSMLs and also consist of domain-specific questions such as "*What is a Collimator?*", or how to specifically model a function in a specific scenario. The information listed here is retrieved from the database that contains the list of question with respect to the function model, collimator, or a sequence of methodical steps that users follow during a function context modelling. The figure shows that the displayed information is model-aware, meaning for the collimator example, a specific question related to a collimator is displayed.

The second tab of the GUI is the *Recommendations* tab and is shown in Figure 7.6. Here, examples of dynamic and model-aware recommendations are displayed for the provided function context diagram, that is based on the current state of the models, their properties, and the analysis of the various linked parts of the models. The tab is split into two sections, the first section provides recommendations that are specific to the collimator function context diagram, while the second section provides recommendations for the specific model elements of the collimator function context as described earlier in this chapter. The first recommendation in the figure lists model elements that are currently not part of the visible function context diagram. These elements are the functional signal material, the sink, and an environmental effect. Here, for this recommendation the rule **R1** is followed, where the logic tier checks the currently listed model items in the diagram and compares it to the legend items, the functional context elements, and the functional signals. Providing this recommendation is beneficial in informing users that perhaps a part of the function context is not yet considered, for example, a sink that stores the final 2D digital radiograph. The rule **R2** is checked for any missing elements that are present as part of the diagram toolbar, but is not listed, as the external connection element is

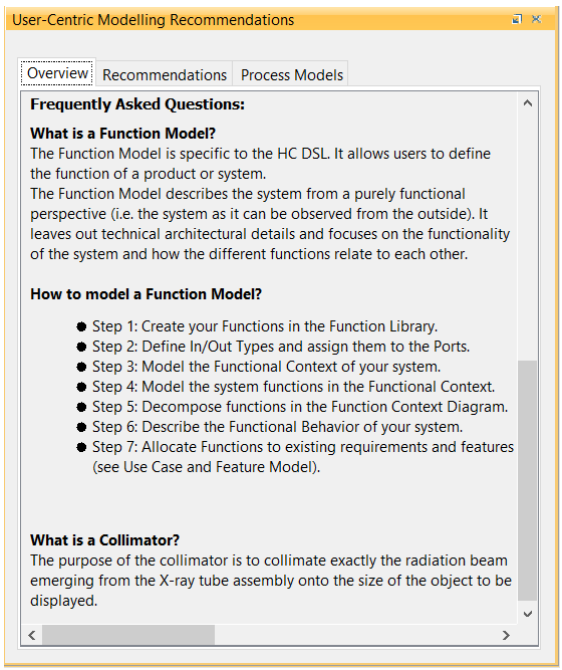


Figure 7.5: An example of providing frequently asked questions in the *Overview* tab for the collimator function context diagram. These questions are stored in the database, and have been collected over the years as the most asked questions to either language engineers, or to domain-experts by the modellers. These questions are associated with the use of the DSML constructs, on how to model specific scenarios with the DSML, or general domain-specific topics. Figure taken from [GJRR23].

already part of the function context model. The following recommendation listed in the figure is based on the analysis of the function context diagram and is checked against the logical rule **R4**. Here, the rule condition stored in the database as "`Source NOT Sink`" is checked against, which validates if a source exists in the diagram, but not a sink. This rule is configured in the database as part of historical models that generally configure both a source and a sink for a functional context. The second section of *Recommendations* tab checks the rule **R3** and lists recommendations that are specific to the visible model elements in the function context diagram. The third recommendation of the *Recommendations* tab displays the information that although a power supply source configured in the diagram and is set to an electrical energy, an accompanying measurement value for this energy source is not yet assigned. Based on historical data for power supply configurations in similar models, a recommended measurement value is also indicated. Such a measurement value is either be populated by a domain expert or is

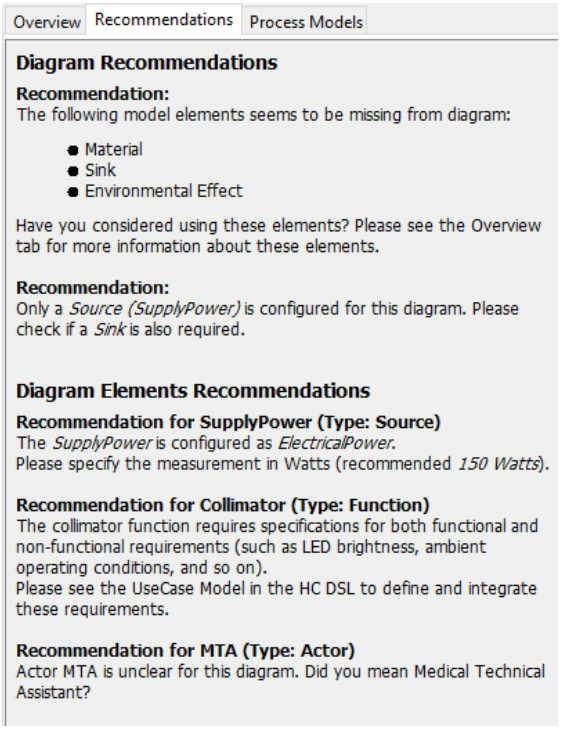


Figure 7.6: The *Recommendations* tab of the UI showing sample recommendations for the collimator function context diagram. The tab provides recommendations for the (1) collimator function context diagram such as which elements, or combination of elements, are missing from the diagram, and (2) specific model elements such as actors, functions, sources, and other DSML constructs. Figure taken from [GJRR23].

extracted from an automatic analysis algorithm [KSK+19]. As soon as the measurement value for this electrical power supply source is initialised, the recommendation would be automatically removed from the list of recommendations on this tab. Finally, the fourth and fifth recommendations are displayed by checking against rules **R4** and are based on the relational conditions. For example, the fifth recommendation is stored as a **R4** rule with the rule condition "`Actor == MTA`".

To illustrate the model-awareness of the proposed recommendation system, the functional context model example is enriched by adding a sink, by setting the measurement value of the supply power to 150 watts, and by renaming the actor MTA as Medical Assistant as shown in Figure 7.7. The patient data sink consists of the data inputs with type information from the data outputs provided by the central collimator.

Figure 7.8 shows the updated recommendations after adding a missing element (sink),

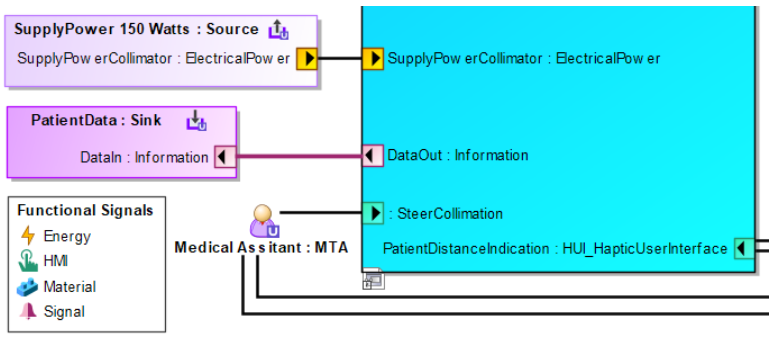


Figure 7.7: The function context diagram is enriched with a sink and a measurement value for the power supply.

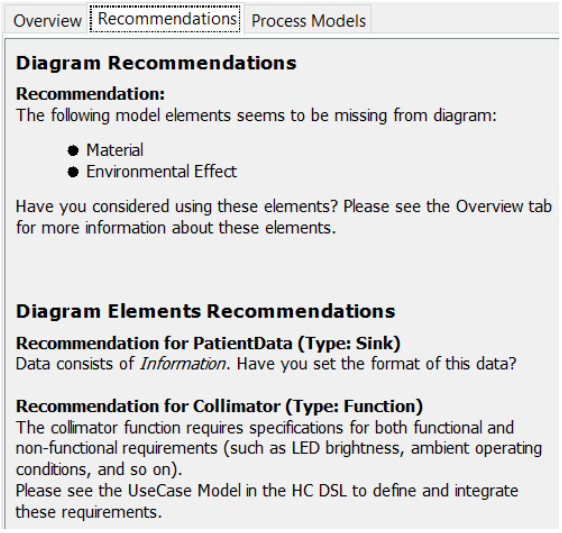


Figure 7.8: The *Recommendations* tab showing the updated recommendations for the enriched collimator function context diagram.

updating the power supply measurement value, and by renaming the actor MTA. The recommendation system therefore removed the sink from the list of missing elements (**R1**), and added a diagram element specific recommendation by checking rule **R3**. This rule identified the patient data as a sink with an information type as an input. A recommendation has been suggested here to the user for checking if the data formats have been set correctly. This shows during modelling how the recommendations are able to change based on the models, thereby demonstrating its model-awareness.

Figure 7.9 displays the *Process Models* tab in the GUI that shows the various process models attached to this function context diagram. The figure details two process models

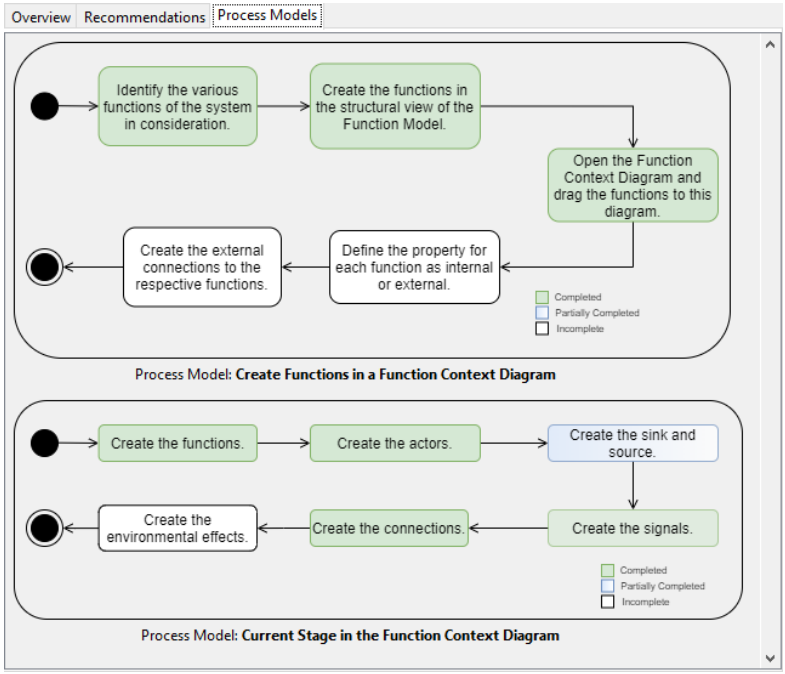


Figure 7.9: The *Process Models* tab of the UI shows process models that are generated and displayed for the various processes, tasks, or activities for the collimator function context diagram. The process models: (top) details processes needed to create functions using the DSML, and (bottom) displays the current state of the collimator and which processes are completed, partially complete, or incomplete. Figure taken from [GJRR23].

that are generated from a list of processes retrieved from a query to the database. In the first process model, a series of methodical steps is described that is needed to create functions in a function context model diagram. Here, some of the information may overlap with the information shown in the *Overview* tab, but the processes are actively checked against the currently opened diagram, any rules attached for this model diagram, and are finally beautified with styling options that differentiate the various processes and its current state. As an example, the solid green processes indicate that the process is complete, the fading blue process indicates that the process is ongoing, while the white processes indicate that no efforts have been made for fulfilling this process. Here, Java's 2D graphics is employed to generate the process models, as well as integrated with legend items that identifies the various processes, and are preconfigured with the legend items "Completed", "Partially Completed", and "Incomplete". In the second process model, the current state of the function context diagram is displayed. This means all the model elements in the function context are identified and analysed against any preconfigured

rules. An appropriate rule **R4** is checked against, for example, if two actors are created in the model diagram, then the process "Create the actors." is considered complete. As we only have a source configured for the example model, the corresponding process in the process model is marked as "Partially Completed", which encourages users to double-check their models. Since an environmental effect is missing from the collimator functions example, this is displayed in the corresponding process as "Incomplete".

7.6 Discussion

The language guidance infrastructure and its concepts presented in this chapter allows language engineers to think and implement DSMLs in a way that it provides effective guidance mechanisms for model-specific methods, techniques, and recommendations. The evaluation of the language guidance infrastructure was carried out by practitioners and researchers in software and systems modelling who reasoned that an integrated DSML infrastructure must provide dynamic model-aware data for the ever growing models and that a guidance infrastructure must be active and synchronised with the ongoing modelling work. It is therefore important for stakeholders of a domain to understand which specific processes, activities, or tasks are viable and necessary to be performed for a more holistic modelling experience. Earlier, such guides for users were provided to users in the form of lengthy documents, handbooks, tutorials, and user manuals that are not only cumbersome to read through, but also lacks in providing up-to-date knowledge for the domain in consideration. It is often that users have to tediously search through endless pages of such static documents, therefore reducing the overall productivity of users, and not being able to fully utilise the functionalities offered by the DSML and the corresponding modelling tool that it has been built on. Further, such documents do not often detail information that is based on the current modelling situation of the user. By providing recommendations based on predefined general rules, language engineers more easily integrate methods within the DSML itself. This chapter shows the example of such a guidance infrastructure implemented in the MagicDraw modelling tool as an additional GUI that provides additional customisations possibilities, and provides model-specific training material, suggestions, and processes that are tailored to the current modelling situation.

So far the literature does not specify a technique that integrates such a methodology described in this chapter within an industrial DSML combined with an appropriate modelling tool. Reusability aspects are generally not considered since project, time, and resource constraints are often limited and integrating such methods is often does not have more priority than defining a good DSML. It is imperative that data is collected from domain experts and practitioners to integrate within a DSML for continuous support. The example of Siemens Healthineers is not the only project that use such a guidance infrastructure. Because each business unit solves different domain challenges, even in the

medical industry, recommendations that are specific to the domain must be provided to modellers. While context conditions are used to validate the well-formedness of a model, it would essentially not tell us much about the semantics in terms of the meaning and behaviour of the models. While context conditions, in the form of validation rules, in the DSMLs generate warnings or errors, they provide a stricter constraint to the syntax. Methods and recommendations provide a more complete and positive outlook that considers the current state of the models and gives users more confidence to effectively model using a DSML.

To solve the above challenges, a three tiered architecture consisting of a presentation tier (GUI), a logic tier (logic and rules), and a data tier (NoSQL database) is developed. The GUI provides information to users regarding a general overview of the currently open DSML diagram, hyperlinks for redirection to different sources of domain-specific information including webpages and videos, frequently asked questions in that domain, active recommendations for a DSML diagram and its elements, and providing a set of process models that demonstrate the various processes performed or missing from the diagram. In each domain, a corresponding dataset is used that consists of a set of configurable general rules providing active domain-specific information that is based on the current state and context of the models of the users. While there is an effort to build such an infrastructure, its cost is neglected when compared to the tool MagicDraw itself. In all, a Java developer familiar with the MagicDraw Open API can build this infrastructure within a month, and the setup can be easily reproduced. The rules and their recommendations are adjusted frequently to include more complex modelling scenarios, which means that the DSML itself does not require constant updates. An ongoing work to provide actions directly within the infrastructure is currently underway, for example, assigning measurement values to a model, or defining complex rules that validates a link between different model elements to allow navigability between such recommendations. Further work on making processes within a process model navigable is also underway, while there are also efforts to provide modelling predictions with machine learning algorithms or with natural language processing techniques.

To identify concrete recommendations, efforts were first made to understand user needs and define a concrete example. This included focus groups, modelling community discussions, interviews, and surveys in different DSML projects within Siemens. A scaled down example of the Siemens Healthineers was chosen as the example to identify such concrete recommendations in the healthcare domain. This example is also evaluated later in detail in Section 7.5. Different models within this example project were evaluated by understanding how practitioners develop such models over a period of time. The focus was to understand how a combination of models work in a project, how the relations between such models are designed, and what kinds of support methods could have been provided to ease the design process. Then, different kinds of rules were identified and a data set was created for hosting the different rules and recommendations related to these rules. Language engineers then developed a GUI that is able to display these

recommendations directly on the modelling tool, MagicDraw. Further, existing handbooks, training materials, and wiki pages were sourced through to enhance the data set. Finally, workflows and activities regarding the models in development were identified and added to this expandable data set.

The validity of the concepts presented in this chapter is the extent to which the overall mechanism is free from systematic errors or bias. MagicDraw, Rational Rhapsody, or Enterprise Architect do not represent all available tools for building DSMLs. Describing the study using MagicDraw introduces a vendor-locked scenario. Other solutions include providing recommendations directly within the context of the system, however it often leads to overload of information shown to users when the models are very complex and such recommendations would restrict their modelling. While context conditions detect and report errors directly in the system context, these are more critical than just providing methods and guidance to users for visible parts of a DSML diagram. The tiered architecture enables separation of concerns, meaning the data tier is completely developed independent of a modelling tool. The reliance on external data sources provides a more general and tool-agnostic solution for the recommendation infrastructure. Updating such data sources externally is conceptually described in a straightforward and intuitive way for non-experts, while changes to DSML is often a time-consuming process for language engineers. Here, recommendations are based on historical data but users are also encouraged to think in a novel way by providing a general overview of the diagram and its accompanying system. A caveat for implementing this methodology is that the modelling tool must expose APIs that retrieves model information including its configured properties. While the data can be rather limited, domain experts can work with language engineers to populate the databases more effectively. It must also be noted that language engineers may often lack software development skills like programming in Java, and there is a natural inclination to focus on the syntax of a DSML.

An example of a real-world industrial function context is used. The example describes the functions of a collimator in an X-ray system modelled by experts at Siemens Healthineers. The example provides ample model information for providing general training overview, suggestions to specific parts of a collimator, recommendations to the function context model, and also an overview of the processes involved for modelling an X-ray collimator and its missing parts. It was observed that for the X-ray collimator function context example, around 12 rules and their recommendations were configured. Other model diagrams, such as a structural view of the function model also contain a similar number of rules. As the recommendations are only calculated only for the currently open diagrams, scalability issues were not observed. This also means that users are not burdened with a plethora of recommendations. Further, language engineers also only configure a simple rules engine. Defining process models for a model diagram is rather intuitive for language engineers as they identify the different parts of a DSML that are modelled during design time. The study is based on the assumption that, as of writing this thesis, there is no existence of another reference implementation that provides ef-

fective modelling methods and recommendations. The presented infrastructure uplifts the modelling experience of users and results in a more positive approach to modelling. Therefore, there is a need to tightly integrate aspects for a more complete DSML infrastructure that is closely dependent on the current modelling situation of users. This is a result that both novice and advanced users better understand the modelling constructs and eventually confidently model their domain aspects. Therefore this chapter describes a good starting point towards improving the state of industrial DSMLs and is more active, aware, and in sync with the modelling situation of users.

7.7 Related Work

In order to successfully implement this reference guidance infrastructure, a deeper knowledge of domain-specific aspects must be considered in relation to the DSMLs and their constructs. Various environments supporting computer-aided method engineering methods [GLB+86, NR08] are present in the literature, but they are mostly concentrated on providing generic methods and little consideration to domain-specific aspects are made [Hon13]. MBSE methods such as MagicGrid exists [AM18] but the kind of methods it provides is overly generic and challenging to implement in the form of an independent guidance infrastructure. More recently, commercial content providers have designed recommender systems that not only gathers and understands the preference of users, but also provide dynamic recommendations [NFT+10, NKBK12]. In the methodology described in this chapter, there is no storage of preferences of users, but rather historical data related to a particular domain is stored and managed in an external database, from which recommendations are generated for the current models of users. Algorithms such as nearest-neighbour approach [Sip16] or mutually reinforcing methods [WXU16] are used to classify recommendations, but their use is rather extensive, and the implementation of the rules described in this chapter is using a simple rule engine in Java [Fow10] that is directly integrated in the modelling tool. Service oriented architecture for linking MBSE tools to services has also been studied [ABN+08] as they offer a composition of smaller services. Research on context-aware methods have been discussed [HGMB13, LEN+14], but they are not primarily used in modelling domain-specific constructs, and do not either provide and methods or recommendations to users. Process models that are process-aware along with their verification exist [KKR+22, CCG+08], but it does not consider the current understanding of the models and does not provide a more holistic view of the current modelling scenario. The database configurations listed in this chapter are designed to be specific to each DSML project, and therefore in comparison to larger domain-specific textual databases [SKOK17], only a simple document retrieval configuration is adequate.

Chapter 8

Case Studies

This chapter presents three distinct case studies in a real industrial context. Earlier chapters described a systematic engineering of industrial DSMLs by separating the concerns of defining interoperable language components, looking at ways to improve the UX and the usability of DSMLs, as well as providing guidance and methods to modellers for successfully deploying, evolving, and using such DSMLs. While individual chapters presented their own examples and case studies in industrial contexts, this chapter presents a combined perspective for end-to-end DSML solutions in the healthcare industry (Section 8.1), digital industry (Section 8.2), and an application in a public funded project, SpesML (Section 8.3). Table 8.1 shows the listing of the case studies that use the different results from the individual chapters in this thesis. Chapter 5 described an exchange mechanism between modelling tools and the application of its concepts was described using a rather small DSML involving use cases, actors, tasks, and their relationship. Thus, those concepts are not specifically referenced in the case studies described in this chapter. The case studies detailed in this chapter use the results from the chapters mentioned in the table and described in the MagicDraw ecosystem.

Number	**Case Study**	**Chapters Referenced**
1	Siemens Healthineers (SHS)	Chapter 3, Chapter 4, Chapter 6 Chapter 7
2	Siemens Digital Industries (DI)	Chapter 3, Chapter 4, Chapter 6 Chapter 7
3	SpesML	Chapter 3, Chapter 4, Chapter 6

Table 8.1: Table listing the case studies that use the results from the respective chapters.

8.1 Case Study 1: Siemens Healthineers (SHS)

8.1.1 Motivation

The healthcare division of Siemens AG is known as Siemens Healthineers (SHS) and is at the time of writing this thesis owned 75% by Siemens. X-ray products, magnetic resonance imaging (MRI) devices, computed tomography (CT) machines, molecular imaging, and molecular diagostic solutions are some of the products that SHS offers. The technology department at SHS has strived for moving towards a digital transformation of their products while ensuring better methods and techniques for SHS products and their lifecycles. As a result, domain experts and modellers in SHS have continuously embraced MBSE methods that achieve modelling of various medical devices, defining requirements for their products, achieving efficient SPLE, and ensuring consistency across various parts within their complex systems. Some of these MBSE concepts have also been applied to DSMLs that define functional, logical, and technical views, for example, in the SPES [BBK+21, GJRR22b] methodology. Previous chapters in this thesis therefore proposed systematic engineering approaches to encourage SHS domain experts and modellers in using DSMLs that are modularly built and support reuse of language and their artefacts.

8.1.2 SHS DSML, its Building Blocks, and the Models

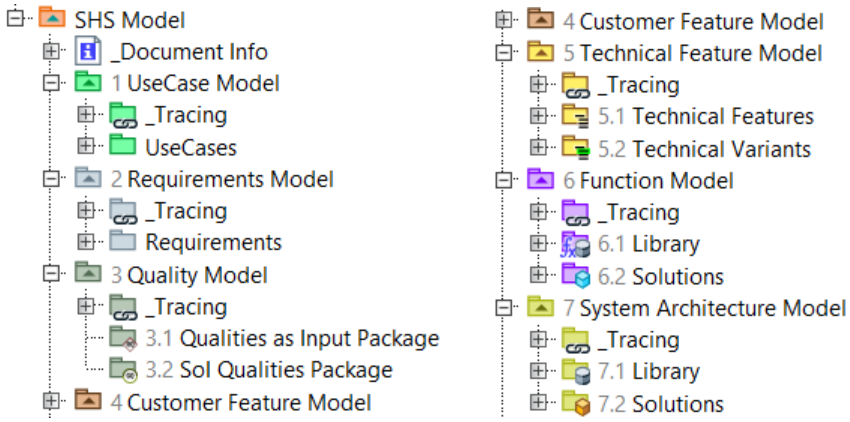


Figure 8.1: The models of an SHS DSML that are created automatically when a new project is instantiated in MagicDraw.

To realise the modelling journey for SHS, a set of DSML building blocks were created or reused (partially or fully). Each of these DSML building blocks is aimed at modelling a certain aspect of the the software and system architecture for a particular domain

(Def. 5). Figure 8.1 details the models that are created automatically for representing individual aspects of the healthcare domain configured by language engineers as DSML building blocks. The collection of all the six main DSML building blocks is collectively referred to as the SHS DSML. In the following, we examine these DSML building blocks, the various elements that are modelled with them, their visual representations, and their applicability in the healthcare domain.

UseCase Model. This DSML building block allows modellers to describe the intended functionality of a system. In particular, this is achieved by modelling use cases, the various actors (human or system), the involved activities, their interactions, and associations as shown in Figure 8.2. Additionally, modellers also create use case diagrams to visualise the created models and their relationships on a custom MagicDraw diagram (Section 6.3.1).

Figure 8.2: (Left) Model elements and (right) custom views that are created using the use case DSML building block.

An example of a model using the use case DSML building block is shown as part of a use case diagram in Figure 8.3. Here, two human actors (*MedicalAssistant*, *Patient*), one system actor (*IT*), are associated to the *Diagnose Patient* use case. The *Medical System* is modelled as the system of interest, therefore all the use cases must exist within this system and are listed under the UseCase Table that is shown in Figure 8.4.

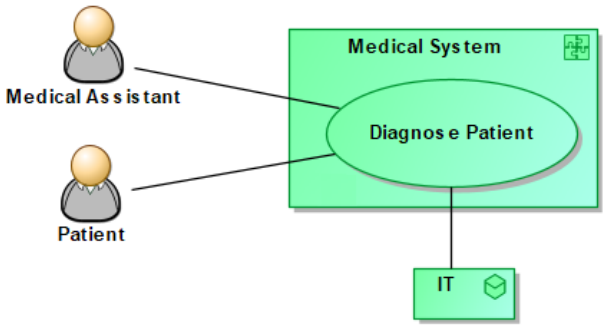


Figure 8.3: A model involving two human actors, a system actor, and a use case in a system of interest.

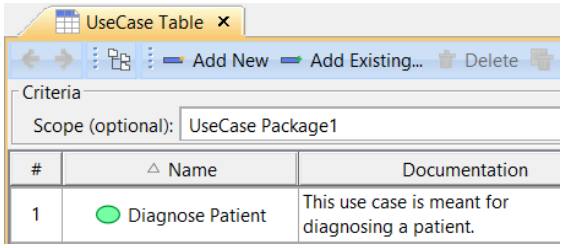


Figure 8.4: A table listing the use cases in a system of interest, the *Medical System*.

Requirements Model. This DSML building block allows modellers to define requirements for their modelling projects. In this building block, both the functional and the non-functional requirements are modelled and these requirements are associated to model elements in other DSMLs building blocks (Figure 8.5). For example, if a requirement is associated to a use case, then it is visually displayed using a custom matrix. The requirements model was briefly discussed in Section 3.5.

Figure 8.5: (Left) Model elements and (right) custom views that are created using the requirements DSML building block.

Figure 8.6 shows an example of data related to the requirements displayed in the requirement table. The figure shows both the functional and non-functional requirements as part of the same table, so that modellers have a quick overview of the different requirements. In addition, properties such as documentation, identifiers, and status of the requirement are also set directly on this table.

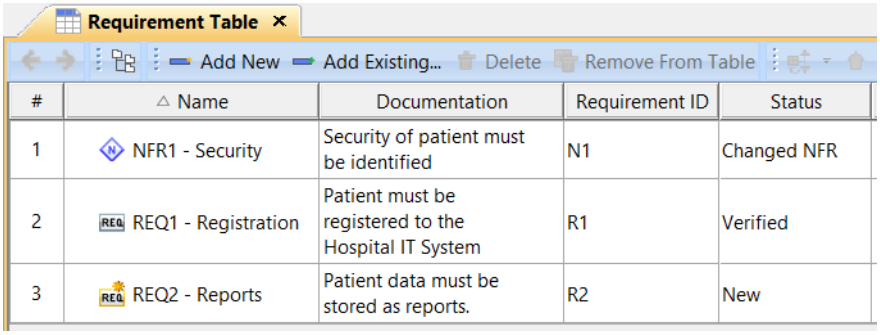


Figure 8.6: An example of the requirement table listing the various functional and non-functional requirements.

Figure 8.7 shows an example of the relations of the various requirements to the use cases we saw earlier in the use case model. The relation, *RelatedToUseCase*, has been configured for the *Diagnose Patient* use case and the *REQ1* requirement. Such a matrix is configured and filtered to display only specific model information, such as only the mapping of functional requirements to a specific system of interest.

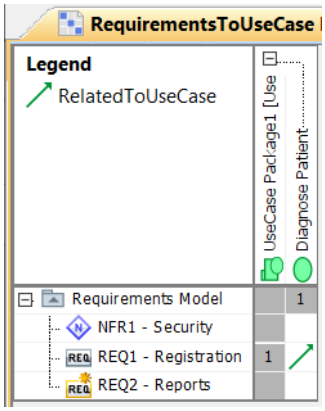


Figure 8.7: A matrix showing the relations between requirements and use cases.

Quality Model. This DSML building block allows modellers to define the guaranteed and required qualities of a system. Figure 8.8 shows the different required qualities that have been configured for the adjustment of a collimator and the examination table. Additionally, modellers establish and define the relevant matching dependencies that exist between the required qualities and the guaranteed qualities. Tables that list the different qualities and matrices that show the relations of the requirements to the use cases are also configured.

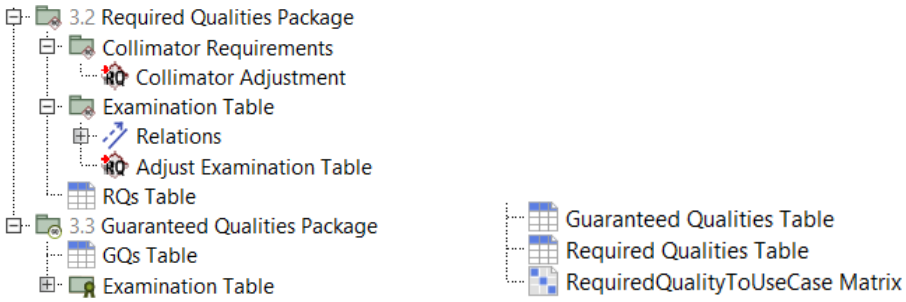


Figure 8.8: (Left) Model elements and (right) custom views such as quality table and matrices that are created using the quality DSML building block.

Figure 8.9 shows an example of required qualities listed in the quality table. In this example, the link to the use case models have been established by referring to the actors

Medical Assistant and *Patient*. Further, in the description, a detailed summary of the respective required quality is mentioned. Different qualities are grouped into individually created quality packages, thereby separating the concerns of the qualities based on the different systems of interest.

RQs Table

Add New | Add Existing... | Delete | Remove From Table

#	Name	Description	Actors
1	Examination Table		
2	Adjust Examination Table	The medical assistant must unlock the patient examination table to operate in longitudinal and/or transversal directions.	Medical Assistant Patient
3	Collimator Requirements		
4	Collimator Adjustment	The medical assistant must adjust the collimator to properly collimate the Xray beams.	Medical Assistant

Figure 8.9: A table listing the required qualities, its descriptions, and the links to the actors from the use case model.

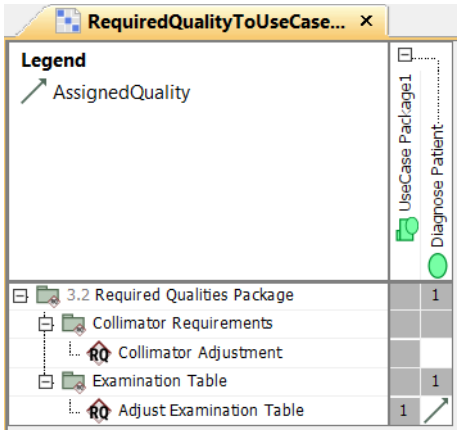


Figure 8.10: A matrix showing the relations between required qualities and use cases.

Figure 8.10 shows a matrix that displays the relations configured between the required qualities and the use cases. In this figure, the *Adjust Examination Table* quality is mapped to the *Diagnose Patient* use case through an *AssignedQuality* relation. The

matrix is configured to include both the required and guaranteed qualities. Other matrices are defined to refer to the other DSML building blocks.

Feature Model. This DSML building block allows modellers to define points of variability and commonalities in a system to be modelled by using features such as mandatory features, optional features, or features, and alternative features (Section 2.3.1). Figure 8.11 lists features that are part of a collimator system that was described earlier in Section 7.5. The *Collimator RX control* (receive) and *Collimator TX control* (transfer) are mandatory sub-features of the *Collimator Controls* feature that must be selected as part of any product or product line that configures this feature. The *Steering* is an *Or* feature, meaning at least one of these *Or* features must be selected as part of a product or product line configuration. The alternative sub-features *Down* and *Up* as part of the *Xray Scatter Control* feature ensures that either of these sub-features are selected during product or product line configuration.

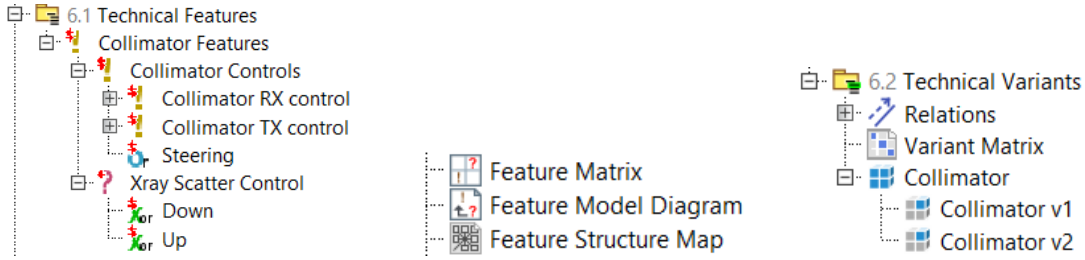


Figure 8.11: (Left) Feature model elements, (centre) custom views, and (right) variant configurations created using the feature model DSML building block.

Figure 8.12 shows a feature structure map that represents the structure of the features and shows the relations between the elements of the entire model. This tree based hierarchy helps users envision features of their system and understand the various connections between the different features and the sub-features. The icons help identify the type of the feature more easily, therefore, they are essential in fostering a good UX.

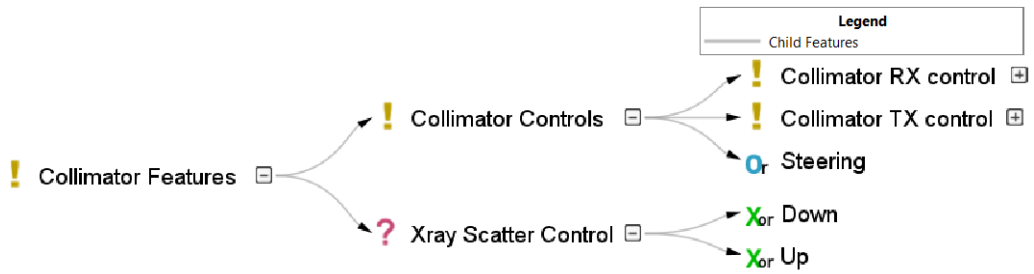


Figure 8.12: A feature structure map showing the hierarchy of the features.

Figure 8.13 shows a matrix representing the configured relations between the features and the requirements they realise. The relation is directly configured on the matrix and specific features are selected to realise a specific requirement. In this figure, all the features of the collimator realise the requirement needed for adjusting the collimator.

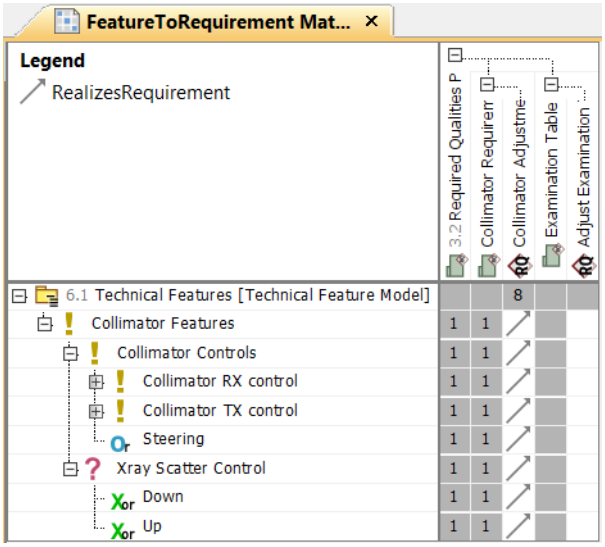


Figure 8.13: A matrix showing the relations between required qualities and use cases.

Figure 8.14 shows a feature model diagram showing the relations between the *Xray Scatter Control* features. In this figure, the optional feature *Xray Scatter Control* consists of two *Xor* features *Down* and *Up*. While both *Down* and *Up* are required for the *Xray Scatter Control* feature, the relation between the *Xor* features are in conflict, as only one of these features are chosen when the *Xray Scatter Control* feature is configured.

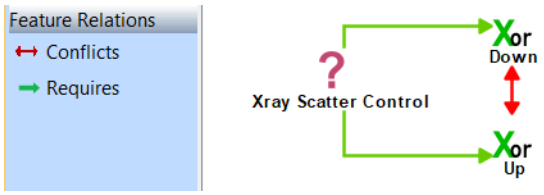


Figure 8.14: A feature model showing relations between an optional and Xor features.

Further, the DSML building block also defines the relations such as requires or excludes, from either a customer point of view or from a technical point of view. Based on the variability, feature models (Def. 3) are also used to define a product line configuration that specifies which features are part of a product or a product line. Previously in Figure 8.11, an example of a collimator as the product line was shown, and the different

versions of the collimator were configured as products. Thereby, *Collimator v1* can be configured with a selection of both the *Collimator Controls* (mandatory feature) and the *Xray Scatter Control* (optional feature), whereas the *Collimator v2* can be configured only with the *Collimator Controls* mandatory feature. This is later demonstrated through a variant configuration GUI in Figure 8.29. To further distinguish between a customer and a technical feature model, the SHS DSML describes both of these concepts as individual building blocks but primarily based on the same feature model concepts and reuse of most language components.

Function Model. This DSML building block allows modellers to define functions for a system in a way that it describes how a system is observable from the outside (Section 2.3.2). The functions are hierarchically decomposed into sub-functions that perform smaller tasks and are associated with elements in the overall system's architecture. Function models (Def. 4) describe a system from a purely functional perspective, and the DSML building block is configured to provide a library of functions, with its inputs, outputs, and external elements such as environmental effects, sinks, and sources, and provide different views to model various parts of the system.

Figure 8.15 shows a library of functions including the defined input and output ports. The *Position Patient and Generate Report* function consists of two input ports *TablePositionRequest* and *SystemPositionRequest*, and one output port *PatientInformation*. The *TablePositionRequest* and *SystemPositionRequest* are configured with an *HMI* parent input type, whereas the *PatientInformation* is configured with the *Signal* parent output type. The above mentioned model elements are configured with the types defined in the lower part of the figure under the *In/Out Types* package. The parent input and output types of these types are listed under Figure 8.16. Thus, domain-specific elements are created from these generalised set of inputs and outputs.

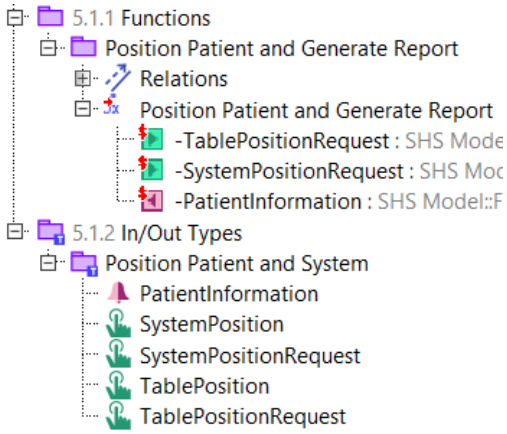


Figure 8.15: Functions and input/output types created for the function model.

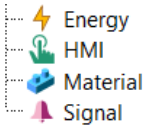


Figure 8.16: Input and output types that are created using the function model DSML building block.

Figure 8.17 lists the various matrices, relation maps, and tracings that are made to other DSML building blocks. As an example, relations from different functions to specific use cases, requirements, or qualities are configured through these matrices. They are generally configured under specific folders to logically group the different matrices that exist within the function model.

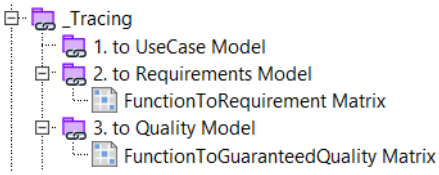


Figure 8.17: The matrices created to refer to models of other DSML building blocks.

An example of a matrix that establishes the relations between the functions in a function model to the requirements are shown in Figure 8.18. Here, the *Position Patient and Generate Report* function realises the functional requirement *REQ2 - Reports* and the non-functional requirement *NFR1 - Security*.

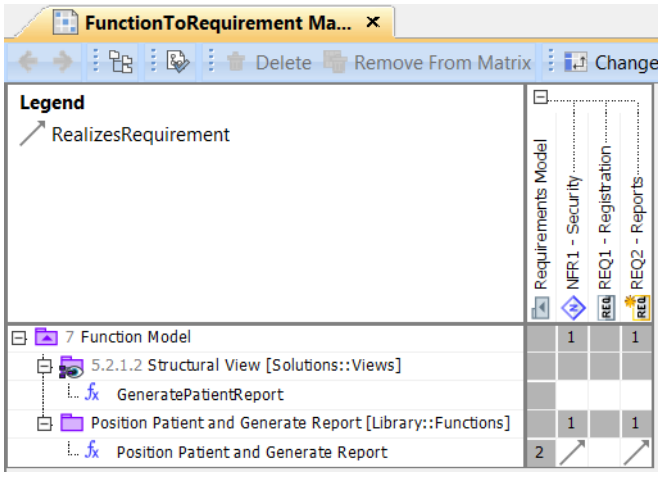


Figure 8.18: A matrix establishing relations between functions and requirements.

Figure 8.19 shows the different views that are configured as part of the function model.

In particular, the context view describes the overall context of the entire system or a part of the system. The structural view describes how the different functions are organised throughout the system. This is achieved through a function connections diagram that shows the different functions and their internal connections. The behavioural view describes the function interaction scenarios between the actors and the functions through messages. These interaction diagrams are fundamentally UML interaction diagrams, but adjusted to the domain-specific needs.

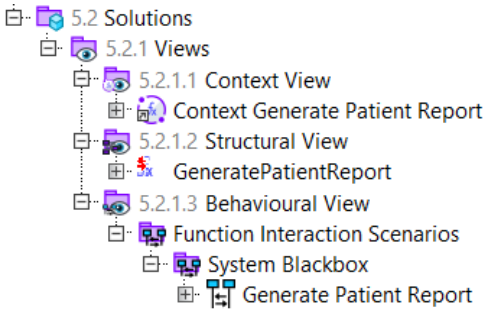


Figure 8.19: The context, structural, and behavioural views for the function model.

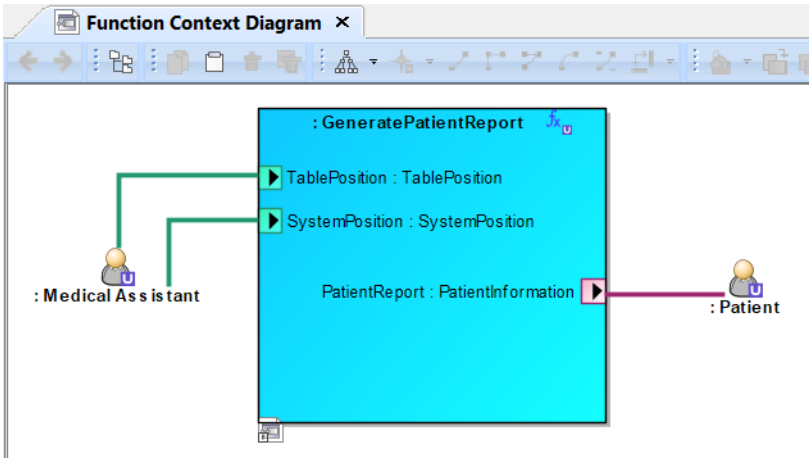


Figure 8.20: The context showing the actors and the function as observed from outside.

In a context view, a function context diagram is configured with the model elements that shows how the systems looks to the outside world. Figure 8.20 shows the two actors *Medical Assistant* and *Patient* and how they are associated through function flows to the *GeneratePatientReport* function. The inputs to the function are provided by the *Medical Assistant* and the output of the function is provided to the *Patient*. A more detailed look at the function is shown in the structural view of the function context in

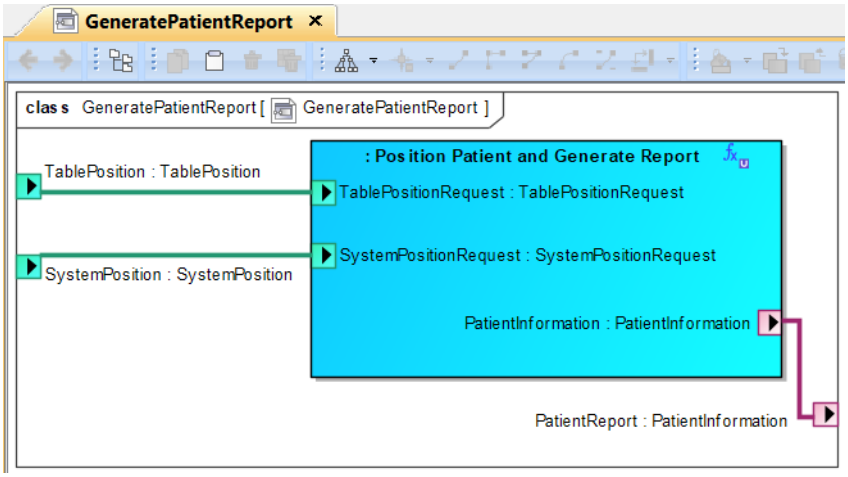


Figure 8.21: The structural view for decomposing a system into sub-functions.

Figure 8.21. Here, the input ports and their connection flows to the inner functions are configured. Further sub-functions are configured for the *Position Patient and Generate Report* function. This is particularly beneficial as functions must be decomposed into smaller functions to effectively design a system of interest. Structural views are therefore aimed at providing the entire decomposition of the system into smaller functions.

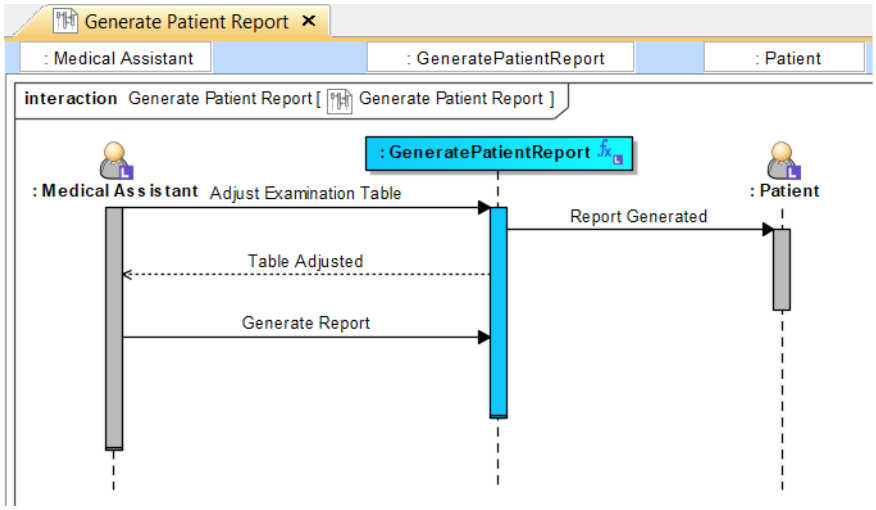


Figure 8.22: An interaction diagram for the function model.

The function interaction diagram in Figure 8.22 depicts the behavioural view of the system. Here, the interactions between the actors and the functions are shown in a

sequence diagram through different lifelines. For example, a *Medical Assistant* first sends a function message to the *GeneratePatientReport* function to *Adjust Examination Table*. Upon receiving a reply *Table Adjusted*, the *Medical Assistant* then sends a *Generate Report* message back. Once the internal functions in *GeneratePatientReport* completes the processing of the patient information, the function forwards the *Report Generated* message to the *Patient*, thereby signalling the end of the interaction of these elements.

Architecture Model. This DSML building block allows modellers to define the physical decomposition of a system using various system elements such as assemblies, electronics, mechanical, chemical, as well as software elements (Figure 8.23). Additionally, each of these elements are configured with cables, connectors, ports, or voltage configurations and allows modellers to set the scope of the entire system architecture using decisions, risks, standards, patents, and other quality standards.

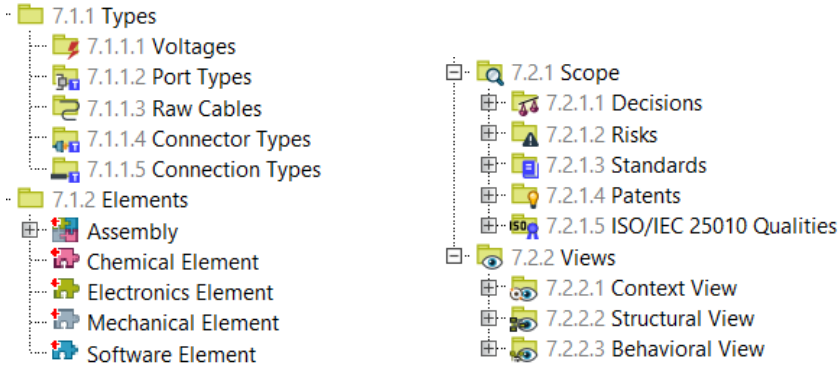


Figure 8.23: (Left) Model elements, and (right) architectural decisions, standards, risks, patents, ISO product qualities, and views that are defined by a modeller using the architecture model DSML building block.

Similar to the decomposition in function models, the architecture model allows visualisation through different views such as context, behavioural, and structural. Further, references to the feature models, such as defining a product line of architecture elements as features is also allowed through variability configurations. Figure 8.24 lists the electronic elements on the left along with the respective mechanical, power, optical, or software ports. The model elements are grouped together logically into their respective packages. Further, the possibility to configure ports according to the respective port types is possible. This allows language engineers to define validation rules in MagicDraw that check if same typed ports are connected through a port connection. This means that an output from a power port cannot be connected to an input of a software port. On the right, different quality criteria defined as part of ISO/IEC 25010 [HSBAC17] are modelled and attached to different model elements in the architectural model. This

allows modellers and domain experts to quickly validate if the respective model elements meet the quality criteria.

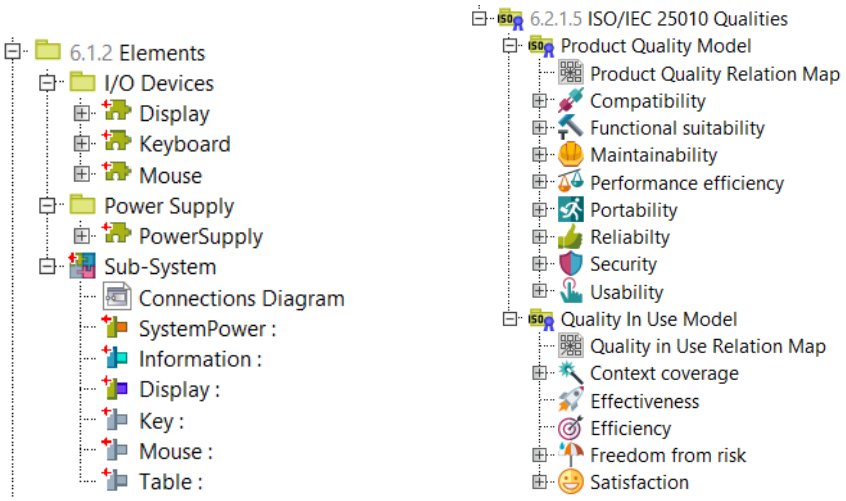


Figure 8.24: (Left) Defining I/O devices, power supply, and the decomposition of the sub-system, and (right) ISO/IEC 25010 qualities for this architectural model.

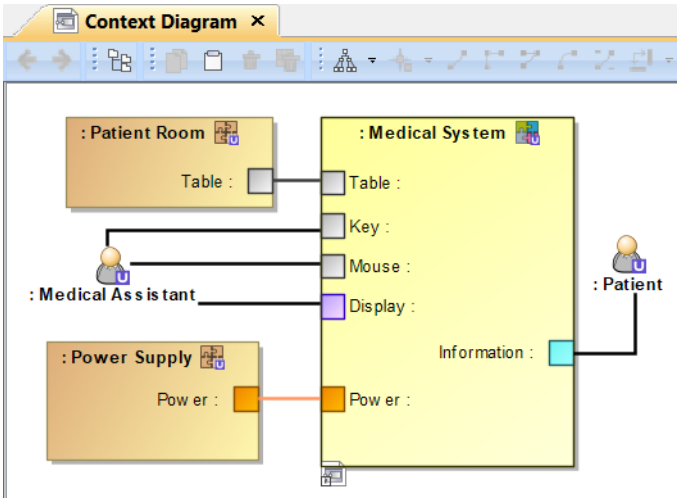


Figure 8.25: An architectural context diagram described for the architectural model.

Views such as context, structural, and behavioural follow the similar layout as described for the function model. In Figure 8.25, an architectural context diagram is depicted that shows how the different architectural elements are connected to the other elements. In particular, elements such as *Patient Room* defines table, that is used in

the entire system of interest. The power supplied through a *Power Supply* allows the operation of I/O devices and the generation of information within the *Medical System*. The associations of the actors, *Medical Assistant* and *Patient*, to the *Medical System* is also defined in this context diagram.

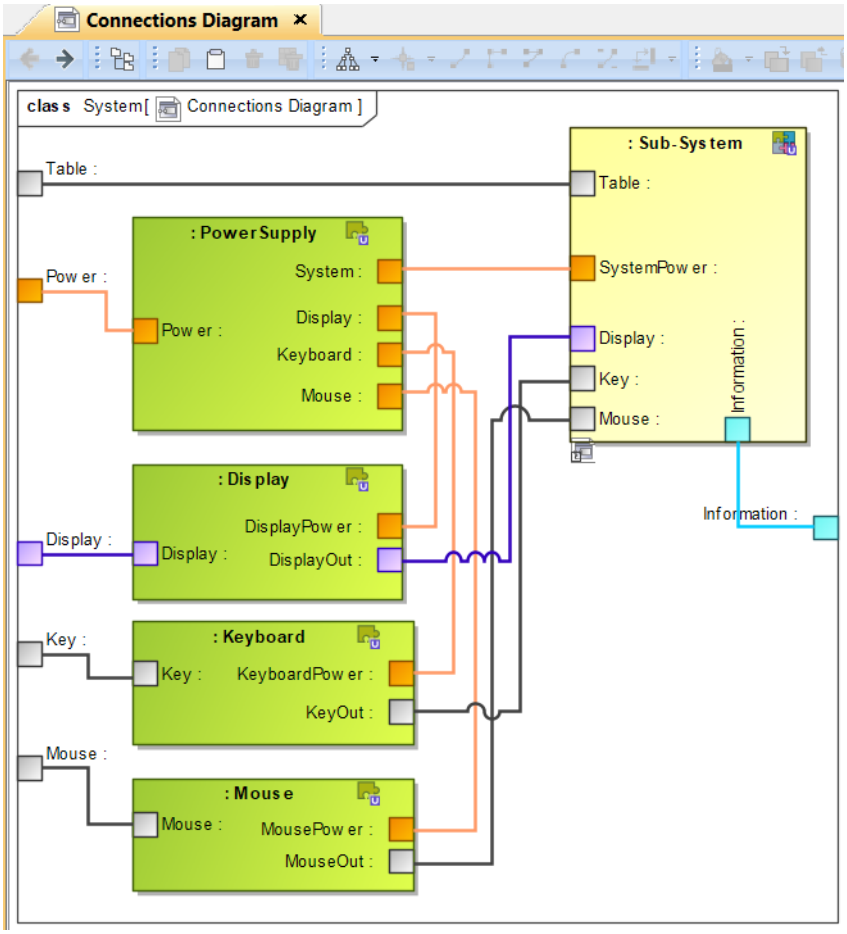


Figure 8.26: An architecture structural diagram described for the architectural model.

Figure 8.26 shows an architectural connections diagram that shows the structural view of the architectural model. This view shows the decomposition of the various model elements described in the context view and presents the models in a more detailed and structured manner. For example, the power supply is configured on a *PowerSupply* electronics element that provides further power to the *Display*, *Keyboard*, *Mouse*, and the *Sub-System*. Similarly, other I/O devices also connect to the sub-system through various port types. The output of the sub-system is a software port that provides the final generated report to the patient in the form of *Information*.

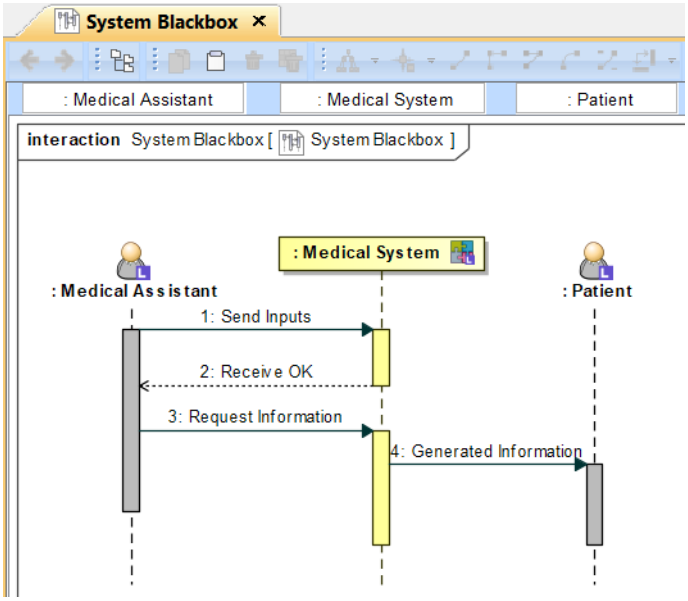


Figure 8.27: An architectural interaction diagram described for the architectural model.

Similar to the function interaction diagram, Figure 8.27 describes the architectural interaction diagram in the form of sequence diagram with different lifelines. Here, the system and the actors are the architectural elements, where the exchange of messages between the actors and the system occurs. This behavioural view allows the definition of interaction diagrams as well as state machines that help associate different interactions and activities to different states that the system could reside in.

Language Components

Chapter 4 discussed defining the language components and composing languages using the example of modelling use cases, actors, and their activities. Language components from these DSMLs were reused and extended to define the language of the `UseCase` building block described in this chapter. To compose a use case DSML building block, the previously defined DSMLs (Section 4.1) were composed together by way of embedding the ActorDSML into the UseCaseDSML (Section 4.5.3). This means that a modeller uses the `UseCase Model` in the SHS DSML to create models of use cases and their actors. However, previously the languages were created stand-alone, meaning they were not associated with any other domain concepts. In Figure 8.1, the introduction of `_Tracing` packages in different DSML building blocks and configuration of various dependencies to other DSML building blocks allow modellers to easily associate model elements across various aspects of a domain. This fosters separation of domain concerns, reusability, and

modularity of different domain concepts as these building blocks are now easily reused for other DSML projects without significant modifications to the language definition. Previously, Figure 8.1 showed the separation of a feature model for realising features for customers and technical scenarios. Both of the defined DSML building blocks along with their language components are fundamentally similar, therefore only one DSML building block are reused as-is to foster modularity. Changes in the identifiers, e.g., name of the model elements, of the respective building blocks ensure the separation of customer concerns of feature model in comparison to the technical features of a system.

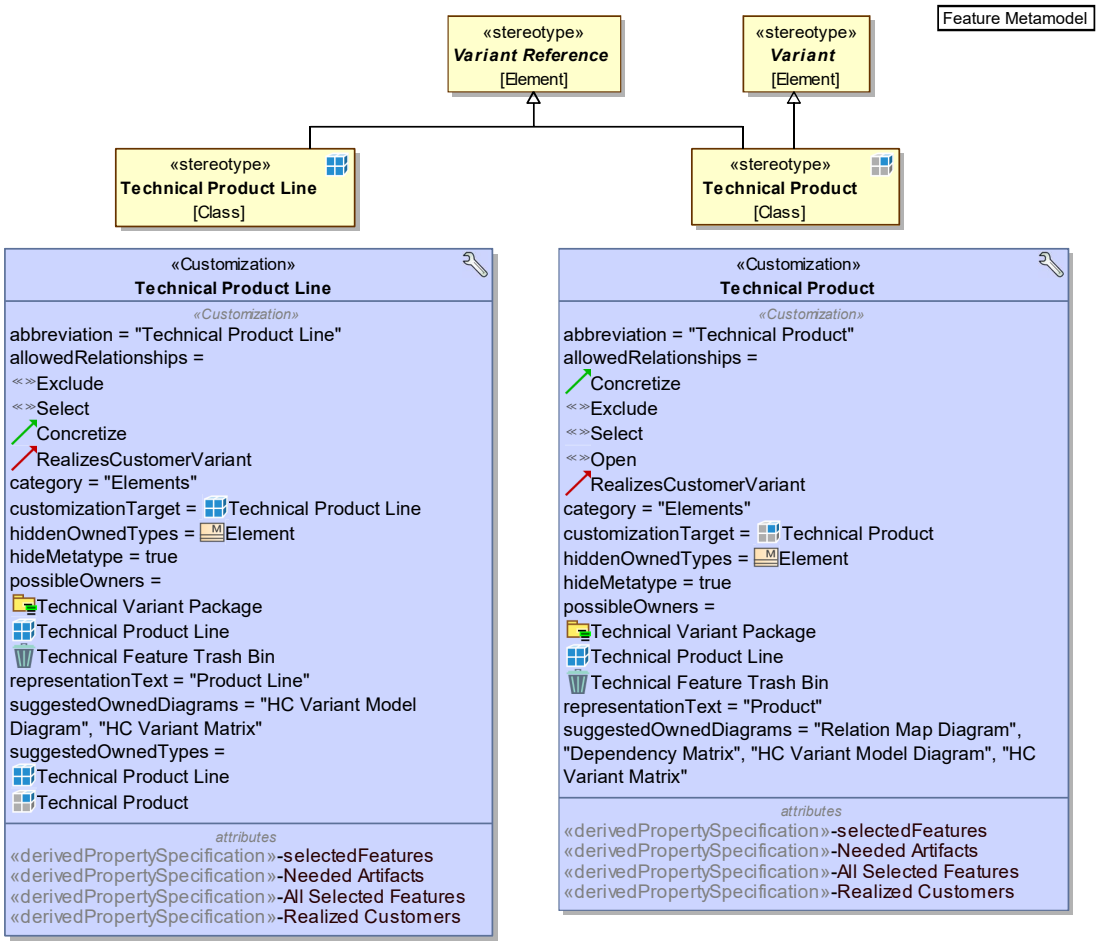


Figure 8.28: The MagicDraw metamodel for the product line and product language elements in a feature model DSML building block. Figure taken from [GJRR22a].

Figure 8.28 shows the configuration of a (technical) product line or product using

the customisation capabilities of MagicDraw. Here, the stereotypes that are attached to the product line or product is enhanced by providing property configurations that ensures the behaviour of these elements in the SHS DSML. In this example, a product line owns (`suggestedOwnedTypes`) both another product line or a product, while a product has no such specification, therefore only exists under the `possibleOwners`. These definitions of the elements are located inside a language profile (Section 3.4.1) that constitute one of the language components (Chapter 4) of the DSML building block. Similarly, the features of the feature model are also defined in a language profile and customised accordingly such that all the properties of the individual features in a feature model (Def. 3) are met.

UXD Considerations

The examples presented so far in this case study discuss the definition of the SHS DSML as well as some of the model elements that are created from the individual DSML building blocks. Previously, UXD aspects were discussed in Chapter 6 detail using a real industrial example of feature models that was built using the SHS DSML. Figure 8.29 shows the variant configuration GUI of *Collimator v1* using icons (V1), colours (V2), modal dialog (V3) detailing documentation and issues, and layout (IA1) on a custom GUI (ID4).

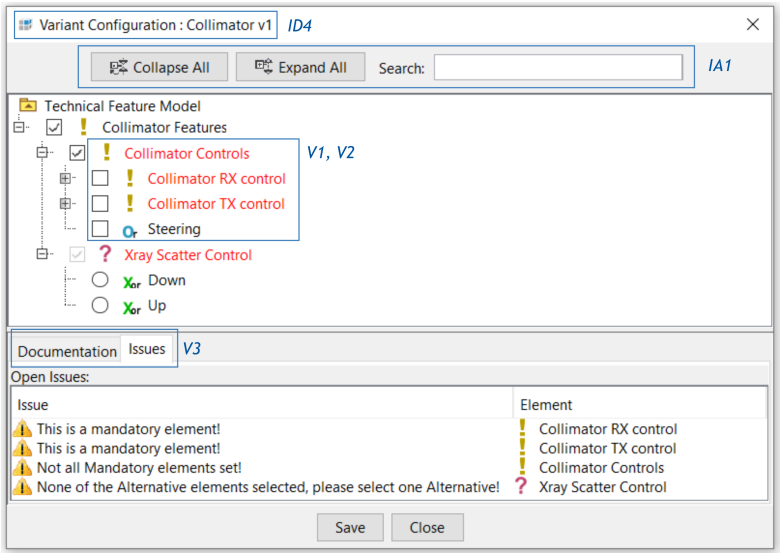


Figure 8.29: The variant configuration custom GUI for a product line which shows the different selectable features as well as documentation and issues for the current selection.

Figure 8.30 shows a feature model diagram (V4) that describes dynamic view plugin (V5) for filtering information, layout (IA1) for positioning the model elements, model

browser (IA2) for navigating model elements, perspective (IA3) that limit the set of functionalities, creation view (IA4) for quick creation of model elements, project template (ID1), and default assigned names to model elements (ID2). The example shown in the figure is based on the model elements that were shown earlier in this chapter as part of the feature model DSML building block for the collimator case study.

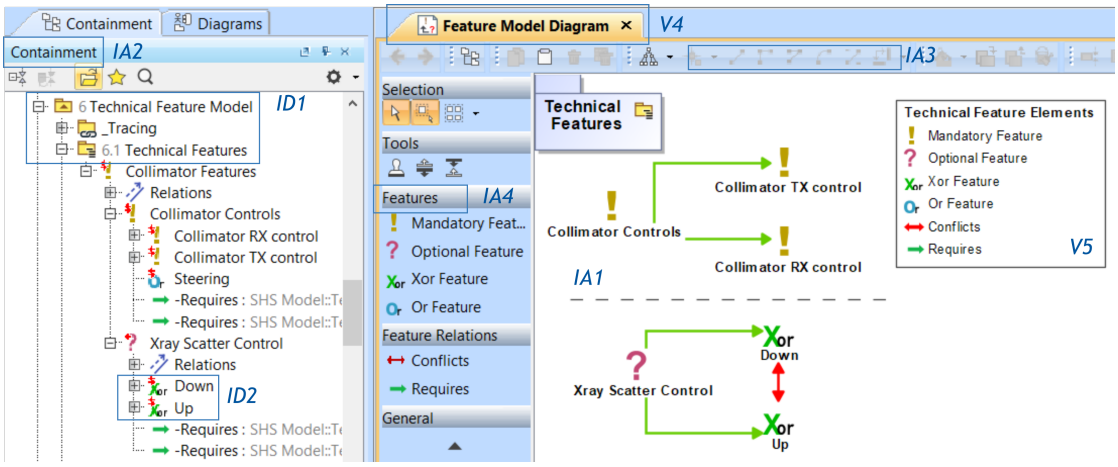


Figure 8.30: An example of a feature model showing different UXD aspects.

User-Centric Recommendations

Using a custom GUI (ID4), certain methods, guides, and recommendations are made for a feature model that is shown in Figure 8.30. These recommendations are based on the DSML guidance infrastructure concepts described in Chapter 7.

Figure 8.31 shows the overview tab for the feature model described in Figure 8.30. Here, a general overview of the feature model is provided, along with hyperlinks and training videos. Frequently asked questions contain information that have been previously asked by domain experts or modellers, thereby allowing such a guidance infrastructure to be directly integrated in the DSML itself. Each individual DSML building block would have their own guidance and recommendations information therefore language engineers also build the dataset according to their domain knowledge.

Figure 8.32 shows recommendations for specific elements on the feature model diagram. Here, recommendations such as specific properties of a feature are suggested to modellers. These recommendations can, therefore, change based on the current modelling situation, and be updated in real time. The recommendations are hence considered active and in sync with the models, and are considered model-aware.

Figure 8.33 shows a process model for the feature model diagram. Here, various processes are described such that the modeller is aware of their current state of modelling,

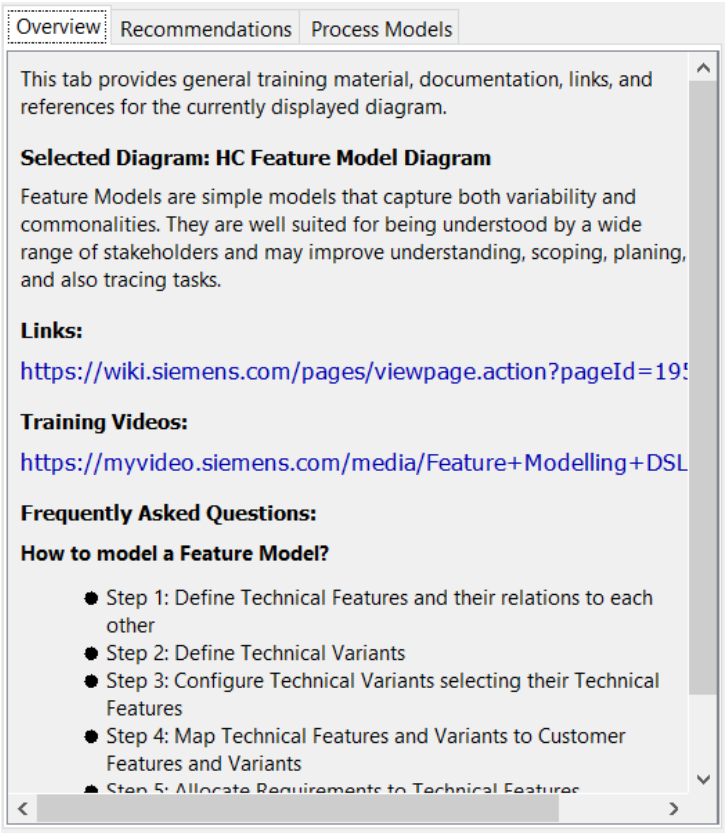


Figure 8.31: The overview tab showing the documentation, hyperlinks, training videos, and commonly asked questions about the feature model in Figure 8.30.

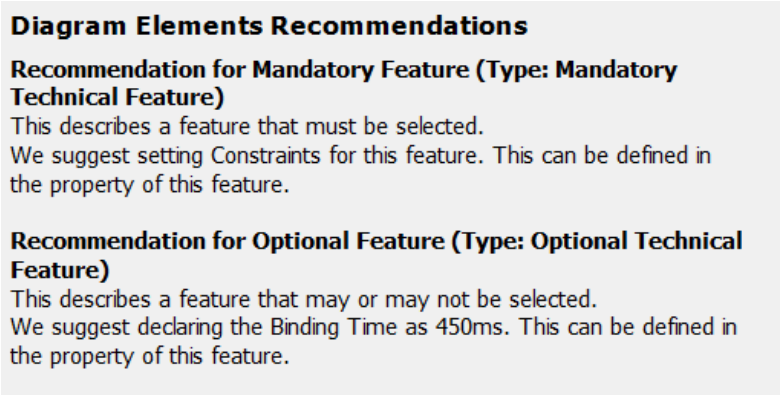


Figure 8.32: The recommendations for specific elements on the feature model diagram.

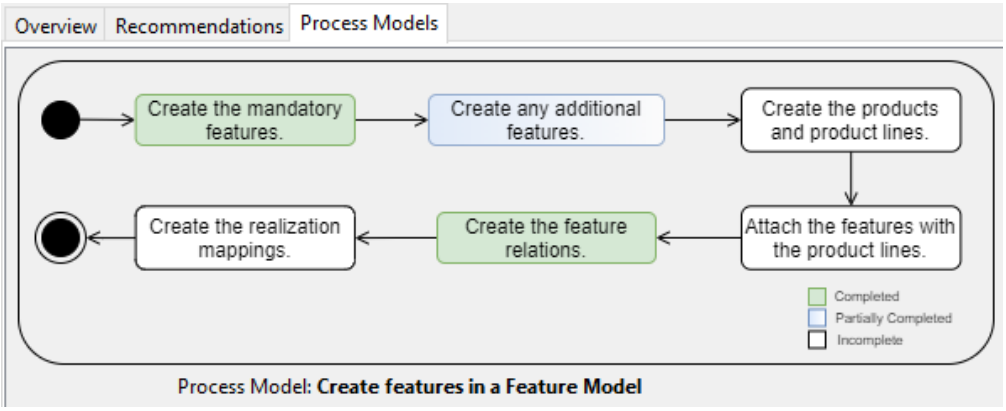


Figure 8.33: A process model for the feature model described in Figure 8.30 showing the different processes that a modeller must consider.

and can accordingly design their models. By providing a comprehensive guidance infrastructure, language engineers support modellers in effective modelling for every single domain aspect and update information frequently without the need to constantly update the DSML building blocks or the language components.

8.1.3 Discussion

Individual chapters in this thesis have looked at specific examples in an industrial context. The evaluation of this case study was performed by a group of language engineers, domain-experts, and modellers at Siemens Healthineers, Siemens AG, and researchers at RWTH Aachen University. The focus group involved project members with 8-15 years of experience in software and systems modelling to understand the current challenges faced in the healthcare domain. The participants considered the development of an active and in sync infrastructure to be valuable in providing continuous support for all kinds of modellers. This was primarily due to the fact that current solutions only rather provide a plain technical view of the entire language engineering infrastructure. The development of such an infrastructure is not only valuable in providing methodological guidance, but also fosters the modular development of language components in order to continuously evolve the DSML in the ongoing modelling projects. Further, such a complete infrastructure implements aspects of being aware of the current state of the models, whether they are context-sensitive or evolve dynamically as DSMLs become more complex. Data from previously designed Siemens Healthineers projects was reused, that were the basis for the examples provided in this thesis. Often, the participants collectively worked together during the course of this project, to develop certain functionalities of the custom GUI, as well as to configure various business rules and recommendations for the guidance infrastructure. Prior to the guidance infrastructure presented in this thesis,

the practitioners used to navigate through pages of static training data, that included books, handbooks, tutorials, and overly generic technical documents, as no alternative solutions to provide a complete guidance infrastructure had been set up. As a result, language engineers often struggled in order to effectively represent domain concepts, and to use techniques for reusing and modularising commonly used domain constructs. This case study aimed at providing insights to language engineers into the development of DSML building blocks in a real industrial context as well as combine the aspects of developing reusable and modular language components, integrated guidance infrastructure, and better UX considerations in order for a domain expert or a modeller to successfully model their systems.

8.2 Case Study 2: Siemens Digital Industries (DI)

8.2.1 Motivation

Combining the real world of automation with its equivalent digital counterparts in the information technology world allows industries to make better decisions and bring about an accelerated change in fostering a digital enterprise. The Siemens Digital Industry (DI) is a division of Siemens AG that makes such a decisive contribution for automation and digitisation such that customers drive their digital transformation much more effectively. To understand the digital journey better, specific teams within DI collect, understand, and eventually use large datasets in domains such as Industrial Internet of Things (IIoT), supply chain disruptions, and digital twin approaches. A strong and interlinked value chain is possible using optimisation techniques that extends from the design of product and production, through manufacture, to finally use and recycling. Innovations, new business models, and better documented architectures are therefore necessary for adding value to all partners in an ecosystem, be it a completely software-driven ecosystem, or an ecosystem with both system and software aspects. DI, therefore, uses an architecture documentation template to model their software and system architectures for their SINAMICS[1] and SINUMERIK[2] control systems.

8.2.2 DI DSML, its Building Blocks, and the Models

To realise the modelling journey for DI and be able to document architectures, a set of DSML building blocks were created or reused (partially or fully). The template of arc42 [SSZM19] is used for architecture documentation by DI. Each arc42 template section represents a DSML building block built in MagicDraw for modelling the specifics of a certain part in the architecture documentation. Figure 8.34 details the models that

[1] https://www.siemens.com/global/en/products/drives/sinamics.html
[2] https://www.siemens.com/de/de/produkte/automatisierung/systeme/cnc-sinumerik.html

are created automatically and instantiated during MagicDraw tool start up for each individual template section for the arc42 architecture documentation template. The collection of all the thirteen DSML building blocks is collectively referred to as the DI DSML or the arc42 DSML.

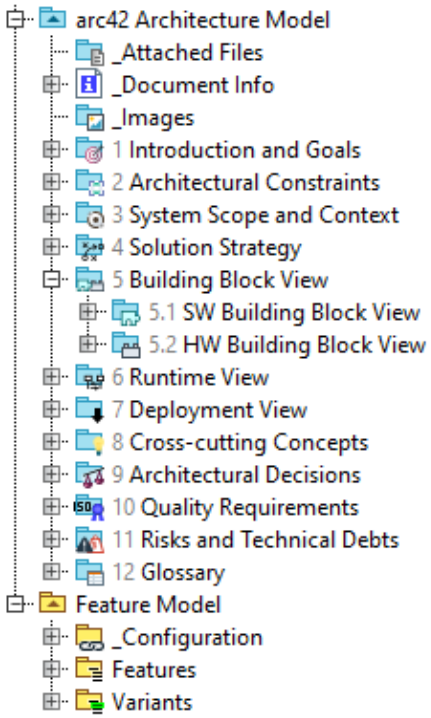


Figure 8.34: The models of a DI DSML created automatically representing the arc42 template concepts.

Introduction and Goals. This DSML building block allows practitioners to create mandatory or optional functional requirements, stakeholders involved in the system, and the various quality goals for the architecture. Figure 8.35 lists the various models that are created as part of the introduction and the quality goals. These models also include custom MagicDraw diagrams (Section 6.3.1) that help visualising more effectively the this part of the arc42 template.

Constraints. This DSML building block allows practitioners to model the technical and organisational constraints in regard to their system design decisions.

System Scope and Context. This DSML building block allows practitioners to delimit a system from all of its users and systems, while differentiating between the business context such as domain specific inputs and outputs with the technical context such as protocols or accompanying hardware.

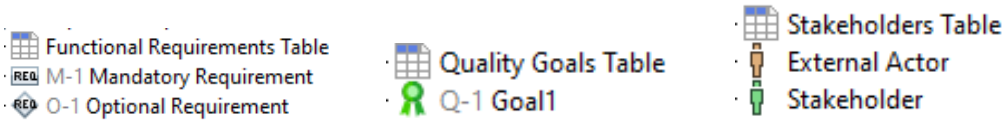


Figure 8.35: (Left) Requirements, (centre) quality goals, and (right) stakeholders that are created using the introduction and goals DSML building block.

Solution Strategy. This DSML building block allows practitioners to model technology decisions, architectural patterns, guidelines to achieving key quality goals, and any other relevant organisational decisions.

Building Block View. This DSML building block allows practitioners to decompose a system into different architecture modules, architecture components, classes, interfaces, packages, frameworks, and so on, while also allowing the modelling of their relations. The building block view itself is different from the term DSML building block used throughout this thesis, as the view strictly refers to the reusable units of the hardware and software architectural views. Figure 8.36 shows the software and hardware architectural views and the different parts along with the ports that are created for each subsystem.

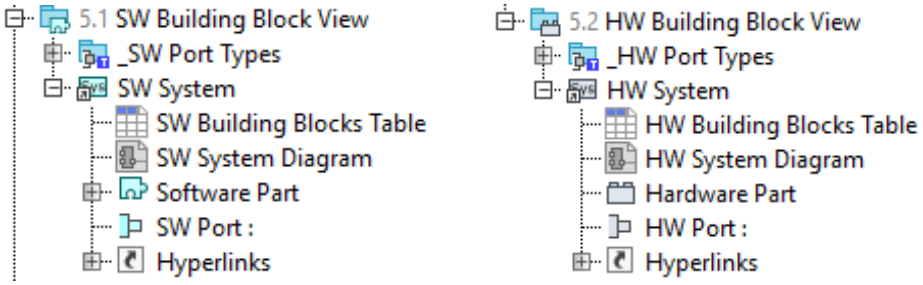


Figure 8.36: Models of (left) software system parts and (right) hardware system parts that are created using the architectural building block view.

Runtime View. This DSML building block allows practitioners to provide additional information regarding the behaviour and interaction of various parts of the system for the views defined in the building block view. These include execution of use cases, interactions at critical external interfaces, and error or exception scenarios.

Deployment View. This DSML building block allows practitioners to describe the technical infrastructure of the system, and the mapping of the software parts to the corresponding hardware infrastructure, which introduces cross-cutting concepts.

Crosscutting Concepts. This DSML building block allows practitioners to model those aspects of the system that cut across the different sections of the arc42 template. They include domain models, architectural patterns, and other concepts that are tied to a variety of hardware and software system parts.

Architectural Decisions. This DSML building block allows practitioners to model those architectural decisions that prefer one solution over the other and includes rationales describing why a certain decision was preferred over the other. This helps stakeholders in comprehending the architectural design decisions that were taken during the architecture of a system.

Quality Requirements. This DSML building block allows practitioners to model various quality requirements that are associated with a specific standard such as the ISO/IEC 25010 quality models [HSBAC17]. These quality requirements complement the quality goals that are defined using the introductions and goals DSML building block.

Risks and Technical Debts. This DSML building block allows practitioners to model a set of identified technical risks or any technical debts that are also ordered by priority. An overall risk analysis and measurement planning is always critical to any architecture that needs to be designed.

Glossary. This DSML building block allows practitioners to define domain and technical terms and their definitions that allows stakeholders to describe their system.

Feature Model. This DSML building block is reused completely from the SHS DSML and allows various features, products, and product lines to be associated and configured with certain hardware and software system parts that are defined using the arc42 template building blocks. While a feature model is not part of the arc42 template, DI often work with variants of system definitions to better describe their architectures, therefore composing the arc42 DSML along with the feature model DSML building block provides the necessary logical connection in defining variability in software and system architectures.

Language Components

Chapter 4 discussed defining the language components and composing languages using the example of modelling use cases, actors, and their activities. Further, the previous case study (Section 8.1) extensively discussed the definition and usage of the various DSMLs and its building blocks. For the DI DSML, many such language components were either reused entirely, extended, or modified to implement the arc42 template definition. The requirements and the stakeholder language components were partially reused, modified, and composed from the SHS DSML building blocks of use case model and requirements model (Figure 8.2 and Figure 8.5). In the SHS DSML practitioners could also model the non-functional requirements, but in the arc42 DSML practitioners model only the mandatory or optional functional requirements. By fostering modularity of language components it is easier for language engineers to define new DSMLs by reusing such concepts.

The software and hardware system definition such as mechanical assemblies or electronics, as well as the software elements are configured using the arc42 architectural building block view. This is analogous to the definition of the architectural model de-

fined in the SHS DSML (Figure 8.23). Here, various elements are defined and then be associated with a feature model in order to define a product, a product line, or variants of the same (Figure 8.11). Therefore, the arc42 building block view allows various features, products, and product lines to be associated and configured with various hardware and software system parts. The quality requirements for the DI DSML reuses the language components that were already defined in the architectural model of the SHS DSML (Figure 8.24 right), for example, the ISO/IEC 25010 quality models. While not all language components from the SHS DSML are reused entirely, many are reused reducing the time taken for building the DI DSML from scratch as well as reducing errors and inconsistencies in different projects reusing similar domain concepts.

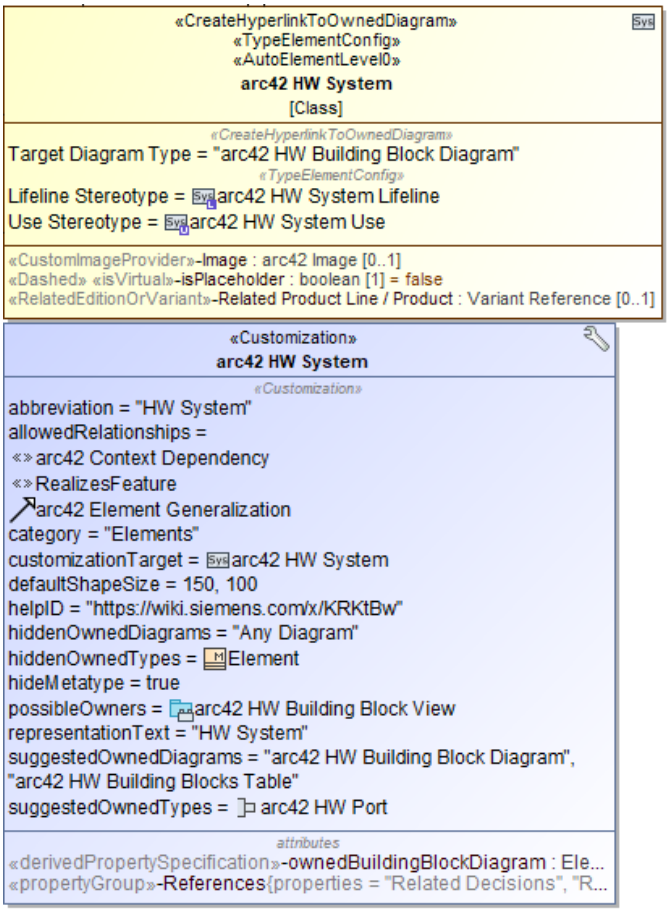


Figure 8.37: The language definition for creating the hardware system in an architectural business or technical context using the customisations of MagicDraw for a system scope and context DSML building block.

Figure 8.37 shows the configuration of an arc42 hardware system language element where relations to the feature model exist as part of the related product line. Relationships and possible owners define the behaviour of a hardware system and certain references allows cross cutting concepts to other model constructs such as relations to any architectural design decisions that were taken for this hardware system.

UXD Considerations

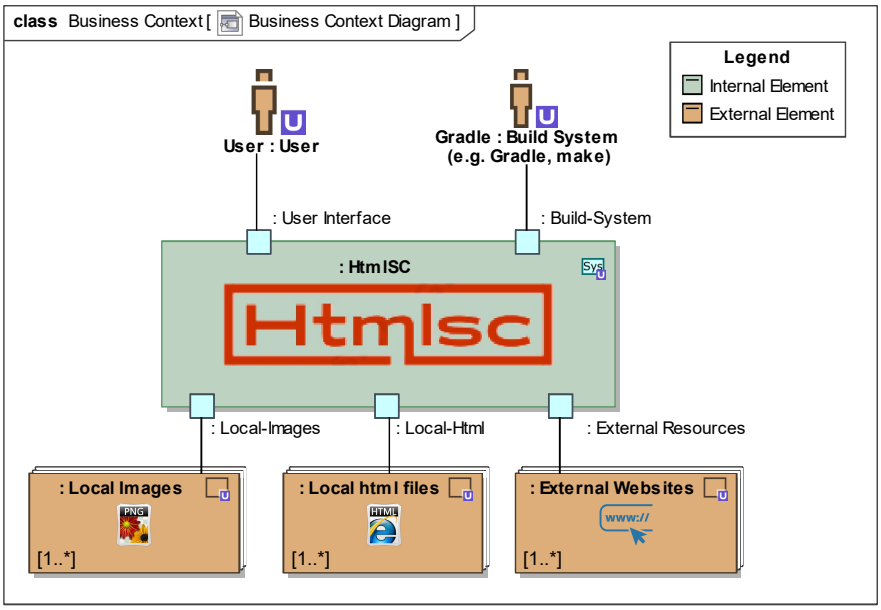


Figure 8.38: An example of an arc42 business context diagram in MagicDraw for an HTML sanity checker [SSZM19].

To illustrate the aspects of UXD that were considered in the arc42 DSML, let us take a look at an example of an arc42 business context diagram (V4) created in MagicDraw for a HTML sanity checker example [SSZM19] using the system scope and context DSML building block. Figure 8.38 shows the models used in an HTML sanity checker including various actors, software system parts, and external elements that are connected via software ports. Figure 8.36 defines these additionally parts using the arc42 architectural building block view. There UXD considerations for default naming scheme (ID2), project template (ID1), perspectives (IA3), and general usability heuristics were considered. In Figure 8.38, icons (V1), colours (V2), layout (IA1), and dynamic view plugin (V5) for filtering information are described that further show how UX is strongly considered to improve the overall experience of practitioners.

User-Centric Recommendations

The implementation of the guidance infrastructure that provides documentation, methods, and guides to a practitioner is achieved by using a custom GUI (ID4). This infrastructure (described in Chapter 7) allows language engineers to build an integrated mechanism directly within the MagicDraw tooling environment that allows modellers of the arc42 template to be guided appropriately.

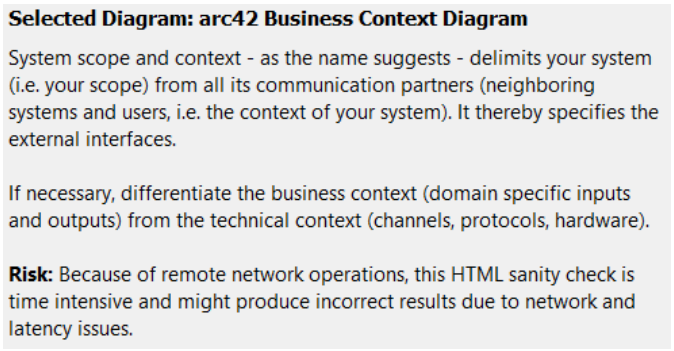


Figure 8.39: The overview tab consisting of documentation and risks related to the HTML sanity checker.

Figure 8.39 shows the overview tab of the HTML sanity checker example (Figure 8.38) that describes firstly a generic documentation for the system scope and context, such as the business and technical contexts, and also lists any additional information, hyperlinks, videos, and commonly asked questions such as if there are any risks involved in an HTML sanity checker. This kind of information is populated directly in the database that are easily configured and maintained by language engineers. Language engineers therefore build the appropriate dataset for providing arc42 specific recommendations to the modeller.

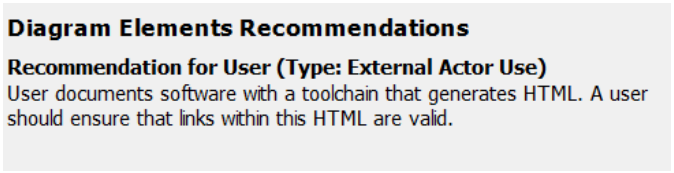


Figure 8.40: The recommendations for a specific actor in the business context of an HTML sanity checker.

Figure 8.40 shows recommendations for specific model elements on the business context diagram for an HTML sanity checker. Here, an example is provided for an actor, for which a specific recommendation is generated and displayed. In the HTML sanity checker

example (Figure 8.38), two actors were modelled, a user and a build system. However, only the recommendation for a user was provided here, as the dataset does not contain any specific recommendation for a build system, but it is easily added to the dataset depending on the specific models designed by a modeller. The recommendations are considered model-aware as the recommendations plugin identifies the various models in the current business context diagram, checks it against the pre-configured business rules in the database, and provides the active and in sync guidance to arc42 modellers.

8.2.3 Discussion

Similar to the evaluation of the SHS DSML, the evaluation of the arc42 DSML was performed by language engineers and software architects at Siemens AG, with the Technology research group and the Digital Industries (DI) group. This focus group consisted of project members with 8-12 years of experience in software architecture as well as software and systems modelling. The primary aim of DI was to use the arc42 template to build their own architecture documentation. The concepts of the arc42 DSML building blocks were provided by the arc42 template [SSZM19] and implemented in MagicDraw by the language engineers. Examples and datasets were configured to validate the different language aspects as part of the arc42 DSML. While not all concepts of the arc42 template were provided in the first version, the DSML and its building blocks were expanded in functionality as and when more requirements from DI were needed. Thus, it is also simple to introduce guidance and recommendations as and when new concepts are added to the arc42 DSML. By reusing many language components from the SHS DSML, language engineers were able to easily compose the arc42 DSML parts, as common concepts such as feature models, requirements, actors, or quality standards could be easily reused. Finally, better UX considerations meant that DI modellers and software architects were able to better comprehend and use the arc42 DSML with increased confidence.

8.3 Case Study 3: SpesML

8.3.1 Motivation

The public funded projects SPES [PHAB12] and its following project SPES_XT (SPES Extended) [PBDH16] have been previously developed to provide the necessary foundations for a comprehensive methodological toolkit in MBSE and model-based development. This methodological foundation helps advance the development of automated (and rather complex) embedded systems and provides a relevant direction to modellers in various systems such as software intensive CPS. In general, the SPES methodology is based on the scientific foundation of consistency and semantic coherence called FOCUS [BS12]. The SPES methodology is based on the following three principles [BBK+21]:

1. the design process must consider interfaces consistently;
2. the interface behaviour and the description of systems and their subsystems must be at different levels of granularity; and
3. the models must be defined based on a variety of cross-sectional topics and analysis options.

To this extent, in the SpesML project, a SysML workbench for the SPES method has been developed for by providing a customised SysML profile using the concepts of DSML building blocks and implemented in the modelling tool, MagicDraw.

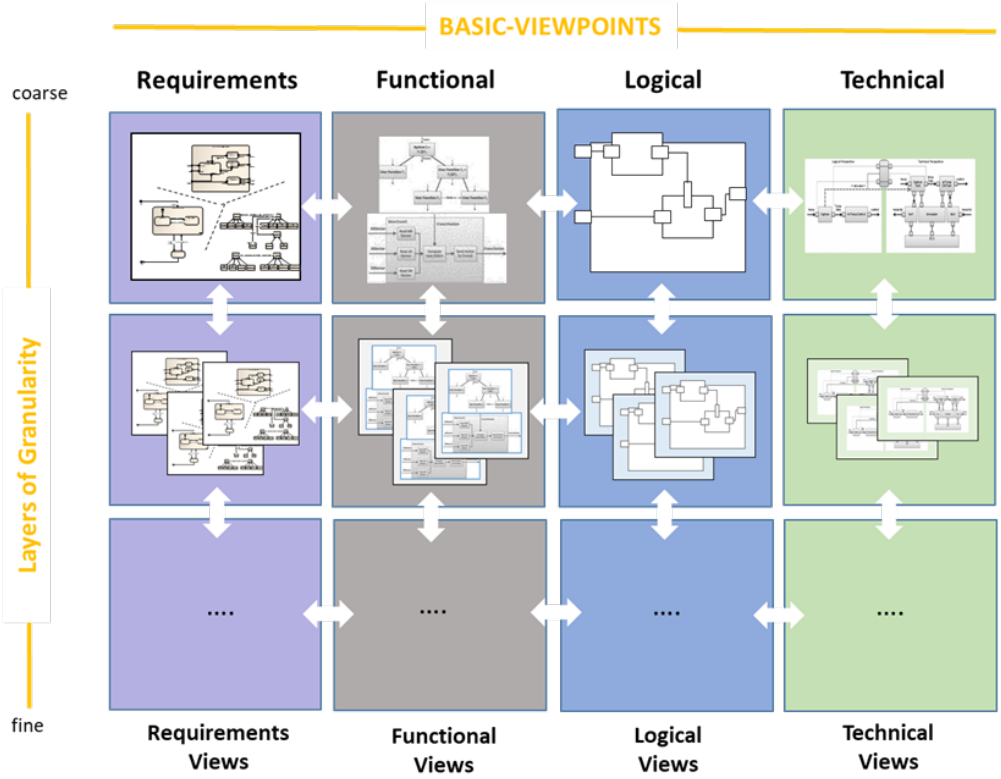


Figure 8.41: An overview of the four basic SPES viewpoints at different levels of granularity. Figure taken from [GJRR22b] and adapted from [BBK+21].

In general, SPES defines a system model in terms of a conceptual model that describes a system, its decomposed parts, and its properties. Therefore, SPES defines an MBSE artefact model based on the concepts of the standard ISO-42010 [ISO11] that assumes a System under Development (SuD) has an architecture and provides for various functions in the system. To achieve this, different viewpoints are considered in the

development process, that separates the concerns of different stakeholders for managing different artefacts of the system. To separate the concerns, the four basic viewpoints are considered at different levels of granularity: requirements, functional, logical, and technical (Figure 8.41). The requirements viewpoint forms the system requirement engineering activities. The functional viewpoint describes a set of system functionalities. The logical viewpoint describes the decomposition of the system functions in a logical manner. The technical viewpoint combines software and hardware aspects that are related to the SuD and how the system is realised. Some parts of this section have been published in a paper [GJRR22b]. Therefore, some passages from the paper may have been quoted verbatim in this section.

8.3.2 SpesML DSML, its Building Blocks, and the Models

To realise the SpesML project, a set of DSML building blocks were created that represent each of the SPES viewpoints. These individual DSML building blocks (Figure 8.42) separate the concerns of the SPES methodology and provides the necessary language infrastructure for the requirements, functional, logical, and technical viewpoints.

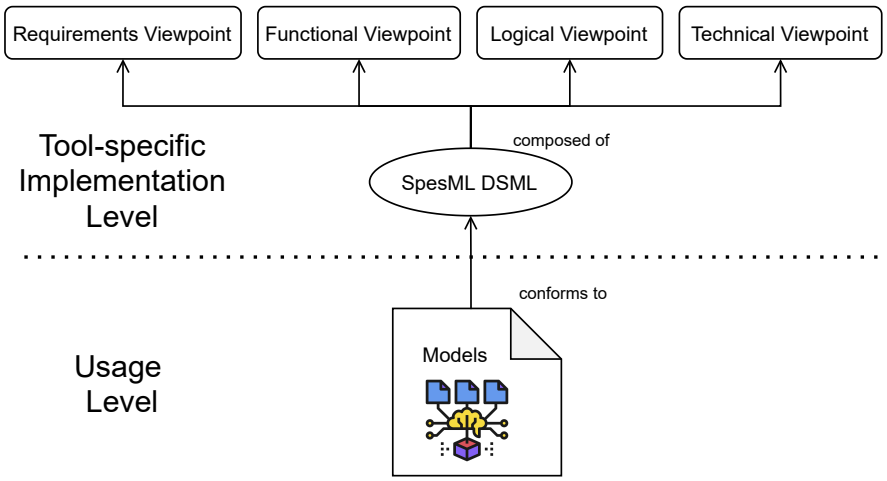


Figure 8.42: A conceptual model describing the different viewpoints as DSML building blocks. Figure adapted from [GJRR22b].

Figure 8.43 details the models that are created automatically during SpesML project creation in MagicDraw. The collection of the four DSML building blocks is collectively referred to as the SpesML DSML.

Requirements Viewpoint. This DSML building block allows practitioners to model and manage requirements for the SuD. Since requirements are generally used to model the necessities of the system, the DSML building block is similar to the one used in

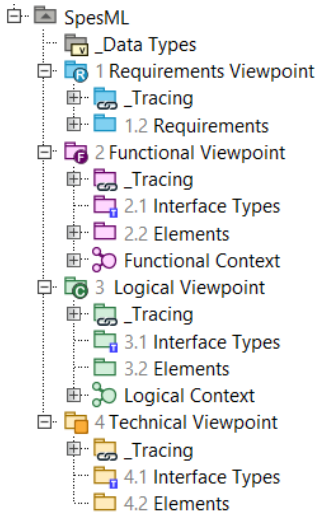


Figure 8.43: The models of a SpesML DSML that are created automatically when a new SpesML project is instantiated in MagicDraw. Figure adapted from [GJRR22b].

the SHS DSML. This means various language components were reused, extended, or modified from the SHS Requirements Model DSML building block or the DI Quality Requirements DSML building block allowing language engineers to more easily build the language infrastructure for defining various kinds of requirements such as capability requirements, functional requirements, or quality requirements for this SPES viewpoint.

Functional Viewpoint. This DSML building block allows practitioners to define functions, their sub-functions, and its properties for the SuD in order to describe the set of system functionalities as it is observed from outside the system. Language components that include functions, actors, and their contexts are reused from the SHS Function Model DSML building block. The UXD aspects are modified by the language engineers to adapt the language components to the newly composed models for the SPES functional viewpoint.

Logical Viewpoint. This DSML building block allows practitioners to decompose their SuD into various logical components for realising the behaviour specified in the functional viewpoint and independent of any technical realisation. Accordingly, certain language components from the functional viewpoint DSML building block are reused and adapted to the SPES logical viewpoint. Modellers also describe the logical context of the SuD, meaning a custom UML component diagram that specifies the different parts that lie within the system and those that lie outside the system.

Technical Viewpoint. This DSML building block allows practitioners to model those parts of the system that describes the platform specific components. This means that

the technical viewpoint describes a way to transition from platform independent logical views to a platform specific technical view. Therefore, hardware and software elements such as mechanical, electronics, or software parts are modelled along with the technical interfaces. Thus, language components from the SHS Architecture Model DSML building block are reused to compose the different elements and its types in the technical viewpoint DSML building block.

Language Components

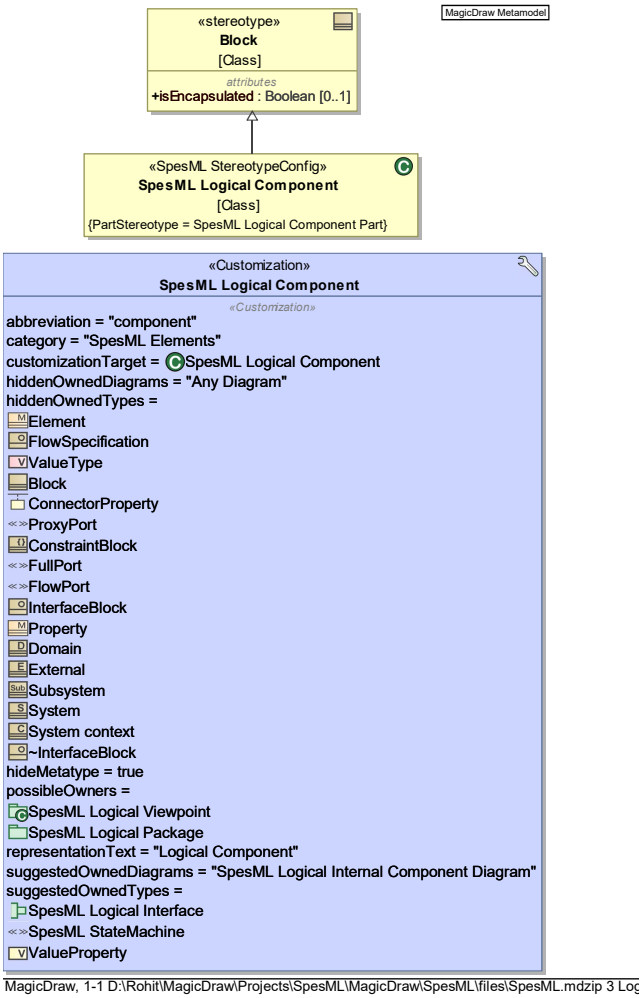


Figure 8.44: The language definition for creating a SPES logical component. Figure adapted from [GJRR22b].

Chapter 4 discussed defining language components and composing languages. Previous

case studies (Section 8.1 and Section 8.2) discussed the definition and usage of the various DSMLs and its building blocks. For SpesML, many such language components were either reused entirely, extended, or adapted to fit the SPES methodology. For example, requirements, functions, actors, and context belonging to the different viewpoints were partially reused, modified, and composed from the SHS and the arc42 DSML building blocks. In the SHS DSML, practitioners could model other requirements such as non-functional requirements, or environmental effects for functions, but in the SpesML DSML such language components were not used. By fostering a modular approach to building language components, language engineers easily compose new DSMLs and their building blocks. An example of a SpesML logical component definition in MagicDraw is seen in Figure 8.44 where the element itself is configured with a UML `class` metaclass, but inherits the `block` stereotype from a SysML profile, thereby customising the logical component to work similar to a SysML block element [H+06].

UXD Considerations

To illustrate the aspects of UXD for the SpesML DSML, let us take a look at an example of a window lifter system that is primarily used in industrial vehicles. Figure 8.45 shows a context diagram (V4) designed using the SpesML logical viewpoint. Here, various UXD considerations such as icons (V1), colours (V2), layout (IA1), and dynamic view plugin (V5) is shown. A reduced set of diagram toolbar options (IA3) allows practitioners to only create connectors between ports. A project template (ID1) is shown in Figure 8.43 that only allows practitioners to create the required packages first before creating the models using the different DSML building blocks.

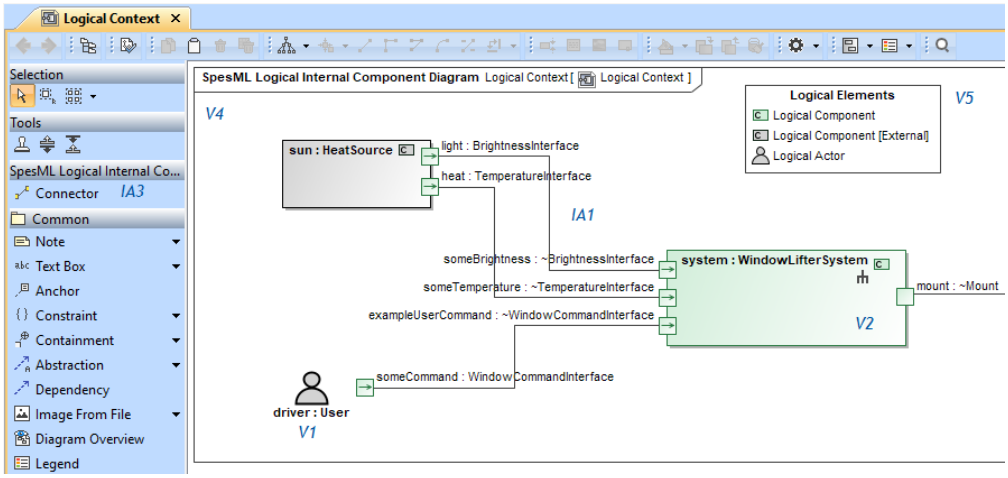


Figure 8.45: An example model of a window lifter system shown in the context of the logical viewpoint. Figure taken from [GJRR22b].

Figure 8.46 shows an additional creation view (IA4) where model elements are created from the predefined groups of either packages or elements. This is possible with the customisation of MagicDraw and to prevent any inconsistencies in the structure and organisation of model elements with the DSMLs. General usability heuristics were also considered to foster a good UX for practitioners.

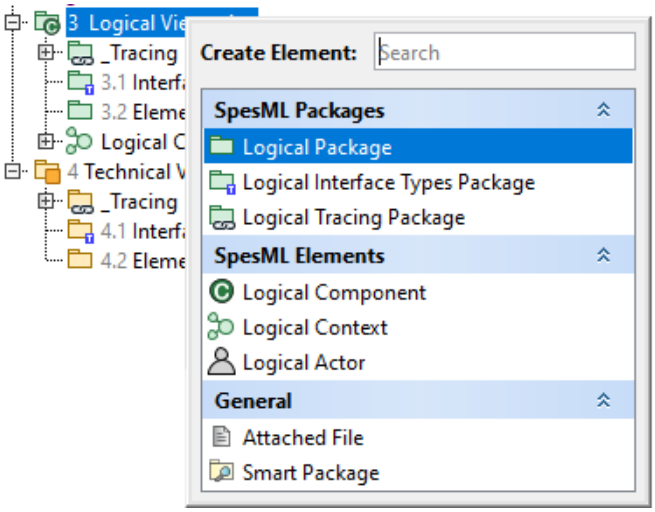


Figure 8.46: A creation view for creating logical viewpoint elements. Figure taken from [GJRR22b].

8.3.3 Discussion

The SPES methodology has been realised by modellers in the past but a reference implementation with a modelling tool and a concrete modelling language had been missing. Further, since the methodology describes different viewpoints, this separation of concerns are implemented in a way that the individual language components are developed in a modular, reusable way. By creating a UML and SysML based workbench for SPES in MagicDraw, we see the benefits of a good modelling experience by integrating language components, methods, and UXD aspects for practitioners. While this case study did not look particularly at user-centric recommendations, this was due to the fact that the realisation of the SpesML project was a rather PoC implementation and not yet developed for an actual business use case. The SPES artefacts were decomposed into smaller units, making the reuse and the extension of previously built language components and DSML building blocks more suitable. The SpesML project was carried out by a consortium of researchers and practitioners from a number of academic institutions and organisations, so the scope of the case study is limited to certain language components and UXD aspects for the implementation workbench in MagicDraw. By reusing many language

components and extending or adapting them to the necessities of the SPES methodology, language engineers could much more easily compose parts of the SPES implementation workbench such as requirements, functional aspects, behavioural logical contexts, and technical aspects such as modelling hardware or software elements. This case study, therefore, explored the systematic engineering of a DSML for a language-agnostic MBSE methodology and discussed the bridging of various heterogeneous domains by creating reusable units for fostering modularity in systems engineering.

Chapter 9

Conclusions

This thesis explored various techniques to foster language engineering primarily in the industrial context and to reduce the gap between the problem space and the solution space [FR07]. The results of the concepts presented in this thesis serve as a basis for engineering higher quality industrial DSMLs that encompasses methods to reuse parts of the language infrastructure and to consider design aspects for improving the modelling experience for practitioners and domain experts alike. Language engineers should be equipped with the state-of-the-art methods and techniques for developing DSMLs that promote the reusability of common language parts in the form of language components, describes exchange mechanisms for the interoperability of DSMLs and their parts, and to implement concepts of intelligently assisting practitioners directly in their respective modelling environments. To this end, research efforts must be undertaken continuously to improve methods for the language engineering processes for language engineers and to ameliorate the modelling experience of users. This chapter first summarises the main results of the thesis and how it answers the research questions presented in Chapter 1 in Section 9.1. Section 9.2 then discusses the potential for further work that arose during the course of this thesis.

9.1 Summary

This thesis presents a systematic approach for engineering modular and reusable DSMLs and its parts in an industrial context. Language components and their composition into heterogeneous DSMLs have been described in detail in this thesis along with methods and guidelines to improve the overall UX for domain modellers. The main results and the answers to the research questions are summarised in the following:

Systematic Engineering of Industrial DSMLs Chapter 3 presents a systematic approach to engineering industrial DSMLs using the concepts of reusable DSML building blocks. These DSML building blocks are individual units of a DSML that concern either a specific aspect of a single or a variety of domains and consist of language components, method support, and UX infrastructure. As industrial DSMLs often comprise intersecting concepts, providing a set of such DSML building blocks are reused across different modelling

projects to build heterogeneous DSMLs. The research question **RQ1** ("*How can language engineers systematically develop reusable building blocks for a language?*") is therefore sufficiently answered with the concepts described in Chapter 3 and through its applicability described via case studies in Chapter 8. The reuse of such DSML building blocks across different modelling projects help develop versions or families of languages without the need to rebuild common parts of a language from scratch. An overview of the activities further details the development and usage journey of industrial DSMLs (Figure 3.4) for language engineers, modellers, and domain experts, whose roles are described in Section 3.3. Further, individual parts of a DSML building block are described in Chapter 4 (language components), Chapter 6 (user experience), and Chapter 7 (methods).

Language Components Chapter 4 provides the definitions, requirements, and properties of language components in the form of software artefacts that eventually compose to form the central part of the individual DSML building blocks and subsequently heterogeneous DSMLs. Further, language composition techniques introduced in the same chapter detailed how versions or family of languages are developed using the concepts of language inheritance, extension, embedding, and aggregation in the technological spaces of MagicDraw and MontiCore. This chapter also discusses mutual notions of language components and their composition that is valid across the two technological spaces of textual and graphical DSMLs for providing a modular and reusable approach to language engineering. The definition and realisation of language components in MagicDraw as well as providing unified concepts of language components in both MagicDraw and MontiCore answers the research question **RQ2** ("*What constitutes a reusable language component?*"), whereas the different forms of language composition discussed in the chapter answers the research question **RQ3** ("*What different forms of language composition can be applied to foster the better development of heterogeneous DSMLs?*").

Interoperability of DSMLs Chapter 5 describes an exchange mechanism that allows language engineers to use the common constructs of a DSML across different modelling environments. To achieve this interoperability of DSML constructs, the chapter described the conceptual and technical implementation of interchanging language components from one modelling tool to another. In this thesis, MagicDraw and Enterprise Architect were the two modelling environments chosen to conduct the experiment for enabling the DSML exchange mechanism, as they are mature and commercial modelling tools with sufficient flexibility for providing customised solutions that complement the technical language definition. Individual add-ons to the respective modelling tools were created using GPL code in Java to achieve the import and export of UML constructs as well as domain-specific constructs using a common exchange file formatted in XML. The mechanism described in this thesis are, in a similar conceptual manner, extended to other modelling environments that provide the basic foundations for language devel-

opment, and even across the technological spaces, as was explored as part of mutual notions of language components in Chapter 4. The bidirectional exchange mechanism detailed in this thesis, therefore, sufficiently answers the research question **RQ4** ("*How can we achieve the interoperability of DSML constructs between multiple modelling environments?*"), and discusses the necessary concepts to further extend the mechanism to other modelling tools as well.

User Experience Chapter 6 presents the definitions, concepts, and guidelines for language engineers to develop DSMLs that enrich the modelling experience for practitioners. The reusability of language parts focusses on promoting a more modular approach to DSML development at the language level. However, there often is a lack of guidelines and design decisions that language engineers should consider to elevate the usability of DSMLs at model level. This thesis motivates why a good UX is necessary for modellers that fosters ease of use not just through visually appealing designs but also through organised and structured model views as well as with a strong focus on the cognitive aspects. The guidelines, design decisions, and usability heuristics presented in this thesis is further categorised into notions and functionalities based on industry standards that language engineers should consider. Generally, design decisions that are relevant to a particular domain improves the overall quality of a DSML, and is aligned with all the involved key stakeholders that must be considered as part of the complete language infrastructure. The presented (non-exhaustive) design guidelines therefore serve as a reference to language engineers in practically developing better DSMLs in the industry and answers the research question **RQ5** ("*How can language engineers develop better DSMLs that improves the overall modelling experience for users?*").

Model-Aware Recommendations Chapter 7 describes a methodology for developing a customisable recommendation tool directly on the modelling environment itself. The tool, built using a combination of configurable business rules, a NoSQL database for data storage, and GPL Java code for the implementation including the GUI, combines aspects of integrating a complete language infrastructure to assist modellers without the need to scour through endless pages of static sources of information such as handbooks or tutorials, that are often outdated as newer functionalities are introduced for a DSML. This thesis details a complete end-to-end solution for actively assisting modellers that provides recommendations, methods, and suggestions based on the current modelling scenario. To complement an answer to **RQ5**, such methods not only provides the necessary direction to modellers but also allows them to think about their models more actively. A guidance infrastructure that is tightly integrated with the technical definition of the language allows practitioners to further understand specific activities, tasks, or processes for their current models. As described in this thesis, language engineers easily configure and reconfigure rules without the need to constantly update a DSML it-

self, such that the overhead costs for deploying such an active and synchronised solution is significantly reduced. The recommendations that are made available through various datasets are easily updated as each individual DSML building block, that represents certain aspects of a single domain, are configured particularly with the help of domain experts. The development of such a model-aware recommendation framework for assisting practitioners in their modelling thus answers the research question **RQ6** ("*How can we establish a modelling methodology for providing integrated recommendations and guidance to modellers that considers their active modelling situation?*").

The individual chapters, the main results summarised in this chapter, and the case studies (Chapter 8) all contribute to the main research question and the partial research questions introduced in Chapter 1. To systematically engineer industrial DSMLs is a challenge, and this thesis presents concepts and methods that language engineers should consider to eventually develop better DSMLs that often comprise of intersecting and heterogeneous domain concepts in the industrial contexts.

9.2 Potential for Further Work

This thesis explores the systematic engineering of industrial DSMLs primarily in the MagicDraw ecosystem. The answers to the partial research questions introduced in Chapter 1 are realised through the use of the commercial modelling tool, MagicDraw. The engineering of DSMLs in the large is a huge effort and consist of a number of stakeholders in any industrial DSML project. To this extent, a number of further questions and alternative solutions arose during the course of this thesis.

Language Workbenches and Tools The concepts presented in this thesis primarily consider MagicDraw as the choice of tool for the realisation and implementation of the DSML building blocks and its parts. However, it must be noted that other graphical modelling tools such as MetaEdit+ [Tol06], Enterprise Architect [Ent23], and Rational Rhapsody [IBM23] also possess similar technical capabilities for language development. Therefore, further exploration of the concepts related to language components and DSML interoperability presented in this thesis with different commercial modelling tools provide additional advantages and disadvantages of the overall industrial DSML engineering process. Even though certain language workbenches such as MontiCore [HKR21], MPS [VV10], Spoofax [WKV14], and Melange [DCB+15] provide the necessary means for language composition and customisations, there is a need to further explore reusability of similar domain concepts. The presented forms of language composition can further be explored in other language workbenches that support language development such that the true modularisation of language components can be achieved independent of any language workbench or modelling tool.

Technological Spaces Defining language components that are valid in different technological spaces is challenging and this thesis attempts to provide a solution for developing such modular and reusable language components. In the textual space, MontiCore was explored as part of defining language components [Wor19], while MagicDraw was explored for the same in the graphical space. However, the concepts presented in this thesis detail primarily these two technological spaces, and do not represent completely the entire technological spectrum for modelling languages. For example, the projectional space [Cam14, RPKK20] is still relatively unexplored in respect to defining language composition techniques as well as for providing variants of products and product lines [SBW22]. While language composition has been applied with MPS [VV10], further investigation is required as to whether reusable language components can be tailored in a similar way as described in this thesis. Therefore, to provide a more holistic engineering approach to combining textual, graphical, and projectional spaces, further research is needed [EvdSV+13].

Design Considerations for High-Quality User Experience in DSMLs The design guidelines and design decisions, presented specifically in Chapter 6, are presented mainly for industrial graphical DSMLs. Such guidelines can also naturally apply to other kinds of graphical DSMLs, such as those in research. DSMLs are a specialised form of DSLs where each model is designed in a particular language with model transformation possibilities that potentially make the model valid in other modelling languages. Best practices for engineering DSL generally exist [BAG18, Voe09], but there is still a gap to the best practices for DSMLs that should be further studied [KKP+09]. Specific guidelines, such as defining a custom UI for generating ASTs from a textual grammar, could also be considered for non-graphical DSMLs but needs more research. UX is a vastly subjective topic, therefore providing a single source of truth for such guidelines is not easy. Further efforts must be made to close the gap and the understanding between researchers and practitioners to better represent the domains in consideration. The list of design decisions described in this thesis are certainly non-exhaustive, meaning further research can lead to additional design decisions and usability aspects that can be considered for a good UX and those that are independent of any modelling tool or language workbench.

Intelligent MDE Although the recommendation framework presented in this thesis can be considered part of artificial intelligence (AI) due to the broad definition of AI [BFG11], deeper knowledge, processes, and introducing machine learning (ML) techniques should be explored [PAC18]. This can ensure that active model-aware recommendations can not only consist of information from static sources such as handbooks or tutorials, but also be made available through the use of more intelligent systems such as by providing a more precise prediction based on historical domain data. For example, the use of OpenAI's ChatGPT [Ope23] can provide instantaneous answers to questions arising dur-

ing modelling, and newer techniques can also provide such recommendations by acting on behalf of the role specified by a practitioner. The role of ChatGPT in understanding, analysing, and improving the intelligence of the models in a system can possibly help language engineers understand better the various development processes in software modelling [CGR23]. Further work to provide actions directly within the presented guidance infrastructure is required that can effectively assign measurement values to models, define complex rules between models, and provide navigability between models and their recommendations. Further, making processes and their process models navigable in such a guidance infrastructure to provide fine tuned and deeper analysis methods based on the current modelling scenario of practitioners is also essential in fostering the usability of industrial DSMLs.

Bibliography

[ABC+17] Silvia Abrahão, Francis Bourdeleau, Betty Cheng, Sahar Kokaly, Richard Paige, Harald Stöerrle, and Jon Whittle. User experience for model-driven engineering: Challenges and future directions. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 229–236. IEEE, 2017. 2.2.6, 6.7

[ABM96] Amund Aarsten, Davide Brugali, and Giuseppe Menga. Patterns for three-tier client/server applications. *Proceedings of Pattern Languages of Programs (PLoP'96)*, 4(6), 1996. 7.3.1

[ABN+08] Aitor Aldazabal, Terry Baily, Felix Nanclares, Andrey Sadovykh, Christian Hein, and Tom Ritter. Automated model driven development processes. In *Proceedings of the ECMDA workshop on Model Driven Tool and Process Integration*, pages 361–375. Citeseer, 2008. 7.7

[AFR06] Daniel Amyot, Hanna Farah, and Jean-François Roy. Evaluation of development tools for domain-specific modeling languages. In *System Analysis and Modeling: Language Profiles: 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31-June 2, 2006, Revised Selected Papers 5*, pages 183–197. Springer, 2006. 3.7

[AGCDL21] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan De Lara. Recommender systems in model-driven engineering: A systematic mapping review. *Software and Systems Modeling*, pages 1–32, 2021. 2.2.7, 3.1.1

[AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017. 2.2.5

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. 2.2.3

[AM18] A Aleksandraviciene and A Morkevicius. MagicGrid® Book of Knowledge-A Practical Guide to Systems Modeling using MagicGrid from No Magic. *Inc, Allen Texas, USA*, 2018. 7.7

[AR08] Mohsen Asadi and Raman Ramsin. MDA-Based Methodologies: An Analytical Survey. In *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, pages 419–431, 2008. 2.2.5

[AR20] Henning Agt-Rickauer. *Supporting domain modeling with automated knowledge acquisition and modeling recommendations.* Technische Universitaet Berlin (Germany), 2020. 3.7

[ARKS19] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. Automated recommendation of related model elements for domain models. In *Model-Driven Engineering and Software Development: 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22-24, 2018, Revised Selected Papers 6*, pages 134–158. Springer, 2019. 2.2.7, 3.7, 4, 4

[ARVGMRMB09] David Alonso-Ríos, Ana Vázquez-García, Eduardo Mosqueira-Rey, and Vicente Moret-Bonillo. Usability: a critical analysis and a taxonomy. *International journal of human-computer interaction*, 26(1):53–74, 2009. 6.3.4

[AT05] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE transactions on knowledge and data engineering*, 17(6):734–749, 2005. 1.1, 2.2.7

[BAG18] Ankica Barišić, Vasco Amaral, and Miguel Goulão. Usability driven DSL development with USE-ME. *Computer Languages, Systems & Structures*, 51:118–157, 2018. 3, 6.7, 9.2

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005. 2.3.1

[BBC+01] Alan F Blackwell, Carol Britton, Anna Cox, Thomas RG Green, Corin Gurr, Gada Kadoda, Maria S Kutar, Martin Loomes, Chrystopher L Nehaniv, Marian Petre, et al. Cognitive dimensions of notations: Design tools for cognitive technology. In *International conference on cognitive technology*, pages 325–341. Springer, 2001. 6.3.3, 6.7

[BBC+10] Jean Bézivin, Hugo Brunelière, Jordi Cabot, Guillaume Doux, Frédéric Jouault, and Jean-Sébastien Sottet. Model driven tool interoperability in practice. In *3rd Workshop on Model-Driven Tool & Process Integration (co-located with ECMFA 2010)*, pages 62–72, 2010. 5.7

[BBK+21] Wolfgang Böhm, Manfred Broy, Cornel Klein, Klaus Pohl, Bernhard Rumpe, and Sebastian Schröck, editors. *Model-Based Engineering of Collaborative Embedded Systems*. Springer, January 2021. 1.4, 2.3.2, 7.5, 8.1.1, 8.3.1, 8.41, 9.2

[BBRC06] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006. 2.3.1

[BCC+10] Hugo Bruneliere, Jordi Cabot, Cauê Clasen, Frédéric Jouault, and Jean Bézivin. Towards model driven tool interoperability: Bridging eclipse and microsoft modeling tools. In *Modelling Foundations and Applications: 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings 6*, pages 32–47. Springer, 2010. 4, 5, 5.7

[BCCIM17] Marco Brambilla, Jordi Cabot, Javier Luis Cánovas Izquierdo, and Andrea Mauri. Better call the crowd: using crowdsourcing to shape the notation of domain-specific languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 129–138, 2017. 6.7

[BCCP21] Antonio Bucchiarone, Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. *Domain-specific Languages in Practice: With JetBrains MPS*. Springer Nature, 2021. 3.6

[BCH15] Nigel Bevan, James Carter, and Susan Harker. ISO 9241-11 revised: What have we learnt about usability since 1998? In *International conference on human-computer interaction*, pages 143–151. Springer, 2015. 2.2.6

[BCL+21] Antonio Bucchiarone, Federico Ciccozzi, Leen Lambers, Alfonso Pierantonio, Matthias Tichy, Massimo Tisi, Andreas Wortmann, and Vadim Zaytsev. What Is the Future of Modeling? *IEEE Software Journal*, 38(2):119–127, March-April 2021. 5.7

[BCOR15] Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Hélene Raynal. MDE in practice for computational science. *Procedia Computer Science*, 51:660–669, 2015. 3.7

[BD07] Matthias Bräuer and Birgit Demuth. Model-level integration of the OCL standard library using a pivot model with generics support. In *International Conference on Model Driven Engineering Languages and Systems*, pages 182–193. Springer, 2007. 5.7

[BDH94] M. Becker and J.L. Diaz-Herrera. Creating domain specific libraries: a methodology and design guidelines. In *Proceedings of 1994 3rd International Conference on Software Reuse*, pages 158–168, 1994. 3.7, 6.7

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *Journal ACM Transactions on Internet of Things*, 3:1–32, February 2022. 4.7

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology (JOT)*, 19(3):3:1–16, October 2020. 3.4.3, 3.7, 4.2.2, 4.3.1, 4.5, 4.5.1, 4.7, 4.8, 5.7

[BEK+18] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018. 2.3.1, 4

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software (JSS)*, 152:50–69, June 2019. 4.3.1, 4.5.2, 4.8, 6.2, 6.5

[Bet16] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016. 2.1, 1, 3.7, 6, 7

[Beu08] Danilo Beuche. Modeling and building software product lines with pure:: variants. In *Software Product Line Conference, International*, pages 358–358. IEEE Computer Society, 2008. 3.2, 6.5

[BFG11] Robin Burke, Alexander Felfernig, and Mehmet H Göker. Recommender systems: An overview. *AI Magazine*, 32(3):13–18, 2011. 9.2

[BGBV16] Kyle Banker, Douglas Garrett, Peter Bakkum, and Shaun Verch. *MongoDB in action: covers MongoDB version 3.0*. Simon and Schuster, 2016. 7.3.1

[BGdL10] Paolo Bottoni, Esther Guerra, and Juan de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Inf. Softw. Technol.*, 52(8):821–844, 2010. 4.8

[BGJ+23] Arvid Butting, Rohit Gupta, Nico Jansen, Nikolaus Regnat, and Bernhard Rumpe. Towards Modular Development of Reusable Language Components for Domain-Specific Modeling Languages in the MagicDraw and MontiCore Ecosystems. *Journal of Object Technology*, 22(1):1:1–21, September 2023. 7, 1.4, 2.1, 2.2.4, 4, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.4.1, 4.9, 4.10, 4.11, 4.12, 4.13, 9.2

[BGM10] Barrett R Bryant, Jeff Gray, and Marjan Mernik. Domain-Specific Software Engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 65–68. ACM, 2010. 1

[BGN+04] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *International journal on software tools for technology transfer*, 6:203–218, 2004. 3.1.1

[BGS05] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. Model bus: Towards the interoperability of modelling tools. In *Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004. Revised Selected Papers*, pages 17–32. Springer, 2005. 5.7

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017. 4.5.2

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *RTSE '97: Proceedings of the International Workshop on Requirements Targeting Software and Systems Engineering*, pages 43–68, London, UK, 1998. Springer-Verlag. 3.1.1, 4.7

[BHRW21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021. 3.6, 4.8

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018. 5.7

[BKK+23] Vincent Bertram, Hendrik Kausch, Evgeny Kusmenko, Haron Nqiri, Bernhard Rumpe, and Constantin Venhoff. Leveraging Natural Language Processing for a Consistency Checking Toolchain of Automotive Requirements. In Kurt Schneider, Fabiano Dalpiaz, and Jennifer Horkoff, editors, *IEEE 31st International Conference on Requirements Engineering (RE)*, pages 212–222. ACM/IEEE, September 2023. 7.2

[BLWPO13] F. P. Basso, C. M. Lima Werner, R. M. Pillat, and T. C. Oliveira. How do You Execute Reuse Tasks Among Tools? In *25th International Conference on Software Engineering and Knowledge Engineering*. Knowledge Systems Institute Graduate School, 2013. 3.7

[BMR22] Arvid Butting, Judith Michael, and Bernhard Rumpe. Language Composition via Kind-Typed Symbol Tables. *Journal of Object Technology (JOT)*, 21:4:1–13, October 2022. 4.5, 4.8, 5.7

[BN07] Scott Brave and Cliff Nass. Emotion in human-computer interaction. In *The human-computer interaction handbook*, pages 103–118. CRC Press, 2007. 6.3.1

[BNL21] Christoph Binder, Christian Neureiter, and Arndt Lüder. Towards a Domain-Specific Approach Enabling Tool-Supported Model-Based Systems Engineering of Complex Industrial Internet-of-Things Applications. *Systems*, 9(2), 2021. 5.3.1

[BNS21] Holger Stadel Borum, Henning Niss, and Peter Sestoft. On Designing Applied DSLs for Non-programming Experts in Evolving Domains. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 227–238. IEEE, 2021. 6.7

[BPRW20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020. 2.1, 4.8

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik Spektrum*, 30(1):3–18, 2007. 3.7, 4.8

[Bri96] Sjaak Brinkkemper. Method engineering: engineering of information systems development methods and tools. *Information and software technology*, 38(4):275–280, 1996. 1.1, 2.2.7

[BS12] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer Science & Business Media, 2012. 8.3.1

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information systems*, 35(6):615–636, 2010. 2.3.1

[BT75] Victor R Basil and Albert J Turner. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, (4):390–396, 1975. 3.4.7

[But23] Arvid Butting. *Systematic Composition of Language Components in MontiCore.* Aachener Informatik-Berichte, Software Engineering, Band 53. Shaker Verlag, February 2023. 1.2, 1.4, 2.1, 2.2.3, 2.2.4, 3.2.1, 3.4.3, 4.2.1, 4.2.2, 4.2.3, 4.3.1, 4.3.2, 4.5.2, 4.8

[BW21] Arvid Butting and Andreas Wortmann. Language Engineering for Heterogeneous Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 239–253. Springer, January 2021. 4, 4.2.2

[Cam14] F. Campagne. *The MPS Language Workbench: Volume I.* The MPS Language Workbench. 2014. 1, 3.7, 7, 9.2

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015. 2.1, 2.1, 2.2.7, 3.7, 4, 4.8, 5.7, 6, 7

[CBW17] Benoit Combemale, Olivier Barais, and Andreas Wortmann. Language engineering with the GEMOC studio. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 189–191. IEEE, 2017. 3.7

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe. On the Globalization of Domain Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 1–6. Springer, 2015. 2.2.5, 3.7, 4.8, 5.7

[CCG+08] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, Xavier Thirioux, and Francois Vernadat. A property-driven approach to formal verification of process models. In *Enterprise Information Systems: 9th International Conference, ICEIS 2007, Funchal, Madeira, June 12-16, 2007, Revised Selected Papers 9*, pages 286–300. Springer, 2008. 7.7

[CFB05] Ian T Cameron, Eric S Fraga, and IDL Bogle. Process modelling goals: concepts, structure and development. In *Computer Aided Chemical Engineering*, volume 20, pages 265–270. Elsevier, 2005. 3.3.3

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016. 2.1, 2.2.7, 3.2.1, 3.6

[CG11] Hyun Cho and Jeff Gray. Design patterns for metamodels. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 25–32, 2011. 3.7

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009. 2.1, 3.2.1, 3.7, 6, 7.1

[CGR23] Benoit Combemale, Jeff Gray, and Bernhard Rumpe. How to define modeling languages? *Journal Software and Systems Modeling (SoSyM)*, 22(2):449–451, April 2023. 9.2

[Cha15] Stephen J. Chapman. *MATLAB programming for engineers*. Nelson Education, 2015. 3

[CJKW07] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-specific development with visual studio dsl tools*. Pearson Education, 2007. 3.6

[CKM+18] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Journal Computer Languages, Systems & Structures*, 54:139 – 155, 2018. 1.1, 4

[CMP18] Gerald Czech, Michael Moser, and Josef Pichler. Best practices for domain-specific modeling. A systematic mapping study. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 137–145. IEEE, 2018. 6

[CMP20] Gerald Czech, Michael Moser, and Josef Pichler. A systematic mapping study on best practices for domain-specific modeling. *Softw. Qual. J.*, 28(2):663–692, 2020. 1.1

[CN02] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002. 2.3.1

[CR16] Noel Carroll and Ita Richardson. Aligning healthcare innovation and software requirements through design thinking. In *Proceedings of the international workshop on software engineering in healthcare systems*, pages 1–7, 2016. 3.5

[CRM16] Thaciana GO Cerqueira, Franklin Ramalho, and Leandro Balby Marinho. A Content-Based Approach for Recommending UML Sequence Diagrams. In *SEKE*, pages 644–649, 2016. 2.2.7

[CTVW19] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. Blended modelling-what, why and how. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 425–430. IEEE, 2019. 3.7

[DCB+15] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*, Pittsburgh, United States, 2015. 1.1, 2.1, 1, 2.1, 3.6, 3.7, 6, 7, 9.2

[DCL13] Papa Issa Diallo, Joël Champeau, and Loïc Lagadec. A model-driven approach to enhance tool interoperability using the theory of models of computation. In *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings 6*, pages 218–237. Springer, 2013. 5.7

[DJK+19] Manuela Dalibor, Nico Jansen, Johannes Kästle, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, and Andreas Wortmann. Mind the Gap: Lessons Learned from Translating Grammars Between MontiCore and Xtext. In Jeff Gray, Matti Rossi, Jonathan Sprinkle, and Juha-Pekka Tolvanen, editors, *International Workshop on Domain-Specific Modeling (DSM'19)*, pages 40–49. ACM, October 2019. 5.7

[DJM+19] Manuela Dalibor, Nico Jansen, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. Towards Sustainable Systems Engineering-Integrating Tools via Component and Connector Architectures. In Georg Jacobs and Jonas Marheineke, editors, *Antriebstechnisches Kolloquium 2019: Tagungsband zur Konferenz*, pages 121–133. Books on Demand, February 2019. 5.7

[DJR22] Florian Drux, Nico Jansen, and Bernhard Rumpe. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology (JOT)*, 21(4):4:1–13, October 2022. 2.2.4, 3.6, 4.2.1, 4.8

[DJRS22] Florian Drux, Nico Jansen, Bernhard Rumpe, and David Schmalzing. Embedding Textual Languages in MagicDraw. In *Modellierung 2022 Satellite Events*, pages 32–43. Gesellschaft für Informatik e.V., June 2022. 3.4.3, 4.8, 5.6, 5.7

[dLSPF16] André de Lima Salgado, Fabrício Horácio Sales Pereira, and André Pimenta Freire. User-Centred Design and Evaluation of Information Architecture for Information Systems. In *Handbook of Research on Information Architecture and Management in Modern Organizations*, pages 219–236. IGI Global, 2016. 6.3.2

[dOCCFD22] Raquel Araújo de Oliveira, Mario Cortes-Cornax, Agnès Front, and Alexandre Demeure. A low-code approach to support method engineering. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 793–797, 2022. 2.2.7

[EB04] Maged Elaasar and Lionel Briand. An overview of UML consistency management. *Carleton University, Canada, Technical Report SCE-04-18*, 2004. 4.5.3

[EGR12] 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, 2012. 1.2, 2.2.4, 4.3.1, 4.3.2, 4.5, 4.8, 5.7

[EKVB+08] Mustafa Suphi Erden, Hitoshi Komoto, Thom J Van Beek, Valentina D'Amelio, Erika Echavarria, and Tetsuo Tomiyama. A review of function modeling: Approaches and applications. *Ai Edam*, 22(2):147–169, 2008. 2.3.2, 2.3.2

[Ent23] Enterprise Architect, 2023. 1.1, 1.3, 2.2.1, 3.6, 4.8, 5, 9.2

[EV06] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006. 4.8

[EvdB10] Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science*, 253(7):105–120, 2010. 4.8

[EvdSV+13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. The State of the Art in Language Workbenches. In *Software Language Engineering*, volume 8225 of *LNCS*. Springer International Publishing, 2013. 1.1, 3.7, 4.8, 5.7, 7.3.2, 9.2

[EVDSV+15] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015. 5.7

[Fey16] Richard Feynman. Ebnf: A notation to describe syntax. *Cited on*, page 10, 2016. 2.2.3

[FGLP10] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical language analysis in software linguistics. In *International Conference on Software Language Engineering*, pages 316–326. Springer, 2010. 1.1, 2.1, 2.2.7, 3.7

[FGLP11] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical language analysis in software linguistics. In *Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers 3*, pages 316–326. Springer, 2011. 3.1.1

[FMS14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2014. 3

[For13] Charles Forsythe. *Instant FreeMarker Starter*. Packt Publishing Ltd, 2013. 2.2.3

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. 1, 2.1, 2.2.3, 3, 3.1, 7.4.2, 7.7

[FR05] Robert France and Bernhard Rumpe. Domain specific modeling. *Journal Software and Systems Modeling (SoSyM)*, 4(1):1–3, 2005. 3.1.1, 3.3.2

[FR07] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering 2007*, 2007. 1, 2.1, 2.2.1, 3, 6, 7, 9

[Fra10] Ulrich Frank. Outline of a method for designing domain-specific modelling languages. Technical report, ICB-research report, 2010. 5, 5

[Fra13] Ulrich Frank. Domain-specific modeling languages: requirements analysis and design guidelines. *Domain engineering: Product lines, languages, and conceptual models*, pages 133–157, 2013. 3.2.1, 3.7, 6.7

[FRR09] Florian Fieber, Nikolaus Regnat, and Bernhard Rumpe. Assessing usability of model driven development in industrial projects. In T. Bailey, R. Vogel, and J. Mansell, editors, *4th European Workshop on "From code centric to model centric software engineering: Practices, Implications and ROI" (C2M)*, University of Twente, NL-Enschede, June 24 2009. CTIT Workshop Proceedings, Enschede. 1, 2.2.1

[Gar10] Jesse James Garrett. *The elements of user experience: user-centered design for the web and beyond*. Pearson Education, 2010. 6.1

[GBD+16] 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, 2016. 3.7

[GBJ+24] Rohit Gupta, Christoph Binder, Nico Jansen, Ambra Calà, Jan Vollmar, Nikolaus Regnat, David Schmalzing, and Bernhard Rumpe. Towards Enabling Domain-Specific Modeling Language Exchange Between Modeling Tools. In *Advances in Model and Data Engineering in the Digitalization Era*, pages 89–103, Cham, March 2024. Springer Nature Switzerland. 7, 1.4, 2.2.5, 5, 5.2, 5.4, 5.5, 9.2

[GFC+08] 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 the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 791–794, 2008. 3, 3.6

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017. 4.2.2

[GJRR22a] Rohit Gupta, Nico Jansen, Nikolaus Regnat, and Bernhard Rumpe. Design Guidelines for Improving User Experience in Industrial Domain-Specific Modelling Languages. In *Proceedings of MODELS 2022. Companion*, pages 737–748. ACM, October 2022. 7, 1.4, 2.2.6, 3.2.1, 6, 6.1, 6.1, 6.1, 6.24, 6.25, 6.26, 8.28, 9.2

[GJRR22b] Rohit Gupta, Nico Jansen, Nikolaus Regnat, and Bernhard Rumpe. Implementation of the SpesML Workbench in MagicDraw. In *Modellierung 2022 Satellite Events*, pages 61–76. Gesellschaft für Informatik, June 2022. 7, 1.4, 3.2, 3.4.3, 4.3.2, 4.7, 4.8, 8.1.1, 8.41, 8.3.1, 8.42, 8.43, 8.44, 8.45, 8.46, 9.2

[GJRR23] Rohit Gupta, Nico Jansen, Nikolaus Regnat, and Bernhard Rumpe. User-Centric Model-Aware Recommendations for Industrial Domain-Specific Modelling Languages. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*

Companion (MODELS-C), pages 330–341. ACM/IEEE, IEEE, October 2023. 7, 1.4, 7, 7.1, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.9, 9.2, 9.2

[GKR+21] Rohit Gupta, Sieglinde Kranz, Nikolaus Regnat, Bernhard Rumpe, and Andreas Wortmann. Towards a Systematic Engineering of Industrial Domain-Specific Languages. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SE&IP)*, pages 49–56. IEEE, May 2021. 7, 1.4, 2.1, 2.2.4, 3, 3.2, 3.1, 3.3.3, 3.4, 4, 4.7, 4.8, 6, 6.3.3, 7.1, 9.2

[GLB+86] Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, and Charles Rich. Kestrel Institute: REPORT ON A KNOWLEDGE-BASED SOFTWARE ASSISTANT. In *Readings in Artificial Intelligence and Software Engineering*, pages 377–428. Elsevier, 1986. 7.7

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 34–43, Ottawa, Canada, 2015. ACM New York/IEEE Computer Society. 5.7

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011. 5.7

[Gre89] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989. 6.7

[Gro09] Richard C Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education, 2009. 3.6

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In *FMOODS/FORTE*, pages 152–166, 2009. 5.7

[GRV09] Ashok K Goel, Spencer Rugaber, and Swaroop Vattam. Structure, behavior, and function of complex systems: The structure, behavior, and function modeling language. *Ai Edam*, 23(1):23–35, 2009. 2.3.2

[H+06] Matthew Hause et al. The SysML modelling language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12, 2006. 8.3.2

[Has08] Marc Hassenzahl. User Experience (UX): Towards an Experiential Perspective on Product Quality. In *Proceedings of the 20th Conference on l'Interaction Homme-Machine*, IHM '08, page 11–15, New York, NY, USA, 2008. Association for Computing Machinery. 1.1, 2.2.6, 6

[HGMB13] José R Hoyos, Jesús García-Molina, and Juan A Botía. A domain-specific language for context modeling in context-aware systems. *Journal of Systems and Software*, 86(11):2890–2905, 2013. 7.7

[HHFN13] Shuhei Hiya, Kenji Hisazumi, Akira Fukuda, and Tsuneo Nakanishi. clooca: Web based tool for Domain Specific Modeling. In *MoDELS (Demos/Posters/StudentResearch)*, pages 31–35, 2013. 1.1

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 114–129. Springer, 2009. 4.8

[HJK+23] Malte Heithoff, Nico Jansen, Jörg Christian Kirchhof, Judith Michael, Florian Rademacher, and Bernhard Rumpe. Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2023, page 194–207, Cascais, Portugal, October 2023. Association for Computing Machinery. 3.2

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009. 3.7, 4.8

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021. 1.1, 1.2, 1.3, 1, 2.2.3, 3.4.3, 3.7, 4, 4.2, 4.2.3, 4.2.3, 4.3.2, 4.5.2, 4.7, 4.8, 9.2

[HLMSN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science*, pages 45–66. Springer, 2015. 4.3.1, 4.8

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology (JOT)*, 18(1):1–60, July 2019. 1

[Hon13] Benjamin Honke. *Situational Method Engineering for the Enactment of Method-Centric Domain-Specific Languages.* doctoralthesis, Universität Augsburg, 2013. 2.2.7, 3.2.1, 2, 2, 7.7

[HR04] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10), 2004. 2.1, 1, 3, 3.7, 4

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017.* Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017. 2.2.3, 3.7, 5.7

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, pages 181–192, 2012. 4.7

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015. 6.3.3

[HRW16] Robert Heim, Bernhard Rumpe, and Andreas Wortmann. Extending Architecture Description Languages With Exchangeable Component Behavior Languages. In *Conference on Software Engineering & Knowledge Engineering (SEKE'16)*, pages 1–6. KSI Research Inc., Fredericton, Canada, June 2016. 2.2.5

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures*, 54:386–405, 2018. 1.1, 3.1.1, 3.3.1, 3.7, 4.8, 5.7

[HSBAC17] Mariem Haoues, Asma Sellami, Hanêne Ben-Abdallah, and Laila Cheikhi. A guideline for software architecture selection based on ISO 25010 quality related characteristics. *International Journal of System Assurance Engineering and Management*, 8:886–909, 2017. 8.1.2, 8.2.2

[HSS17] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. Reusable and generic design decisions for developing UML-based domain-specific languages. *Information and Software Technology*, 92:49–74, 2017. 3.7

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth international conference on software reuse (Cat. No. 98TB100203)*, pages 134–142. IEEE, 1998. 3.7

[IBM23] IBM Rhapsody, 2023. 2.2.1, 4.8, 5, 9.2

[IC16] Javier Luis Cánovas Izquierdo and Jordi Cabot. Collaboro: a collaborative (meta) modeling tool. *PeerJ Computer Science*, 2:e84, 2016. 6.7

[ICLF+13] Javier Luis Cánovas Izquierdo, Jordi Cabot, Jesús J López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan De Lara. Engaging end-users in the collaborative development of domain-specific modelling languages. In *Cooperative Design, Visualization, and Engineering: 10th International Conference, CDVE 2013, Alcudia, Mallorca, Spain, September 22-25, 2013. Proceedings 10*, pages 101–110. Springer, 2013. 3.7

[ISO10] Ergonomics of human system interaction - part 210: Human-centred design for interactive systems. Standard, International Organization for Standardization, Geneva, Switzerland, March 2010. 2.2.6, 6.1, 6.7

[ISO11] ISO/IEC/IEEE. Systems and software engineering–architecture description. *ISO/IEC/IEEE 42010: 2011 (E)(Revision of ISO/IEC 42010: 2007 and IEEE Std 1471-2000)*, 2011:1–46, 2011. 8.3.1

[ISO16] Ergonomics of human-system interaction — Part 161: Guidance on visual user-interface elements. Standard, International Organization for Standardization, Geneva, Switzerland, February 2016. 2.2.6

[Jav23] Java™ Platform, Standard Edition 8 API Specification, 2023. 7.4.1

[JIMK] Timo Jokela, Netta Iivari, Juha Matero, and Minna Karukka. The Standard of User-Centered Design and the Standard Definition of Usability: Analyzing ISO 13407 against ISO 9241-11, year = 2003. In *Proceedings of the Latin American Conference on Human-Computer Interaction*, CLIHC '03, page 53–60, New York, NY, USA. Association for Computing Machinery. 2.2.6

[Jos06] Simon Josefsson. The base16, base32, and base64 data encodings. Technical report, 2006. 7.4.3

[KBC+19] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, 18:2361–2397, 2019. 5.7

[KBM16] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016. 3.7

[Ker14] Heiko Kern. Study of interoperability between meta-modeling tools. In *2014 Federated Conference on Computer Science and Information Systems*, pages 1629–1637. IEEE, 2014. 5.7

[Kha09] Lena Khaled. A comparison between UML tools. In *2009 second international conference on environmental and computer science*, pages 111–114. IEEE, 2009. 4.8

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009. 1.1, 3, 3.7, 6, 6.7, 9.2

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *Journal of Object Technology (JOT)*, 18(2):1–20, July 2019. 4.7

[KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *Journal ACM Transactions on Internet of Things*, 3:1–30, November 2022. 7.7

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels*. Pearson Education, 2008. 2.1, 2.1, 3.7

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020. 4.7

[KMS+09] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Systematic transformation development. *Electronic Communications of the EASST*, 21, 2009. 6.3.3

[Koe07] Dierk Koenig. *Groovy in action*. 2007. 3.4.3

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017. 4.7

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006*, pages 150–158, Finland, 2006. University of Jyväskylä. 3.3, 3.7, 4.8

[KSK+19] Pushpendu Kar, Arish Shareef, Arun Kumar, Koh Tsyr Harn, Balaji Kalluri, and Sanjib Kumar Panda. ReViCEE: A recommendation based approach for personalized control, visual comfort & energy efficiency in buildings. *Building and Environment*, 152:135–144, 2019. 7.5

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008. 1

[KTK09] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*, 2009. 6.7

[KV10] Lennart CL Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 444–463, 2010. 4.8

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003. 2.2.5

[KWJM02] Holger M Kienle, Anke Weber, Jens Jahnke, and Hausi A Müller. Tackling the adoption problem of domain-specific visual languages. In *2nd Domain-Specific Modeling Languages Workshop at OOPSLA 2002*, pages 77–88, 2002. 3.6

[LEM02] WenQian Liu, Steve Easterbrook, and John Mylopoulos. Rule-based detection of inconsistency in UML models. In *Workshop on Consistency Problems in UML-Based Software Development*, volume 5, 2002. 4.5.3

[LEN+14] Mohammed Lethrech, Issam Elmagrouni, Mahmoud Nassar, Abdelaziz Kriouile, and Adil Kenzi. Domain Specific Modeling approach for context-aware service oriented systems. In *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, pages 575–581. IEEE, 2014. 7.7

[LFGGdL16] Jesús J López-Fernández, Antonio Garmendia, Esther Guerra, and Juan de Lara. Example-based generation of graphical modelling environments. In *Modelling Foundations and Applications: 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings 12*, pages 101–117. Springer, 2016. 2.2.1

[LGC14] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and How to Use Multilevel Modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2), dec 2014. 4.8, 5.7

[LJJ07] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. DSL classification. In *OOPSLA 7th workshop on domain specific modeling*, 2007. 1.1

[LMT+18] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17:91–113, 2018. 3.1.1

[LNH06] Daniel Leroux, Martin Nally, and Kenneth Hussey. Rational Software Architect: A tool for domain-specific modeling. *IBM systems journal*, 45(3):555–568, 2006. 1.1

[Mag20] Magicdraw: A software and system visual modelling tool, 2020. 1.1, 1.2, 1.3, 2.2.1, 2.2.2, 3.4.2, 3.6, 4, 4.7, 5

[MAGD+16] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. Leveraging software product lines engineering in the development of external dsls: A systematic literature review. *Computer Languages, Systems & Structures*, 46:206–235, 2016. 2.1, 4, 4.4.1

[MC08] Jennifer Munnelly and Siobhán Clarke. A domain-specific language for ubiquitous healthcare. In *2008 Third International Conference on Pervasive Computing and Applications*, volume 2, pages 757–762. IEEE, 2008. 3.2

[MCGDL12] Bart Meyers, Antonio Cicchetti, Esther Guerra, and Juan De Lara. Composing textual modelling languages in practice. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pages 31–36, 2012. 4.8, 5.7

[MDLV12] Sadaf Mustafiz, Joachim Denil, Levi Lúcio, and Hans Vangheluwe. The FTG+PM Framework for Multi-Paradigm Modelling: An Automotive Case Study. In *Proceedings of the 6th International Workshop on Multi-paradigm Modeling*, pages 13–18, 2012. 5.7

[Mer10] Bernhard Merkle. Textual modeling tools: overview and comparison of language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 139–148, 2010. 5.7

[Met] MetaCase website `http://www.metacase.com`. 1.1, 2.2.1

[MGD+16] David Méndez-Acuña, José Angel Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. *Comput. Lang. Syst. Struct.*, 46:206–235, 2016. 1.1, 2.2.4, 3.7

[MHS05] 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. 3.7, 4.8

[MKHdK22] Qin Ma, Monika Kaczmarek-Heß, and Sybren de Kinderen. Validation and verification in domain-specific modeling method engineering: an integrated life-cycle view. *Software and Systems Modeling*, pages 1–21, 2022. 2.2.7

[Mod23a] Model Interchange Wiki, 2023. 5.7

[Mod23b] Model Driven Generation (MDG) Technologies, 2023. 5.1.1

[Mod23c] Modelio: Open Source UML and BPMN modeling tool, 2023. 2.2.1

[Moo09] Daniel Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Trans. Softw. Eng.*, 35(6):756–779, November 2009. 6, 6.3.1, 6.3.2

[MPHH10] Peter Membrey, Eelco Plugge, Tim Hawkins, and DUPTim Hawkins. *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Springer, 2010. 7.3.1

[MRAR20] Eduardo Mosqueira-Rey and David Alonso-Ríos. Usability heuristics for domain-specific languages (DSLs). In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1340–1343, 2020. 6.3.4, 6.7

[MRR11] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011. 5.7

[MSA+15] Salome Maro, Jan-Philipp Steghöfer, Anthony Anjorin, Matthias Tichy, and Lars Gelin. On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 1–12, 2015. 4.8

[MvdSC+18] Josh GM Mengerink, Bram van der Sanden, Bram CM Cappers, Alexander Serebrenik, Ramon RH Schiffelers, and Mark GJ van den Brand. Exploring DSL evolutionary patterns in practice: a study of DSL evolution in a large-scale industrial DSL repository. In *6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018*, pages 446–453. SciTePress Digital Library, 2018. 3

[MWCS11] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST*, 42, January 2011. 3

[Neu06] David Neuendorf. Review of MagicDraw UML® 11.5 Professional Edition. *J. Object Technol.*, 5(7):115–118, 2006. 3.4.1, 4.4.2

[NFT+10] Makoto Nakatsuji, Yasuhiro Fujiwara, Akimichi Tanaka, Tadasu Uchiyama, and Toru Ishida. Recommendations over domain specific user graphs. In *ECAI 2010*, pages 607–612. IOS Press, 2010. 7.1, 7.7

[Nie94] Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 152–158, 1994. 6.3.3

[Nie00] Jakob Nielsen. Designing web usability. 2000. 2.2.6

[NKBK12] Stefan Nadschläger, Hilda Kosorus, Andreas Boegl, and Josef Kueng. Content-based recommendations within a QA system using the hierarchical structure of a domain-specific taxonomy. In *2012 23rd International Workshop on Database and Expert Systems Applications*, pages 88–92. IEEE, 2012. 3.7, 7.7

[NR08] Ali Niknafs and Raman Ramsin. Computer-aided method engineering: an analysis of existing environments. In *Advanced Information Systems Engineering: 20th International Conference, CAiSE 2008 Montpellier, France, June 16-20, 2008 Proceedings 20*, pages 525–540. Springer Berlin Heidelberg, 2008. 2.2.7, 7.7

[Ope23] R OpenAI. GPT-4 technical report. *arXiv*, pages 2303–08774, 2023. 9.2

[OWH+21] Kofoworola Adebowale Odukoya, Robert Ian Whitfield, Laura Hay, Neil Harrison, and Malcolm Robb. An architectural description for the application of MBSE in complex systems. In *2021 IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–8. IEEE, 2021. 5

[Ozk19] Mert Ozkaya. Are the UML modelling tools powerful enough for practitioners? A literature review. *IET Software*, 13(5):338–354, 2019. 4.8, 5.7

[PAC18] Ivens Portugal, Paulo Alencar, and Donald Cowan. The use of machine learning algorithms in recommender systems: A systematic review. *Expert Systems with Applications*, 97:205–227, 2018. 9.2

[PB19] Henderik A. Proper and Marija Bjekovic. Fundamental challenges in systems modelling. *EMISA Forum*, 39(1):13–28, 2019. 6, 7

[PB20] Henderik A. Proper and Marija Bjeković. Fundamental challenges in systems modelling. In Heinrich C. Mayr, Stefanie Rinderle-Ma, and Stefan Strecker, editors, *40 Years EMISA 2019*, pages 13–28, Bonn, 2020. Gesellschaft für Informatik e.V. 1, 3, 3.1

[PBDH16] Klaus Pohl, Manfred Broy, Heinrich Daembkes, and Harald Hönninger. Advanced model-based engineering of embedded systems. In *Advanced Model-Based Engineering of Embedded Systems*, pages 3–9. Springer, 2016. 8.3.1

[PBVDL05] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer, 2005. 2.3.1, 3.2

[PHAB12] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy. *Model-based engineering of embedded systems: The SPES 2020 methodology*. Springer, 2012. 8.3.1

[PK15] Samad Paydar and Mohsen Kahani. A semantic web enabled approach to reuse functional requirements models in web engineering. *Automated Software Engineering*, 22:241–288, 2015. 2.2.7

[PPZ+21] Ildevana Poltronieri, Allan Christopher Pedroso, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. Is usability evaluation of DSL still a trending topic? In *Human-Computer Interaction. Theory, Methods and Tools: Thematic Area, HCI 2021, Held as Part of the 23rd HCI International Conference, HCII 2021, Virtual Event, July 24–29, 2021, Proceedings, Part I 23*, pages 299–317. Springer, 2021. 3.7

[PRBCZ17] 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. 2.2.6, 6.3.4, 6.7

[PRS+] Jakob Pietron, Alexander Raschke, Michael Stegmaier, Matthias Tichy, and Enrico Rukzio. A study design template for identifying usability issues in graphical modeling tools. 3.1.1

[PRS+16] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *Proceedings of the 25th international conference on World Wide Web*, pages 263–273, 2016. 7.3.1

[PZBdBC18] Ildevana Poltronieri, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. Usa-dsl: usability evaluation framework for domain-specific languages. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 2013–2021, 2018. 6.3.4, 6.7

[Reg18] Nikolaus Regnat. Why SysML does often fail - and possible solutions. In *Modellierung 2018, 21.-23. Februar 2018, Braunschweig, Germany*, pages 17–20, 2018. 6, 7.1

[RG02] Mark Richters and Martin Gogolla. OCL: Syntax, semantics, and tools. In *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002. 3.4.3

[Roq16] Pascal Roques. MBSE with the ARCADIA Method and the Capella Tool. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016. 3.2.1, 7.1

[RPKK20] Dennis Reuling, Christopher Pietsch, Udo Kelter, and Timo Kehrer. Towards projectional editing for model-based SPLs. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–10, 2020. 9.2

[Rum13] Bernhard Rumpe. Towards Model and Language Composition. In Benoit Combemale, Walter Cazzola, and Robert Bertrand France, editors, *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013. 4.3.2, 4.5

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016. 2.1, 2.2.5, 3.2.1, 5.7

[RW11] Bernhard Rumpe and Ingo Weisemöller. A Domain Specific Transformation Language. In Bernhard Schätz, D. Deridder, A. Pierantonio, Jonathan Sprinkle, and D. Tamzalit, editors, *ME 2011 - Models and Evolution*, Wellington, New Zealand, October 2011. 3.3.2

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018. 3.7, 4.8

[RWC11] Dan Rubel, Jaime Wren, and Eric Clayberg. *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 2011. 7.3.2

[RWZ09] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE software*, 27(4):80–86, 2009. 2.2.7

[SBW22] Johannes Schröpfer, Thomas Buchmann, and Bernhard Westfechtel. Projectional Editing of Software Product Lines Using Multi-variant Model Editors. *SN Computer Science*, 4(1):35, 2022. 9.2

[SCGL11] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Generic model transformations: write once, reuse everywhere. In *International Conference on Theory and Practice of Model Transformations*, pages 62–77. Springer, 2011. 4.8, 5.7

[Sch08] Markus Scheidgen. Textual Modelling Embedded into Graphical Modelling. In *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, pages 153–168, 2008. 2.2.5, 4.8

[Sch16] Hendrik Scholta. Similarity of activities in process models: Towards a metric for domain-specific business process modeling languages. 2016. 2.2.7

[Sei14] Ed Seidewitz. UML with meaning: executable modeling in foundational UML and the Alf action language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, pages 61–68, 2014. 4.8

[Sel03] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5), 2003. 1, 5

[Sip16] Sigurd Sippel. Domain-specific recommendation based on deep understanding of text. *Informatik 2016*, 2016. 7.7

[SJ07] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007. 2.1

[SKOK17] Ranka Stanković, Cvetana Krstev, Ivan Obradović, and Olivera Kitanović. Improving document retrieval in large domain specific textual databases using lexical resources. In *Transactions on Computational Collective Intelligence XXVI*, pages 162–185. Springer, 2017. 7.7

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *ACM Sigplan Notices*, 21(11):38–45, 1986. 2.2.4

[Spe06] OMG Infrastructure Specification. Object management group. *Needham, MA, USA*, 2(2), 2006. 2.2.5

[Spe07] OMG Available Specification. OMG Unified Modeling Language (OMG UML), Superstructure, v2. 1.2. *Object Management Group*, 70, 2007. 4.1.2

[Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of systems and software*, 56(1):91–99, 2001. 4.8

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010. 4.8, 5.2

[SSZM19] Gernot Starke, Michael Simons, Stefan Zörner, and Ralf D Müller. *arc42 by Example: Software architecture documentation in practice*. Packt Publishing Ltd, 2019. 8.2.2, 8.38, 8.2.2, 8.2.3, 9.2

[Sta09] Miroslaw Staron. Transitioning from code-centric to model-driven industrial projects: empirical studies in industry and academia. In *Model-Driven Software Development: Integrating Quality Assurance*, pages 236–262. IGI Global, 2009. 2.2.1

[ŞvdBV18] Ana Maria Şutîi, Mark van den Brand, and Tom Verhoeff. Exploration of modularity and reusability of domain-specific languages: an expression DSL in metamod. *Computer Languages, Systems & Structures*, 51:48–70, 2018. 4, 4.8

[TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70. Springer, July 2021. 3.7, 4.8, 5.7

[TG15] Saurabh Tiwari and Atul Gupta. A systematic literature review of use case specifications research. *Information and Software Technology*, 67:128–158, 2015. 4.7

[Tha12] Bernhard Thalheim. Syntax, semantics and pragmatics of conceptual modelling. In *International Conference on Application of Natural Language to Information Systems*, pages 1–10. Springer, 2012. 3.2.1

[Tid10] Jenifer Tidwell. *Designing interfaces: Patterns for effective interaction design*. O'Reilly Media, Inc., 2010. 6.3

[TK05] Juha-Pekka Tolvanen and Steven Kelly. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, pages 198–209, 2005. 2.1, 3.1.1, 3.7, 4.4.1

[Tol06] Juha-Pekka Tolvanen. MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 690–691, 2006. 2.1, 3.6, 6, 9.2

[TR03] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+ defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93, 2003. 4.8

[UTY95] Yasushi Umeda, Tetsuo Tomiyama, and H Yoshikawa. FBS modeling: modeling scheme of function for conceptual design. In *Proc. of the 9th int. workshop on qualitative reasoning*, pages 271–8, 1995. 2.3.2

[Val10] Antonio Vallecillo. On the combination of domain specific modeling languages. In *European Conference on Modelling Foundations and Applications*, pages 305–320. Springer, 2010. 4.8

[Voe09] Markus Voelter. Best Practices for DSLs and Model-Driven Development. *Journal of Object Technology*, 8(6):79–102, 2009. 2.2.6, 3.6, 6, 9.2

[Voe10] Markus Voelter. Implementing feature variability for models and code with projectional language workbenches. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 41–48, 2010. 3.2.1, 4.8

[Voe11] Markus Voelter. Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 383–430. Springer, 2011. 4.8

[VV10] Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 301–304, 2010. 3.6, 4.3.1, 4.3.2, 4.8, 9.2, 9.2

[WBCW20] Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. Modeling Languages in Industry 4.0: an Extended System-

atic Mapping Study. *Software and Systems Modeling*, 19(1):67–94, January 2020. 2.2.1

[WDS+10] W Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F Siok. Recent catastrophic accidents: Investigating how software was responsible. In *2010 fourth international conference on secure software integration and reliability improvement*, pages 14–22. IEEE, 2010. 2.3.1

[WG09] Christoph Wienands and Michael Golm. Anatomy of a visual domain-specific language project in an industrial context. In *Model Driven Engineering Languages and Systems: 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings 12*, pages 453–467. Springer, 2009. 2.2.1

[WGN16] Georg Weichhart, Wided Guédria, and Yannick Naudet. Supporting interoperability in complex adaptive enterprise systems: A domain specific language approach. *Data & Knowledge Engineering*, 105:90–106, 2016. 5.7

[WHR+13] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *International Conference on Model Driven Engineering Languages and Systems*, pages 1–17. Springer, 2013. 6

[Wir96] Niklaus Wirth. *Compiler Construction (International Computer Science Series)*. Addison-Wesley Pub (Sd), 1996. 4.1.1

[WKV14] Guido H. Wachsmuth, Gabriël DP Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE Software*, 31(5):35–43, 2014. 3.6, 9.2

[Wor19] Andreas Wortmann. Towards Component-Based Development of Textual Domain-Specific Languages. In Luigi Lavazza, Herwig Mannaert, and Krishna Kavi, editors, *International Conference on Software Engineering Advances (ICSEA 2019)*, pages 68–73. IARIA XPS Press, November 2019. 9.2

[WXU16] Jun Wang, Junfu Xiang, and Kanji Uchino. Domain-specific recommendation by matching real authors to social media users. In *Advances in Web-Based Learning–ICWL 2016: 15th International Conference, Rome, Italy, October 26–29, 2016, Proceedings 15*, pages 246–252. Springer, 2016. 7.7

[XMI23] XML Metadata Interchange (XMI) Specification, 2023. 2.2.5, 5.3.2, 5.7

[XML23] Extensible Markup Language (XML), 2023. 2.2.5, 3.4.3, 5.7

[Zac03] John A Zachman. The zachman framework for enterprise architecture. *Primer for Enterprise Engineering and Manufacturing.[si]: Zachman International*, 2003. 5.1.1

[ZMVS16] František Zezulka, Petr Marcon, Ivo Vesely, and Ondrej Sajdl. Industry 4.0–An Introduction in the phenomenon. *IFAC-PapersOnLine*, 49(25):8–12, 2016. 5.3.1

[ZNG15] Una Ieva Zusane, Oksana Nikiforova, and Konstantins Gusarovs. Several issues on the model interchange between model-driven software development tools, 2015. 5.7

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011. 4.7

List of Figures

Listings

List of Tables

Related Work from the SE Group, RWTH Aachen

The following section gives an overview of related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR+19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04c]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question of how digital and organizational techniques help to cope with the physical distance of developers and [RRSW17] addresses how to teach agile modeling.

Modeling will increasingly be used in development projects if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKR+06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum11, Rum12] and [Rum16, Rum17], the UML/P, a variant of the UML especially designed for programming, refactoring, and evolution is defined.

The language workbench MontiCore [GKR+06, GKR+08, HKR21] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09], and refactoring in various modeling and programming languages [PR03]. To better understand the effect of an agile evolving design, we discuss the need for semantic differencing in [MRR10].

In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation. In [KMA+16] we have also introduced a classification of ways to reuse modeled software components.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, the size, and the number of artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17].

To keep track of relevant structures, artifacts, and their relations in order to be able, e.g., to evolve or adapt models and their implementing code, the *artifact model* [GHR17, Gre19] was introduced. [BGRW18] and [HJK+21] explain its applicability in systems engineering based on MDSE projects and [BHR+18] applies a variant of the artifact model to evolutionary development, especially for CPS.

An artifact model is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model, therefore, covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks.

For analysis of MLOps in an agile development, a software 2.0 artifact model distinguishing different kinds of artifacts is given in [AKK+21].

According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Such operators have been defined for Activity Diagrams [MRR11d], Class Diagrams [MRR11b], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18a].

We apply logic, knowledge representation, and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests, or find counterexamples using a theorem prover. We have defined a core theory in [BKR+20], which is based on the core concepts of Broy's Focus theory [RR11, BR07], and applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, KMP+21, HRR12].

Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams (ADs) and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11, KLPR12] and city quarters [GLPR15] to optimize operational efficiency and prevent unneeded CO_2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P, [Hab16] for MontiArc is used in domains such as cars or robotics

[HRR12], and [AMN+20a] for enterprise information systems based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HKR21].

In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKR+06, GHK+15, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12, HRW15, Hoe18], we demonstrate how to systematically derive a transformation language in concrete syntax and, e.g., in [HHR+15, AHRW17] we have applied this technique successfully for several UML sub-languages and DSLs.

[HNRW16] presents how to generate extensible and statically type-safe visitors. In [NRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLN+15, HLN+15a].

To understand the implications of executability for UML, we discuss the needs and the advantages of executable modeling with UML in agile projects in [Rum04c], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML) & the UML-P Tool

Starting with the early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] and is implemented in [Sch12].

Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09], [BCGR09a], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied when checking variants of class diagrams [MRR11e] and object diagrams [MRR11c] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11a] which allows us to check for semantic differences in activity diagrams [MRR11d]. The basic semantics for ADs and their semantic variation points are given in [GRR10].

We also discuss how to ensure and identify model quality [FHR08], how models, views, and the system under development correlate to each other [BGH+98b], and how to use modeling in agile development projects [Rum04c], [Rum03] and [Rum02].

The question of how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99a], [FEL+98] and [SRVK10].

The UML-P tool was conceptually defined in [Rum16, Rum17, Rum12, Rum11], got the first realization in [Sch12], and is extended in various ways, such as logically or physically distributed computation [BKRW17a]. Based on a detailed examination [JPR+22], insights are also transferred to the SysML 2.

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use than general-purpose programming languages but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HKR21] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HKR21] for easy development. New languages

and tools can be defined in modular forms [KRV08, GKR+07, Voe11, HLN+15, HLN+15a, HRW18, BEK+18b, BEK+19, Sch12] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages already in [KRV06] and elaborate on the engineering aspect of DSL development in [CFJ+16].

[Wei12, HRW15, Hoe18] present an approach that allows the creation of transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses, and model evolution have been discussed in [LRSS10] and [SRVK10]. [BJRW18] describes a mapping bridge between both. DSL quality in [FHR08], instructions for defining views [GHK+07] and [PFR02], guidelines to define DSLs [KKP+09], and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

A broader discussion on the *global* integration of DSMLs is given in [CBCR15] as part of [CCF+15a], and [TAB+21] discusses the compositionality of analysis techniques for models.

The MontiCore language workbench has been successfully applied to a larger number of domains, resulting in a variety of languages documented, e.g., in [AHRW17, BEH+20, BHR+21, BPR+20, HHR+15, HJRW20, HMR+19, HRR12, PBI+16, RRW15] and Ph.D. theses like [Ber10, Gre19, Hab16, Her19, Kus21, Loo17, Pin14, Plo18, Rei16, Rot17, Sch12, Wor16].

Software Language Engineering

For a systematic definition of languages using a composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15a]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HKR21, HRW18, BPR+20, BEK+19].

In [SRVK10] we discuss the possibilities and the challenges of using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Voe11, Naz17, KRV08, HLN+15, HLN+15a, HNRW16, HKR21, BEK+18b, BEK+19] and the backend [RRRW15b, NRR16, GMR+16, HKR21, BEK+18b, BBC+18]. In [GHK+15, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV10] describes the roles in software development using domain specific languages.

Language derivation is to our belief a promising technique to develop new languages for a specific purpose, e.g., model transformation, that relies on existing basic languages [HRW18].

How to automatically derive such a transformation language using a concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs.

We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15, HHK+13] that are derived from base languages to be able to constructively describe differences between model variants usable to build feature sets.

The derivation of internal DSLs from grammars is discussed in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services.

We use streams, statemachines, and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR10, HRR12] for architecture design and extensions for states [RRW13c, BKRW17a, RRW14a, Wor16]. In [RRW13], we introduce a code generation framework for MontiArc. [RRRW15b] describes how the language is composed of individual sublanguages.

MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [PR13], security in [HHR+15], and the robotics domain [AHRW17, AHRW17b]. Extension mechanisms for MontiArc are generally discussed in [BHH+17].

[GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants.

[MRR14b] provides a precise technique for verifying the consistency of architectural views [Rin14, MRR13] against a complete architecture to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. An experience report [MRRW16] and a methodological embedding [DGH+19] complete the core approach.

Extensions for co-evolution of architecture are discussed in [MMR10], for powerful analyses of software architecture behavior evolution provided in [BKRW19], techniques for understanding semantic differences presented in [BKRW17], and modeling techniques to describe dynamic architectures shown in [HRR98, HKR+16, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09, TAB+21] motivate the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HKR21] that can even be used to develop modeling tools in a compositional form [HKR21, HLN+15, HLN+15a, HNRW16, NRR16, HRW18, BEK+18b, BEK+19, BPR+20, KRV07b]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09].

[Voe11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15b] applies compositionality to robotics control.

[CBCR15] (published in [CCF+15a]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information, we have developed the concept of tagging languages in [GLRR15, MRRW16]. It allows the description of additional information for model elements in separated documents, facilitates reuse, and allows typing tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision, and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by

using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96, RK96]. An extended version especially suited for the UML is given in [GRR09], [BCGR09a] and in [BCGR09] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08] or sequence diagrams in [BGH+98a].

To better understand the effect of an evolved design, detection of semantic differencing, as opposed to pure syntactical differences, is needed [MRR10]. [MRR11d, MRR11a] encode a part of the semantics to handle semantic differences of activity diagrams. [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. And [BKRW17] compares component and connector architectures similar to SysML' block definition diagrams.

In [BR07, RR11], a precise mathematical model for distributed systems based on black-box behaviors of components is defined and accompanied by automata in [Rum96]. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of exemplary object interaction, today called sequence diagram. [BGH+98b] discusses the relationships between a system, a view, and a complete model in the context of the UML.

[GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these to class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10].

[Rum12] defines the semantics in a variety of code and test case generation, refactoring, and evolution techniques. [LRSS10] discusses the evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied to the development of CPS.

A first encoding of these theories in the Isabelle verification tool is defined in [BKR+20].

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code, they are not initially correct and need to be changed, evolved, and maintained over time. Model transformation is therefore essential to effectively deal with models [CFJ+16].

Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04c, MRR10], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11e, Rum12], systematic model transformation language development [Wei12, HRW15, Hoe18, HHR+15], repair of failed model evolution [KR18a].

[Rum04c] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97] and refining pipe-and-filter architectures is explained in [PR99]. This has e.g. been applied for robotics in [AHRW17, AHRW17b].

Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. [HRRS11, HRR+11, HRRS12] encode these in constructive Delta transformations, which are defined in derivable Delta languages [HHK+13].

Translation between languages, e.g., from class diagrams into Alloy [MRR11e] allows for comparing class diagrams on a semantic level. Similarly, semantic differences of evolved activity diagrams are identified via techniques from [MRR11d] and for Simulink models in [RSW+15].

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example, cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08, GKPR08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12].

Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRRS11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe special variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13, HHK+15] and [HRW15] describe an approach to systematically derive delta languages.

We also apply variability modeling languages to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for their compositional reuse [BEK+18b, BEK+19], and applied it as a semantic language refinement on Statecharts in [GR11].

Digital Twins and Digital Shadows in Engineering and Production

The digital transformation of production changes the life cycle of the design, the production, and the use of products [BDJ+22]. To support this transformation, we can use Digital Twins (DTs) and Digital Shadows (DSs). In [DMR+20] we define: "A digital twin of a system consists of a set of models of the system, a set of digital shadows, and provides a set of services to use the data and models purposefully with respect to the original system."

We have investigated how to synthesize self-adaptive DT architectures with model-driven methods [BBD+21a] and have applied it e.g. on a digital twin for injection molding [BDH+20]. In [BDR+21] we investigate the economic implications of digital twin services.

Digital twins also need user interaction and visualization, why we have extended the infrastructure by generating DT cockpits [DMR+20]. To support the DevOps approach in DT engineering, we have created a generator for low-code development platforms for digital twins [DHM+22] and sophisticated tool chains to generate process-aware digital twin cockpits that also include condensed forms of event logs [BMR+22].

[BBD+21b] describes a conceptual model for digital shadows covering the purpose, relevant assets, data, and metadata as well as connections to engineering models. These can be used during the runtime of a DT, e.g. when using process prediction services within DTs [BHK+21].

Integration challenges for digital twin systems-of-systems [MPRW22] include, e.g., the horizontal integration of digital twin parts, the composition of DTs for different perspectives, or how to handle different lifecycle representations of the original system.

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12, BBR20] are complex, distributed systems that control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied to

the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12, KRRW17], autonomous driving [BR12b, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07].

In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest to European avionics [ZPK+11]. Optimized [KRS+18a] and domain specific code generation [AHRW17b], and the extension to product lines of CPS [RSW+15, KRR+16, RRS+16] are key for CPS.

A component and connector architecture description language (ADL) suitable for the specific challenges in robotics is discussed in [RRW13c, RRW14a, Wor16, RRSW17, Wor21]. In [RRW12], we use this language for describing requirements and in [RRW13], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as an Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition of contributing to systems engineering in automotive [FND+98] and [GHK+08a], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration.

To facilitate the modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for production engineering based on these concepts [BKL+18] and addressed to bridge the gap between functions and the physical product architecture by modeling mechanical functional architectures in SysML [DRW+20]. For that purpose, we also did a detailed examination of the upcoming SysML 2.0 standard [JPR+22] and examined how to extend the SPES/CrEST methodology for a systems engineering approach [BBR20].

Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20], and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09a, BCGR09], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems.

In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07].

We apply these techniques, e.g., in MontiArcAutomaton [RRW13, RRW14a, RRW13, RW18], in a robot task modeling language [THR+13], and in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behavior (2) based on information from previously stored and real-time monitored structural context and behavior data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20, MRZ21] or UML/P based languages [MNRV19]. [MM15] describes a process of how languages for assistive systems can be created. MontiGem [AMN+20a] is used as the underlying generator technology.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18a] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view of the system design allows to track who does what, when, why, where, and how with personal data, makes information about it available via information services and provides support using assistive services.

Modeling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires the composition and the interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers the broad propagation of robotics applications.

The MontiArcAutomaton language [RRW12, RRW14a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13c, RRRW15b, HKR21] that perfectly fit robotic architectural modeling.

The iserveU modeling framework describes domains, actors, goals, and tasks of service robotics applications [ABH+16, ABH+17] with declarative models. Goals and tasks are translated into models of the planning domain definition language (PDDL) and then solved [ABK+17]. Thus, domain experts focus on describing the domain and its properties only.

The LightRocks [THR+13, BRS+15] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible with different robotics

middleware platforms.

Many of the concepts in robotics were derived from automotive software [BBR07, BR12b].

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment, and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed, and tested. A consistent requirement management connecting requirements with features in all development phases for the automotive domain is described in [GRJA12].

The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. A methodical embedding of the resulting function nets and their quality assurance using automated testing is given in the SMaRDT method [DGH+19, KMS+18].

[HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses the means to extract a well-defined Software Product Line from a set of copy and paste variants.

Potential variants of components in product lines can be identified using similarity analysis of interfaces [KRR+16], or execute tests to identify similar behavior [RRS+16]. [RSW+15] describes an approach to using model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we model dynamic reconfiguration of architectures applied to cooperating vehicles.

Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12b, BR12], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in the development and the evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12b].

[MMR10] gives an overview of the state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions.

MontiSim simulates autonomous and cooperative driving behavior [GKR+17] for testing various forms of errors as well as spatial distance [FIK+18, KKRZ19]. As tooling infrastructure, the SSELab storage, versioning, and management services [HKR12] are essential for many projects.

Internet of Things, Industry 4.0 & the MontiThings Tool

The Internet of Things (IoT) requires the development of increasingly complex distributed systems. The MontiThings ecosystem [KRS+22] provides an end-to-end solution to modeling, deploying [KKR+22], and analyzing [KMR21] failure-tolerant [KRS+22] IoT systems and connecting them to synthesized digital twins [KMR+20]. We have investigated how model-driven methods can support the development of privacy-aware [ELR+17, HHK+14] cloud systems [PR13], distributed systems security [HHR+15], privacy-aware process mining [MKM+19], and distributed robotics applications [RRRW15b].

In the course of Industry 4.0, we have also turned our attention to mechanical or electrical applications [DRW+20]. We identified the digital representation, integration, and (re-)configuration of automation systems as primary Industry 4.0 concerns [WCB17]. Using a multi-level modeling framework, we support machine as a service approaches [BKL+18].

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO_2 emissions are important challenges. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales.

During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for the technical specification of building services already.

We adapted the well-known concept of statemachines to be able to describe different states of a facility and validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing and Services

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, facilitate web-based innovations, and increase the efficiency and cost-effectiveness of web development [KRR14].

Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15a], Big Data, Apps, and Service Ecosystems bring attention to aspects like responsiveness, privacy, and open platforms. Regardless of the application domain, developers of such systems need robust methods and efficient, easy-to-use languages and tools [KRS12].

We tackle these challenges by perusing a model-based, generative approach [PR13]. At the core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale.

We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our development platforms. New services, e.g., for collecting data from temperature sensors, cars, etc. are now easily developed and deployed, e.g., in production or Internet-of-Things environments.

Security aspects and architectures of cloud services for the digital me in a privacy-aware environment are addressed in [ELR+17].

Model-Driven Engineering of Information Systems & the MontiGem Tool

Information Systems provide information to different user groups as the main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HKR21], we developed several generators for such data-centric information systems.

MontiGem [AMN+20a] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular, and incremental, handwritten and generated code pieces are well integrated [GHK+15a, NRR15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

We are using MontiGem for financial management [GHK+20, ANV+18], for creating digital twin cockpits [DMR+20], and various industrial projects. MontiGem makes it easier to create low-code development platforms for digital twins [DHM+22]. When using additional DSLs, we can develop assistive systems providing user support based on goal models [MRV20], privacy-preserving information systems using privacy models and purpose trees [MNRV19], and process-aware digital twin cockpits using BPMN models [BMR+22].

We have also developed an architecture of cloud services for the digital me in a privacy-aware environment [ELR+17] and a method for retrofitting generative aspects into existing applications [DGM+21].

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016. 9.2

[ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse.* Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017. 9.2

[ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017. 9.2

[AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017. 9.2, 9.2, 9.2, 9.2, 9.2

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017. 9.2, 9.2, 9.2, 9.2

[AKK+21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 55–68. ACM, October 2021. 9.2

[AMN+20a] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020. 9.2, 9.2, 9.2

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, CEUR Workshop Proceedings 2097, pages 75–79. CEUR-WS.org, May 2018. 9.2

[BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation

in Computational Neuroscience: A Review of Tools and Techniques. *Journal Frontiers in Neuroinformatics*, 12, 2018. 9.2

[BBD+21b] Fabian Becker, Pascal Bibow, Manuela Dalibor, Aymen Gannouni, Viviane Hahn, Christian Hopmann, Matthias Jarke, Istvan Koren, Moritz Kröger, Johannes Lipp, Judith Maibaum, Judith Michael, Bernhard Rumpe, Patrick Sapel, Niklas Schäfer, Georg J. Schmitz, Günther Schuh, and Andreas Wortmann. A Conceptual Model for Digital Shadows in Industry and its Application. In Aditya Ghose, Jennifer Horkoff, Vitor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling, ER 2021*, pages 271–281. Springer, October 2021. 9.2

[BBD+21a] Tim Bolender, Gereon Bürvenich, Manuela Dalibor, Bernhard Rumpe, and Andreas Wortmann. Self-Adaptive Manufacturing with Digital Twins. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 156–166. IEEE Computer Society, May 2021. 9.2

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007. 9.2, 9.2, 9.2

[BBR20] Manfred Broy, Wolfgang Böhm, and Bernhard Rumpe. Advanced Systems Engineering - Die Systeme der Zukunft. White paper, fortiss. Forschungsinstitut für softwareintensive Systeme, Munich, July 2020. 9.2, 9.2

[BCGR09] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009. 9.2, 9.2, 9.2

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009. 9.2, 9.2, 9.2

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007. 9.2, 9.2

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007. 9.2, 9.2, 9.2

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020. 9.2, 9.2, 9.2, 9.2

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *Journal ACM Transactions on Internet of Things*, 3:1–32, February 2022. 9.2

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018. 9.2, 9.2

[BDR+21] Christian Brecher, Manuela Dalibor, Bernhard Rumpe, Katrin Schilling, and Andreas Wortmann. An Ecosystem for Digital Shadows in Manufacturing. In *54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0*. Elsevier, September 2021. 9.2

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology (JOT)*, 19(3):3:1–16, October 2020. 9.2

[BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018. 9.2, 9.2, 9.2, 9.2

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software (JSS)*, 152:50–69, June 2019. 9.2, 9.2, 9.2, 9.2

[Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010. 9.2

[BGH+97] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, LNCS 1357. Springer Verlag, 1997. 9.2

[BGH+98a] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. *Journal Computer Standards & Interfaces*, 19(7):335–345, November 1998. 9.2

[BGH+98b] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998. 9.2, 9.2

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND'17) Workshop*, July 2017. 9.2

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018. 9.2

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017. 9.2

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017. 9.2

[BHK+21] Tobias Brockhoff, Malte Heithoff, István Koren, Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, Merih Seran Uysal, Wil M. P. van der Aalst, and Andreas Wortmann. Process Prediction with Digital Twins. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 182–187. ACM/IEEE, October 2021. 9.2

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998. 9.2, 9.2

[BHR+18] Arvid Butting, Steffen Hillemacher, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Shepherding Model Evolution in Model-Driven Development. In *Joint Proceedings of the Workshops at Modellierung 2018 (MOD-WS 2018)*, CEUR Workshop Proceedings 2060, pages 67–77. CEUR-WS.org, February 2018. 9.2

[BHR+21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021. 9.2

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018. 9.2, 9.2

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018. 9.2, 9.2

[BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle Theory for distributed, interactive systems - the untimed case*. Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020. 9.2, 9.2

[BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017. 9.2, 9.2

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017. 9.2, 9.2, 9.2

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software (JSS)*, 149:437–461, March 2019. 9.2, 9.2

[BMR+22] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)*, 70, June 2022. 9.2, 9.2

[BPR+20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020. 9.2, 9.2, 9.2

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007. 9.2, 9.2, 9.2, 9.2, 9.2

[BR12b] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012. 9.2, 9.2, 9.2

[BR12] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012. 9.2

[BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*, 2015. 9.2

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015. 9.2, 9.2, 9.2

[CCF+15a] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015. 9.2, 9.2, 9.2

[CEG+14] Betty H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, LNCS 8378, pages 101–136. Springer International Publishing, Switzerland, 2014. 9.2

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016. 9.2, 9.2

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008. 9.2, 9.2

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009. 9.2, 9.2

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019. 9.2

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018. 9.2, 9.2

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and

Andreas Wortmann. SMArDT modeling for automotive software testing. *Journal on Software: Practice and Experience*, 49(2):301–328, February 2019. 9.2, 9.2, 9.2, 9.2

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology (JOT)*, 20:1–24, November 2021. 9.2

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020. 9.2

[DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, 70, June 2022. 9.2, 9.2

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019. 9.2

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020. 9.2, 9.2, 9.2

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020. 9.2, 9.2

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999. 9.2

[EFLR99a] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. Â«UMLÂ»'98: Beyond the Notation*, LNCS 1618, pages 336–348. Springer, Germany, 1999. 9.2

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line

Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019. 9.2

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017. 9.2, 9.2, 9.2, 9.2

[FEL+98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Journal Computer Standards & Interfaces*, 19(7):325–334, November 1998. 9.2

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008. 9.2, 9.2, 9.2

[FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018. 9.2

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011. 9.2

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011. 9.2, 9.2

[FND+98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. BMW-ROOM An Object-Oriented Method for ASCET. In *SAE'98, Cobo Center (Detroit, Michigan, USA)*, Society of Automotive Engineers, 1998. 9.2

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012. 9.2, 9.2, 9.2

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007. 9.2, 9.2, 9.2

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008. 9.2, 9.2, 9.2

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008. 9.2

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015. 9.2, 9.2

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015. 9.2, 9.2, 9.2

[GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020. 9.2

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects.* Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017. 9.2

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008. 9.2, 9.2

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996. 9.2, 9.2

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0: Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006. 9.2, 9.2, 9.2

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007. 9.2

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008. 9.2, 9.2, 9.2

[GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017. 9.2

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015. 9.2

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015. 9.2, 9.2, 9.2

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020. 9.2

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016. 9.2, 9.2, 9.2, 9.2

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995. 9.2

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011. 9.2, 9.2, 9.2, 9.2

[Gre19] Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte.* Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019. 9.2, 9.2

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012. 9.2, 9.2, 9.2

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-based Definition of Modeling Language Semantics. In *Proc. of FMOODS/FORTE 2009, LNCS 5522*, Lisbon, Portugal, 2009. 9.2

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010. 9.2, 9.2

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems.* Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016. 9.2, 9.2

[Her19] Lars Hermerschmidt. *Agile Modellgetriebene Entwicklung von Software Security & Privacy.* Aachener Informatik-Berichte, Software Engineering, Band 41. Shaker Verlag, June 2019. 9.2

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013. 9.2, 9.2, 9.2

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014. 9.2, 9.2

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015. 9.2, 9.2

[HHK+15a] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Journal Future Generation Computer Systems*, 56:701–718, 2015. 9.2

[HHR+15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, CEUR Workshop Proceedings 1463, pages 18–23, 2015. 9.2, 9.2, 9.2, 9.2, 9.2

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021. 9.2

[HJRW20] Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In Dominik Bork, Dimitris Karagiannis,

and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 189–190. Gesellschaft für Informatik e.V., February 2020. 9.2

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013. 9.2, 9.2

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007. 9.2

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009. 9.2, 9.2

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011. 9.2

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012. 9.2, 9.2

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, LNCS 9839, pages 175–182. Springer, December 2016. 9.2

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021. 9.2, 9.2, 9.2, 9.2, 9.2, 9.2, 9.2

[HLN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015. 9.2, 9.2, 9.2, 9.2

[HLN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015. 9.2, 9.2, 9.2, 9.2

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology (JOT)*, 18(1):1–60, July 2019. 9.2, 9.2, 9.2

[HNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016. 9.2, 9.2, 9.2

[Hoe18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018. 9.2, 9.2, 9.2

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, 37(10):64–72, October 2004. 9.2

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017.* Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017. 9.2

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998. 9.2

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, Informatik-Bericht 2010-01, pages 13 – 22. fortiss GmbH, Germany, 2010. 9.2

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011. 9.2, 9.2, 9.2

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012. 9.2, 9.2, 9.2, 9.2, 9.2

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011. 9.2, 9.2, 9.2

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012. 9.2, 9.2, 9.2

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012. 9.2, 9.2

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015. 9.2, 9.2, 9.2, 9.2, 9.2

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures*, 54:386–405, 2018. 9.2, 9.2, 9.2

[JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology (JOT)*, 21:1–15, July 2022. 9.2, 9.2

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software Journal*, 35(6):40–47, 2018. 9.2

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag. 9.2

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009. 9.2, 9.2

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *Journal of Object Technology (JOT)*, 18(2):1–20, July 2019. 9.2, 9.2, 9.2

[KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *Journal ACM Transactions on Internet of Things*, 3:1–30, November 2022. 9.2

[KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019. 9.2

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012. 9.2, 9.2, 9.2

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR'16)*, LNCS 9679, pages 122–137. Springer, June 2016. 9.2

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021. 9.2

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020. 9.2, 9.2

[KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 197–209. ACM, October 2021. 9.2

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018. 9.2, 9.2, 9.2

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019. 9.2

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997. 9.2, 9.2

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012. 9.2, 9.2, 9.2

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019. 9.2

[KR18a] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018. 9.2, 9.2

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010. 9.2, 9.2

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996. 9.2

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014. 9.2

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016. 9.2, 9.2

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019. 9.2

[KRRW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017. 9.2, 9.2

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012. 9.2, 9.2

[KRS+18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018. 9.2

[KRS+22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software (JSS)*, 183:1–21, January 2022. 9.2

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006. 9.2, 9.2, 9.2

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007. 9.2, 9.2

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007. 9.2, 9.2, 9.2

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008. 9.2, 9.2, 9.2

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010. 9.2, 9.2, 9.2, 9.2, 9.2

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering Journal*, 27:119–151, April 2020. 9.2

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems.* Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021. 9.2

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011. 9.2

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE.* Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017. 9.2

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010. 9.2, 9.2, 9.2, 9.2

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019. 9.2

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019. 9.2, 9.2

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013. 9.2

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015. 9.2

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer Journal*, 43(5):42–48, May 2010. 9.2, 9.2, 9.2

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017. 9.2

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019. 9.2, 9.2

[MPRW22] Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Integration Challenges for Digital Twin Systems-of-Systems. In *10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, pages 9–12. ACM, May 2022. 9.2

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010. 9.2, 9.2, 9.2

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011. 9.2, 9.2, 9.2, 9.2

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011. 9.2, 9.2

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011. 9.2, 9.2

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011. 9.2

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011. 9.2

[MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011. 9.2, 9.2

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011. 9.2

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013. 9.2

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014. 9.2, 9.2

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *International Conference on Software Engineering (ICSE'14)*, pages 95–105. ACM, 2014. 9.2

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, CEUR Workshop Proceedings 1723, pages 19–24, October 2016. 9.2, 9.2

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020. 9.2, 9.2

[MRZ21] Judith Michael, Bernhard Rumpe, and Lukas Tim Zimmermann. Goal Modeling and MDSE for Behavior Assistance. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 370–379. ACM/IEEE, October 2021. 9.2

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017. 9.2

[Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017. 9.2

[NRR15a] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015. 9.2

[NRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016. 9.2, 9.2, 9.2

[PR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013. 9.2, 9.2, 9.2

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, LNI 254, pages 93–108. Bonner Köllen Verlag, March 2016. 9.2

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002. 9.2, 9.2, 9.2

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen*. Aachener Informatik-Berichte, Software Engineering, Band 17. Shaker Verlag, 2014. 9.2

[Plo18] Dimitri Plotnikov. *NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project*. Aachener Informatik-Berichte, Software Engineering, Band 33. Shaker Verlag, February 2018. 9.2

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994. 9.2, 9.2

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999. 9.2, 9.2

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001. 9.2

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003. 9.2, 9.2

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016. 9.2

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 2014. 9.2

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996. 9.2, 9.2

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995. 9.2

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017. 9.2

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011. 9.2, 9.2

[RRRW15b] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015. 9.2, 9.2, 9.2, 9.2, 9.2

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016. 9.2, 9.2

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017. 9.2, 9.2

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012. 9.2, 9.2

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013. 9.2, 9.2, 9.2

[RRW13c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013. 9.2, 9.2, 9.2

[RRW14a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton.* Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014. 9.2, 9.2, 9.2, 9.2

[RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015. 9.2

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015. 9.2, 9.2, 9.2

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996. 9.2, 9.2

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002. 9.2, 9.2, 9.2

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003. 9.2, 9.2, 9.2

[Rum04c] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004. 9.2, 9.2, 9.2, 9.2

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage.* Springer Berlin, September 2011. 9.2, 9.2, 9.2

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012. 9.2, 9.2, 9.2, 9.2, 9.2

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016. 9.2, 9.2, 9.2

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017. 9.2, 9.2

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018. 9.2, 9.2, 9.2, 9.2, 9.2, 9.2

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012. 9.2, 9.2, 9.2, 9.2

[SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020. 9.2

[SM18a] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018. 9.2

[SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communications and Networks, pages 227–246. Springer International Publishing, 2020. 9.2

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010. 9.2, 9.2, 9.2

[TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70. Springer, July 2021. 9.2, 9.2

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013. 9.2, 9.2

[Voe11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen.* Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011. 9.2, 9.2, 9.2

[WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017. 9.2

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012. 9.2, 9.2, 9.2, 9.2

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling.* Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016. 9.2, 9.2, 9.2

[Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems.* Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, October 2021. 9.2

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days.* EUROCONTROL, 2011. 9.2, 9.2