Wolfgang Böhm
Manfred Broy
Walter Koch
Nikolaus Regnat
Bernhard Rumpe
David Schmalzing (Eds.)

Model-Based Systems Engineering with the SPES Modeling Language

A SysML Workbench for the SPES Methodology

GfSE Verlag
ISBN (Print): 978-3-910649-07-1
ISBN (elektronisch): 978-3-910649-08-8
Copyright © 2025 GfSE e.V.,
veröffentlicht unter Creative Commons Attribution 4.0 International

GfSE Verlag
GfSE e.V.
Hermann Köhl Str. 7
29199 Bremen
Registernummer: VR 16037

Preface

Any system of interest today is so complex that understanding and, in particular, its development is only practical using dedicated and explicitly defined models. Model-based system development (MBSE) becomes especially interesting when a useful tool assists a comfortable modeling language based on a formal theory and an appropriate methodology accompanied by a semantically usable blueprint to capture the relevant properties of systems and their components.

The field of model-based development is characterized by a variety of pragmatic and more fundamentally oriented modeling approaches. In this book, we describe such an approach which is based on the SPES methodology. This methodology was developed to bridge the gap between pragmatic- and fundamentals-oriented, model-based approaches. In the SPES projects, significant results were achieved that further advance the development of high-quality embedded systems and that have already become established as a methodological approach in industry.

In industrial practice, especially among small and medium-sized enterprises, system engineering projects often rely on SysML, despite all the open questions and shortcomings associated with it. SysML is not inherently built on a modeling theory or an appropriately fitting methodology. Instead, SysML merely provides abstract graphical modeling. Semantical relations exceeding the structural relations of the language towards a conceptual model are missing. Furthermore, SysML has no formal semantic foundation; thus, neither structural nor behavioral models have clear meanings. However, this is necessary for deeper analyses, automatic or at least guided synthesis mechanisms, and a common and unambiguous understanding across organizational boundaries. Thus, by using SysML without an appropriate semantic concept and a consistent development methodology, essential potentials of MBSE are not realized.

The SPES methodology overcomes many of these shortcomings by harmonizing the necessary scientific work with pragmatic procedures. In particular, the procedures were implemented in the sense of a consistent model-based development approach for embedded systems. The methodology has been deliberately worked out without concrete syntax for the models, which means that the representation, the set of valid models, is only defined abstractly. Thus, different concrete syntaxes are possible. This ensures the greatest possible breadth of application but decreases the direct usability of the methodological approach in industrial practice.

To fill the gaps, we made a significant effort in a project called "SysML Workbench for the SPES Methodology" to express the SPES methodology using a SysML language derivate (subset of SysML) and provide a prototype for a corresponding tool implementation. In doing so, we restrict the “level of freedom” (and with it the “level of ambiguity”) of SysML and their tool implementations and offer only the elements that are needed for or fitting to the SPES methodology. As a benefit, we get a description of a complete structure model of the system and define a set of well-formedness rules of the models and their mutual relations. It comprises parts of the SysML, the formal foundations of an adequate modeling theory, called *Focus*, and the comfort that a widely used modeling tool, namely MagicDraw offers. In addition, it comes with the SPES methodology that defines how to use the language elements.

In the SpesML project, we developed a modeling language that supports the SPES methodology, including the underlying *Focus* theory. This language is based on the modeling language SysML

and makes the SPES methodology seamlessly available to numerous industrial practitioners already using the widely used modeling language SysML. As adequate tools are required, which support the methodology consistently and semantic correctly and offer advanced analysis options, the SpesML, therefore, also provides a prototypical open source implementation based on a commercially available, lightweight development tool.

The book starts with a general introduction in Chapter 1 and an introduction to the SpesML in Chapter 2. In Chapter 3, the book presents an industrial outlook on model-based system development and sheds light on the challenges faced by industrial practitioners. Therein, Chapter 3.2 introduces a running example, which is used to demonstrate the methodology's application and tool implementation throughout the book.

Chapter 4 introduces the SPES methodology. Starting with the fundamental principles applied in SPES, the chapter briefly describes the underlying modeling theory and the vital core concept of the *Universal Interface Model* (UIM). Furthermore, Chapter 4 describes an approach to modeling the behavior of CPSs.

Chapter 5 introduces the elements and core diagrams of SpesML. It describes the subset of the SysML model elements used, including statecharts and internal block diagrams. Furthermore, it introduces the concept of expressions as an extension of the SysML language for formal behavior descriptions. Afterwards, Chapter 6 describes the realization of the core building blocks in SPES (UIM, viewpoints, granularity layers) through SpesML language elements.

An essential goal of SpesML was to develop a tool prototype that implements the SpesML approach. In the SpesML project, we have chosen the tool MagicDraw as a basis and built a SpesML plugin on top of MagicDraw. A description of the plugin implementation is given in Chapter 7.

Finally, Chapter 8 introduces three industry case studies and Chapter 9 concludes with a summary and an outlook.

Wolfgang Böhm
Manfred Broy
Walter Koch
Nikolaus Regnat
Bernhard Rumpe
David Schmalzing

Summer 2023

Inhalt

Inhalt 1
1 Introduction 7
2 Introduction to the SPES modeling language (SpesML) 11
2.1 On Model Based System Development 11
2.1.1 Physical and Technical World 11
2.1.2 Use Cases for Modeling Languages 11
2.2 The SPES Modeling Language 12
2.2.1 SpesML Semantic Theory 13
2.2.2 System Modeling Blueprint 14
2.2.3 Stages in Development 16
3 Challenges – The Industrial View 19
3.1 Initial Situation 19
3.2 Introduction of a Running Example 20
3.2.1 Hardware Setup 21
3.2.2 Functionality 21
3.2.3 Requirements 22
4 SPES Methodology 25
4.1 SPES Principles 25
4.1.1 Principle 1: Interfaces 25
4.1.2 Principle 2: Refinement 26
4.1.3 Principle 3: Abstraction 26
4.1.4 Principle 4: Decomposition 27
4.1.5 Principle 5: Views 27
4.1.6 Principle 6: Granularity Layers 28
4.1.7 Principle 7: Traceability 28
4.2 Underlying Modeling Theory of SpesML 29
4.2.1 Interfaces in the Focus Theory 29
4.2.2 Streams 29
4.2.3 Stream Processing Functions 30
4.2.4 Composition and Feedback 30
4.2.5 State Machines 32

4.3 Core Concepts of SPES ... 33
4.3.1 Generic System Model ... 33
4.3.2 Universal Interface Model ... 34
4.3.3 Interface Modeling of Physical and Mechanical Systems ... 36
5 SpesML – A Modeling Language for the SPES Methodology ... 41
5.1 From Methodology to Language ... 41
5.2 Data Types ... 43
5.2.1 Primitive Types ... 44
5.2.2 Enumerations ... 45
5.2.3 Value Types ... 45
5.3 Expressions ... 45
5.3.1 Operands ... 46
5.3.2 Operators ... 46
5.3.3 Evaluation Order ... 46
5.3.4 Implicit Casting ... 46
5.3.5 Well-formedness ... 47
5.3.6 Arithmetic Operators ... 47
5.3.7 Assignment Operators ... 47
5.3.8 Comparison Operators ... 48
5.3.9 Logical Operators ... 48
5.4 Functions ... 48
5.4.1 Overloading ... 50
5.4.2 Recursion ... 51
5.4.3 Package and Modules ... 51
5.5 Statecharts ... 52
5.5.1 States ... 53
5.5.2 Transitions ... 53
5.5.3 Actions ... 54
5.5.4 Hierarchy ... 55
5.6 Application of the Universal Interface Model ... 55
5.6.1 System Element ... 55
5.6.2 Syntactic Interface ... 56

5.6.3 Channels ... 56
5.6.4 Data Types ... 56
5.6.5 Composition ... 56
5.6.6 Example ... 57
5.6.7 The Spes ML Metamodel for the UIM ... 58
5.6.8 Well-formedness Rules ... 60
6 Usage of the Language Constructs in the SPES Methodology ... 63
6.1 The system and its context ... 63
6.1.1 Elements of the operational context ... 64
6.1.2 Dimensions in context modeling ... 65
6.2 Requirements Viewpoint ... 71
6.2.1 Model Elements ... 73
6.2.2 Tracing ... 74
6.2.3 Requirements Fulfillment ... 75
6.2.4 Methodology ... 77
6.3 Functional Viewpoint ... 80
6.3.1 Model Elements ... 80
6.3.2 Tracing ... 85
6.3.3 Well-Formedness Rules ... 86
6.4 Logical Viewpoint ... 88
6.4.1 Model Elements ... 88
6.4.2 Logical Context ... 89
6.4.3 Decomposition of Logical Components ... 90
6.4.4 Tracing ... 90
6.4.5 Modeling the Transition to the Technical Viewpoint ... 91
6.4.6 Well-formedness Rules ... 91
6.5 Technical Viewpoint ... 92
6.5.1 Model Elements ... 93
6.5.2 The Software Execution Subsystem ... 96
6.5.3 Tracing ... 99
6.5.4 Well-Formedness Rules ... 100
6.6 Modeling of Subsystems ... 102

6.6.1 Layers of Granularity 103
6.6.2 Development Styles with Subsystems 103
6.6.3 Integration of Subsystems 105
6.6.4 Context of Subsystems 106
6.6.5 Tracing Between Layers of Granularity 106
6.6.6 Software Execution Subsystem as Explicit Example 106
6.7 Tracing between Views and Layers of Granularity 107
6.7.1 Models in SPES 107
6.7.2 Requirements Tracing 109
6.7.3 Tracing between Elements in the Functional and Logical Views 110
6.7.4 Definition of Software-Components in the Logical Architecture 112
6.7.5 Tracing Relationships between Elements in the Logical View and Technical View 113
6.7.6 Tracing Relations for the Software Execution Subsystem 114
6.7.7 Tracing of Context Elements 114
6.7.8 Tracing between Layers of Granularity (Tracing between Subsystems) 115
6.7.9 Tracing for Software Execution Subsystems 116
6.7.10 Tracing of the Operational Context 117
6.7.11 Development Against Assumed Requirements 117
7 A MagicDraw Plugin for the SpesML 120
7.1 Overview 120
7.1.1 MagicDraw SpesML Profile 120
7.1.2 MagicDraw SpesML Model Template 121
7.1.3 MagicDraw SpesML Perspectives 122
7.1.4 MagicDraw SpesML Java Plugins 122
7.2 Extension of the SysML Metamodel 122
7.2.1 SpesML Elements 123
7.2.2 SpesML Diagrams 124
7.3 Extension of MagicDraw with Expressions 127
7.3.1 Implementation of a Textual Language 128
7.3.2 Adapting Graphical Elements to MontiCore Symbols 131
7.3.3 Integration of Textual and Graphical Model Elements 135
7.4 Validation Rules 136

7.4.1 ChannelOutDirection 136

7.4.2 ConnectedProxyPortsOfBlock 136

7.4.3 DistinctTaskAllocation 136

7.4.4 EmptyName 137

7.4.5 EmptyType 137

7.4.6 EnoughMemorySpaceForTask 137

7.4.7 EnoughRAMCapForTask 137

7.4.8 MatchingTaskPortAllocation 137

7.4.9 NoPorts 138

7.4.10 OnlyOneBehavior 138

7.4.11 PlatformElementConnections 138

7.4.12 PortTiming 138

7.4.13 PureSoftwareTracing 138

7.4.14 STMEffects 138

7.4.15 STMGuards 139

7.4.16 SufficientAsilLevelForTask 139

7.4.17 TaskPortConnection 139

7.4.18 EmptyEnum 140

7.4.19 PartsAndValueProperties 140

7.4.20 PortCycles 140

7.5 Simulation 140

7.5.1 Concept 141

7.5.2 MontiArc Background 141

7.5.3 Translation 142

7.5.4 Test Execution 145

8 Case Studies 148

8.1 Parking-Lock (Schaeffler) 148

8.1.1 Domain context 148

8.1.2 Case Study Introduction 148

8.1.3 Requirement View 149

8.1.4 Functional View 149

8.1.5 Logical View 152

8.1.6 Technical View 156
8.1.7 Crosscutting Concepts 158
8.1.8 Sum-up and Overall Evaluation 161
8.2 Simplified Radiography System (Siemens Healthineers) 162
8.2.1 Domain context 162
8.2.2 Case Study Introduction 162
8.2.3 Requirement View 163
8.2.4 Functional View 164
8.2.5 Logical View 169
8.2.6 Technical View 180
8.2.7 Crosscutting Concepts 187
8.2.8 Final Thoughts 189
8.3 Anomaly Detection System (Qualicen) 189
8.3.1 Domain Context 189
8.3.2 Case Study Introduction 190
8.3.3 Requirement View 190
8.3.4 Functional View 192
8.3.5 Logical View 194
8.3.6 Technical View 198
8.3.7 Crosscutting Concepts 201
8.3.8 Sum-up and Overall Evaluation 204
9 An Outlook based on Spes, Crest, and SpesML 207
9.1 Modeling and analyses of advanced topics 208
9.2 Fostering reuse 209
9.3 Standardization and Tooling 209
9.4 Successful introduction of SpesML 209
9.5 Conclusion 209
Appendix 211
Partner 211
Author Index 215

Wolfgang Böhm
Manfred Broy
Bernhard Rumpe

1 Introduction

Industrial Introduction of SpesML

Model-based development in software and systems engineering (MBSE) is of great relevance for the design of software-intensive systems. To introduce MBSE into a company, the following elements are required:

- A modeling framework and language that provides notation and a methodology to create and use (semantically) valid models,
- a tool implementing the modeling language and methodology, possibly expanded by options for analysis, simulation and refinement of the models, and
- a process for staff onboarding and training.

With SpesML and the prototypical tool in form of a MagicDraw plugin, as well as the MBSE maturity model, all necessary elements are available for a successful MBSE introduction.

SpesML Methodology

In the SPES projects [1], [2] a methodology for the efficient model-based development of embedded systems is worked out. The methodology is based on the scientifically sound, mathematical theory FOCUS [5] and enables a fundamental understanding of the structure and semantics of systems. Different views of the systems address the concerns of the stakeholders involved in the development. The methodology is deliberately worked out without a concrete syntax for the models, which means that the representation of the set of valid models is only defined abstractly. Various concrete syntaxes are thus possible.
The declared goal of the SpesML [3] project was to supplement the SPES methodology by a modeling language with a concrete syntax and elaborated semantics. The defacto standard language SysML was chosen as the syntactic basis, such that the "rules" of the SPES methodology are applied to SysML models. SysML clearly provides the syntactic universe of all possible models. In this universe, SpesML defines for a real subset the set of valid and useful models according to the SPES methodology. This approach follows the paradigms of abstraction and simultaneous restriction that have also led from general programming languages to domain-specific languages and from assembly languages to modern programming languages.
SpesML goes a few steps further: Industrial as well as governmental projects in which the SPES methodology was used revealed gaps in the SPES methodology that could (to some extent) be closed in the project. Approaches to the reuse of models, the transition from systems to subsystems (levels of granularity) and the abstract behavioral modeling of physical systems which enable a semantic specification of cyber-physical systems (CPS) address practical needs.
Perhaps the most important extension of SPES is the modeling of tracing relationships between the elements modeling the different system views including the system and its subsystems

across levels of granularity. Since such tracing relationships can become complex, in general, SpesML gives design recommendations that allow semantically meaningful tracing. Valid system models in SpesML are thus created in various stages, first by rules of the SPES methodology and are later restricted more and more by design recommendations. The result is a set of conditions (well-formedness rules) that define the well-formedness of SpesML system models within each view, as well as across views and levels of granularity.
The methodology gives the freedom to choose between very careful development (taking into account all important aspects such as security, robustness, etc.) and a more pragmatic way of working, where the developers of the system are left to decide how extensively they want to adhere the available methodology.

Tool Implementation

The resulting SpesML language is available as a prototypical implementation of a plugin for Dassault Systems' MagicDraw tool [8]. The well-formedness rules implemented there enable the tool to make statements about the semantic validity of the models in terms of the methodology. With the support of the tool, semantically invalid models are iteratively "converted" into valid models and thus support users in creating semantically correct models.
The implementation of the views and the well-formedness rules in the tool also allows context-sensitive menu navigation, with which atomic process steps are also mapped. The far-reaching guidance of the user in modeling and the validation of the created models (partly in real time) results in valid models more quickly (in terms of methodology) and also makes it easier for less experienced users to get started with system modeling with SpesML.

Introduction and training

Extensive training measures are indispensable for the sustainable introduction of the methodology in a company. With the MBSE maturity model [6], a tool is available to identify the current level of knowledge of the employees to target measures to support the structured introduction of model-based development in the company that is adapted to the individual circumstances and the training of the employees.
Extensive learning material, which was developed in a previous SPES project [4], is available for training employees in the SPES methodology.

Advantages of SpesML over similar Approaches

Why use SpesML as language for MBSE instead of "plain SysML"? SysML [7] is undisputedly the most widely used modeling language for MBSE. There are a number of tools that support SysML. However, a fundamental conception of a well-founded, powerful development method was not worked out for SysML so far, since semantic relationships of the individual model elements are not fully recorded with SysML. In contrast, the worked out method and the semantic model of SPES is a decisive success factor for MBSE.
As a language, SysML does not offer structuring of the system model that goes beyond the existing language elements. However, adequate structuring is a prerequisite for the development of semantically complete models. Careful selection of the individual modeling elements useful for a particular modeling task is required.

SysML and most of the tools that support it are "general purpose", i.e. they cover the widest possible range of applications and offer users no explicit support in the practical application of the methodology.
SpesML restricts the set of SysML models that are valid according to the method by defining rules about semantic model relationships that apply for a semantically valid SpesML model.
This means that the language is easier and more effective to use in practice, since there are fewer different syntactic ways to express facts in semantically correct ways. Since, in contrast to languages such as SysML, SpesML always imparts rules for the semantics of the models and the model relationships, the correctness of the models can be checked with the implementation of these rules using automatic and manual analyses. Overall, this leads to clearer models and fewer errors.

Role Out into Industrial Practice

With the completion of the SpesML project, the question arises as to how the modeling language SpesML can be sustainably established in the industry on the basis of its positive assets. Two things are crucial for a sustainable, successful introduction: (1) A strategic commitment by leading industrial companies to (the introduction of) MBSE in general and SpesML in particular and (2) to the hitherto scientifically shaped leadership role in the process of pragmatic implementation by industry.
The source code of the plugin based on MagicDraw is freely available under an open source license. However, the implementation of the plugin is not yet a product of industrial strength. Thus, support, further development and maintenance must be taken over (1) by the future user community, (2) by a consortium of using companies, or (3) by a tool vendor. Since the introduction of a method and the corresponding tool is a long-term, strategic decision associated with considerable investment, it means additional effort and a certain risk for industrial companies to take on these tasks on their own initiative. However, it offers a strategic option, because individual companies may not only use SpesML as is, but extend and adapt the SpesML with domain specific concepts, thus turning SpesML into a domain specific language (DSL) on top of SpesML.
Moreover, implementations based on tools that go beyond MagicDraw are mandatory. Many companies, especially large ones, have already strategically decided to use tools from other manufacturers (e.g. IBM, PTC, Sparcs) and often a larger toolset is needed to also address very early and regulate activities in a software and systems engineering process. While SpesML can easily be integrated with other tools in the toolchain, SpesML implementations must also be made available for tools from other manufacturers.
It is probably difficult for a small, product-oriented company to handle such an introduction on its own. A possible course of action would be to form a consortium of a number of companies interested in this approach and set up an organization or joint venture to help put the approach into practice. Such an organization or joint venture should pursue the following business objectives:

- Further development of the prototype into a stable tool and its maintenance
- Porting of the MagicDraw plugin to other desired tool platforms
- Customization of the tool

- Adaptation of the training material developed in the SPEDiT project to the SpesML methodology as a basis for SpesML training and introductory consulting

This requires investments. A key factor for success is therefore a strategic decision by leading industrial partners to provide support, for example through direct participation or through long-term, binding commitments for orders.
Ideally, a handful of leading industrial companies would set up a joint venture with the business goals described above as a joint venture, putting it on a solid footing from the start. At the same time, a strong signal is sent in the direction of MBSE and SpesML introduction in the industry, which other companies will then follow.
A joint venture can also be established as part of an industry forum for the dissemination of SpesML, for example along the lines of the "Center for Systems Engineering" at RWTH Aachen University. In order to underline the required leading role of industry, academic partners should only play a supporting role in the background in such a forum.

References

[1] Pohl, K., Hönninger, H., Achatz, R., Broy, M. (Eds.). “Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology”. Springer, 2012.

[2] Pohl, K., Broy, M., Daembkes, H., Hönninger, H. (Eds.): “Advanced Model-Based Engineering of Embedded Systems, Extensions of the SPES 2020 Methodology”, Springer 2016.

[3] SysML Workbench für die SPES Methodik, Projektantrag an das Bundesministerium für Bildung und Forschung (BMBF), Technische Universität München, 2020

[4] SPES Dissemination und Transfer (SPEDiT) home page, https://spedit.in.tum.de/

[5] Broy, M., Stølen, K.: “Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement”. Springer, Heidelberg (2001)

[6] Böhm, W., Broy, M., Junker, M., Vogelsang, A., Voss, S.: „Praxisnahe Einführung von MBSE, Vorgehen und Lessons Learnt“, fortiss-Whitepaper, 2020.

[7] OMG, OMG Systems Modeling Language, Version 1.6, 2019

[8] MagicDraw home page, http://www.magicdraw.com/

Wolfgang Böhm
Manfred Broy
Bernhard Rumpe

2 Introduction to the SPES modeling language (SpesML)

2.1 On Model Based System Development

Any system of interest today is so complex that understanding and, in particular, its development is only practical using dedicated and explicitly defined models. Model-based system development (MBSE) becomes especially interesting, when a useful tool assists a comfortable modeling language, which is based on a formal theory, and an appropriate methodology accompanied by a semantically usable blueprint to capture the relevant properties of systems and their components.
The field of model-based development is characterized by a variety of pragmatic and more fundamentally oriented modeling approaches. In this book we describe such an approach which is based on the *SPES methodology*. In the *SPES projects* [9], [10], [11] a modeling approach was developed that bridges the gap between pragmatic and fundamentals-oriented, model-based approaches. In the projects, significant results were achieved that further advance the development of high-quality embedded systems and that have already become established as a methodological approach in industry.

2.1.1 Physical and Technical World

The goal of the SPES methodology is to model real world systems. While a physical, real system is a concrete physical artifact which exists or will exist in the real world, modern systems almost always contain also software to a large extent. It is therefore necessary to use an integrated modeling technique that allows to describe physical and software components of a complex, integrated system, often called *Cyber Physical System* (CPS) (see [14]) in a systematic way.
There are different ways to specify a such world life system. One relevant way is based on functional *models,* which are an *abstraction* of an aspect of reality, built for the obvious purposes of development, analysis, and understanding the system's functionality according to the users' and stakeholders' needs. To describe a model, it is essential to have a comfortable, precise, and expressive *modeling language* at hand. Such a modeling language consists of a *syntax*, describing the syntactic form of the model elements and a *semantics* specifying their meaning. Hereby, it is not sufficient to specify the meaning of a single model element in isolation from the context in which it is used. A methodology, embedded in a formal *semantic theory*, which defines valid models and their relationships is needed. The linking of the models to real world systems is achieved by the concepts defined in the methodology.

2.1.2 Use Cases for Modeling Languages

The use of ad hoc system models, which describe specific views of real life systems in an informal improvised way can be considered a sub optimal approach.

In fact, a number of different approaches for a more or less “model-based development” exist and if different modeling theories are selected and used as the concepts underlying and integrating the various modeling views. their smart tooling assistance is rather heterogeneous. The pattern is always the same: The more strongly the different components of a modeling theory and their connection to the system under development are integrated, and the more densely and coherently they fit together, the more useful is the modeling theory for its methodical application and also as a basis for tool support.
Manfred Broy Bernhard Rumpe [15] argue that

1. the choice of the syntax should be guided by the *semantic domain* and its underlying theory and not the other way around and
2. the choice of the methodology and the semantic theories are closely related to the intended use cases for the modeling language.

They give a set of use cases for applying modeling languages in the development of embedded systems (Table 2-1)

Table 2-1: Common use cases for applying modeling languages in development.

D1	Specify a system to agree in a team on a specific system behavior.
D2	Describe the interface of a system/component.
D3	Describe properties for a component without looking at its internal structure.
D4	Describe the interplay of components as composed in a modeled architecture.
D5	Given a model, refine it such that additional properties hold.
D6	Given an interface model and the behavioral description, find a decomposition that implements the behavioral description, so that the decomposition is actually a refinement.
D7	Given two models, check whether they are equivalent and if not, find the inhibiting model elements.
D8	Given two models, check whether one is a refinement of the other.
D9	Given a test (for instance, input/output sequences and a property specification), analyze, simulate, or execute the model in such a way that it results in a test verdict.
D10	Check whether a model guarantees specific properties.

2.2 The SPES Modeling Language

By the *SPES Modeling Language* (in short: SpesML), a modeling language for the engineering of software intensive embedded systems has been developed. which matches all the use cases in Table 2-1:
The SPES project series ([9], [10], [11]) produced a sematic theory, the SPES methodology, which is embedded into a logical calculus *Focus* [13]. Originally, the SPES methodology has been developed without a concrete formal syntactic representation. With SpesML we extend the SPES methodology and define an appropriate syntactic representation of SPES which uses SysML language elements [16] as a basis. The result is a semantically coherent modeling

language for the engineering of software intensive embedded systems in which the various constructs can be understood and used together without semantic contradictions.
Note: The approach follows the proposal from [15] that the semantic theory must first be developed before it is mapped to a semantic representation.
SpesML concentrates on functionality, disregarding physical models and the obviously also relevant geometric layout as well as materials and software implementation details. Those, however, can be integrated in the overall organizational structure SPES provides.

2.2.1 SpesML Semantic Theory

The SpesML approach described in this book, outlines a general, systematic framework for model-based development. The scientific basis is the theory and the logical calculus Focus [13], which defines the basic concepts for systems, regardless of where and how these are applied in individual models are used in development to describe certain aspects ("views") of systems.
In the SpesML approach to model-based system development, a set of views is predefined to be worked out in the course of system development. Typically, views may overlap and be in relationships to each other. This has to be reflected in the modeling theory, its models and their mutual dependencies.
Strictly speaking, the SpesML modeling theory comprises abstract formal models together with rules that describe their dependencies. These are usually formed by mathematical structures consisting of specific sets, relations and functions that can represent certain relationships, perspectives and properties of systems. For these abstract models, description techniques and a *concrete syntax* are needed for their practical application.
For example, *state machines* can be described by *state transition diagrams*. A state transition diagram is a concrete *artifact*, actually a drawing, a diagram. The diagram is a mathematical graph-structure consisting of nodes and edges, which can be additionally annotated by certain formulas. From these marked graphs mathematical formulas can be derived, which describe state machines. It would also be possible to use a tabular notation to represent the same information. All three, the diagrams, the tables, and the mathematical formulas are syntax. The abstract model is a state machine, describing a *state space* (a set of states), a set of initial states, and a *state transition function*. Classical elements of mathematics and logics are used to formulate the *abstract model*, often also understood as *language semantics domain [12]*.
The meaning of the models is formalized by giving *semantics* to the syntactic forms. Semantics is formally given by mapping the concrete syntactic model into such an abstract model, defined by mathematical constructs, sets, relations, functions, predicates plus a logical calculus to derive logical properties from the syntactic forms. This is what we call *abstract formal models*. Concrete and abstract syntax and the corresponding abstract formal models together with a logical calculus allow us to manipulate, transform, and reason about these models in semantically useful ways. They form a modelling theory.
However, this does not mean that developers have to work in terms of these formal theories. Instead pragmatic tools are offered (and provided by SpesML) that embed these formal theories into practical smart algorithms of various forms. This includes consistency checks, checks for conformance of refinement, tracing, behavioral consistency, refactoring assistance, variant selection, code synthesis, but also checking of completeness of the specifications, and various other language and model specific smart algorithms. The understanding of syntactic forms as

well as their meaning as descriptions of properties of real life systems is called the *pragmatics* of the modelling theories.
In SpesML the semantic theory is represented by a set of well-formedness rules leading to semantically correct (in the sense of the SPES methodology) and comprehensive models which also allow for verification and analyses (see chapter 5.1).
A highly relevant aspect is the achievable *precision* for a model and the *expressive power* of a description technique and the related modeling theory. There are many different aspects and properties of systems. In principle, we may talk about all of these using natural language. However, sufficient precision is rarely reached with natural language. Modeling theories and formal description techniques give the needed abilities to describe the desired precision. Therefore, the modeling language used must be sufficiently expressive to cover all relevant aspects and the developers need to be aware, where the limits are. Moreover, there is always a pragmatic aspect in the interpretation of the descriptions in terms of properties of real world systems, even though many of today's languages have their interpretation rather fixed at least informally.

2.2.2 System Modeling Blueprint

A *system model* in SpesML is a model of a system (usually a system to be developed, a *system under development* - SUD) which must not be confused with the real world system it represents. A system model usually consists of a larger set of sub-models that are composed hierarchically. For real world systems we typically develop (modeling) *views* that are the result of taking specific *viewpoints* to look at the system. Viewpoints are described by sets of models together with a definition of how these models compose to describe that specific view typically addressing the concerns of specific *stakeholders*. Real world systems show many views. In SpesML there are a number of standard views, but more specific forms of views that may depend on the application domain are also common and may be integrated into the SpesML framework. Those views may address material, color, temperature, and many more. As a consequence, it is literally impossible to describe all details of real world systems in a comprehensive manner. Every set of descriptions is necessarily incomplete and thus represents an abstraction.
A system model, which may include a number of sub-models for describing different views, can therefore be understood to represent such an abstraction of the real life system.
After all, what is needed is a kind of a *blueprint* of the system modeling concept which describes the overall way systems and their properties are described at different levels of abstractions, and for each modeling element we need a syntactic description, sometimes also alternatives, and a semantic definition for them. In addition, it has to be worked out how the different models fit and work together. Typically, the different abstract modeling elements are used and reused at different parts of the overall system model. The SpesML concept model describes the principal form of systems under consideration. It specifies what constitutes a system and its properties (see Figure 2-1).
SpesML uses the *component* concept as such a blueprint, quite like object oriented approaches use the concept "class". The component concept generally plays an important role in methodologies for developing systems with distributed, interacting elements.
From an abstract point of view, typically a system we are interested in is embedded in its environment and communicates with other systems in the environment. We call this the system's

operational context. The system under consideration interacts with its operational context that influences and is influenced by the system at runtime. The *system's behavior*, essential system properties, and other aspects that have to be considered during development determine this interaction and can be observed by an external observer at the *system interface*, separating the system from its operational context. This constitutes the *black box view* onto the system and enables to describe the system and its properties at an appropriate level of abstraction.
The *glass box view* reveals the *inner structure* (*architecture*) of the system consisting of a state machine or of connected and interacting elements (components and *subsystems*), which can be considered as systems by themselves, with contexts and observable behavior and interface properties. This way the blueprint is applied hierarchically, allowing to shift the system under consideration (actually the component under consideration) between various levels of the system/component composition hierarchy.
During the course of the system development more and more detail is added to the system models and their specifications. This process is captured by the concept of *refinement*:

- *Glass box refinement* captures the transition from a black box view to the glass box view representing the system by a state machine or decomposing the system (or a system element) into a distributed network of elements (e.g. functions, architecture components), which again can have an internal structure (system sub-models). The refinement of system sub-models leads to a refinement of the overall system (*modularity*).
- *Property refinement* describes the transition between models or model specifications, where the refined models have all the properties of the initial models and some more. This process restricts the set of specified models. For example, this allows us to add additional requirements to a specification or further restrict possible behavior of the system described by the specifications.
- *Interaction refinement* covers the transition from a model or a model specification (for example, interface specifications) to another model or a specification that changes the granularity of the interaction, the channels of the interface, and their message types. We refer to the *syntactic interface* of the system in a system model (an architecture) as the *level of abstraction* at which the model is considered. The system can be described independently at different levels of abstraction for each system model. Interaction refinement changes the syntactic interface of the system and thus the level of abstraction on which the system is modeled in the current architecture view or maps the system interface of different architecture views to each other.

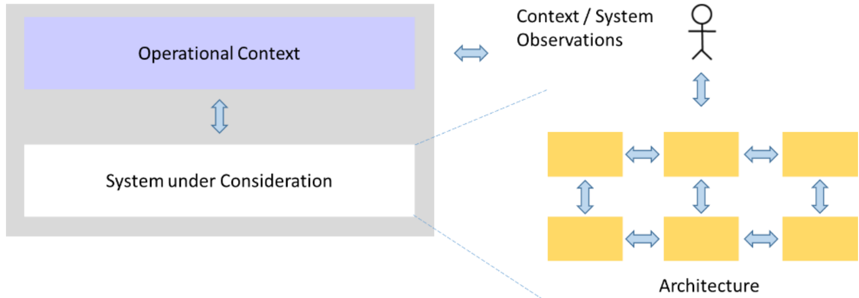


Figure 2-1: System modeling blueprint (visualized as concept model)

Relations between the models at the same or different grades of refinement and levels of abstraction maintain consistency of the *system specification*.

2.2.3 Stages in Development

To specify a system our approach provides different views on the system's architecture which are described in more detail later in this book:

- *Requirements view (collecting the requirements the SUD is supposed to implement)*
- *Functional view* (providing a functional system specification)
- *Logical component view* (providing the internal system structure while abstracting from implementation details)
- *Technical component view* (modeling implementation details).

The system specification is documented by a set of artifacts which must be validated and verified to guarantee consistency. Hereby the grade of formalization determines automatic validation and verification options.

References

[9] Pohl, K., Hönninger, H., Achatz, R., Broy, M. (Eds.) (2012): Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology. Springer, 2012.

[10] Pohl, K., Broy, M., Daembkes, H., Hönninger, H. (Eds.) (2016): Advanced Model-Based Engineering of Embedded Systems, Extensions of the SPES 2020 Methodology, Springer 2016.

[11] Pohl, K., Broy, M., Böhm, W., Klein, C., Rumpe, B., Schröck, S. (Eds.) (2020): Model-Based Engineering of Collaborating Embedded Systems, Springer 2020.

[12] Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"?. In: IEEE Computer Journal, Volume 37(10), pp. 64-72, Oct. 2004.

[13] Broy M., Stølen K.: Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement. Monographs in Computer Science, Springer 2001.

[14] Broy, M.: Cyber-Physical Systems: Innovation durch softwareintensive eingebettete Systeme, Springer 2010.

[15] Broy, M., Rumpe, B.: Development Use Cases for Semantics-Driven Modeling Languages, Communications of the ACM, volume 66, issue 5, pp. 62-71, May 2023.

[16] Object Management Group: OMG SysML v. 1.6 Specification, 2019.
[17] MagicDraw Home page: http://www.magicdraw.com/ , 2023

Sebastian Bergemann
Walter Koch

3 Challenges – The Industrial View

3.1 Initial Situation

For many years now, the complexity of innovative technical products that companies offer on the market has been increasing due to the integration of new functions, the networking of assemblies and the ability to exchange information with other devices. This increase in complexity poses major challenges for development departments in companies. A possible solution, to increase the size of the development teams, also quickly reaches its limits, because the coordination of the development, if necessary, over several development locations, brings additional complexity into the system.

For some years now, model-based system engineering (MBSE) has been seen as a promising approach to mastering the increasing complexity in the development of mechatronic systems. However, it has become apparent that this new method is not easy to introduce into the development organization. Those responsible are confronted with a multitude of challenges, all of which must be addressed simultaneously. There is no correct sequence. That this is not a specific problem of a single company can be seen, for example, in the Systems Engineering Best Practice Circle (SE-BPC) of the Gesellschaft für Systems Engineering (GfSE), which has been very popular for years. At the regular meetings of interested parties, the topic of MBSE is regularly on the agenda.

Very often the first question asked when it comes to introducing a new method to companies is: "Is there an IT tool we can use?" There is no shortage of them; there are several vendors of modeling tools on the market that support different modeling languages. What all tools have in common is that the user interface is populated with a multitude of controls, so that just operating the tool adds a new level of complexity to the problem space. The suppliers of the modeling tools can hardly be blamed for this, since they want to sell their solution to as many customers as possible, who in turn have many purposes, have implemented different development processes, and live in a wide variety of organizational structures. On the level of tools, it is therefore important that they have a configurable user interface that offers easy, complete adaptation to the needs of the different user groups. For this project, we used the Cameo Systems Modeler ™ from Dassault Systèmes and created a user interface adapted to the methodology via the programming interface.

The next question to be answered is which modeling language should be used. For the system description of a technical system, the Systems Modeling Language (SysML®) of the Object Management Group (OMG) has become established in the industry in recent years, so that it was also decided to use this language in this project. However, this creates the need for all persons involved in the development to learn a new language. Of course, the selected language must also be supported by the selected IT tool.

Now that the language and the tool have been determined, a decision must be made as to which methodology is to be used for modeling. In recent years, many methodological approaches for modeling technical systems have been developed at universities. In this project, in turn, the

methodological approach from the Software Platform Embedded Systems (SPES) Modelling Framework was used. Previous studies on the introduction of the SPES methodology in the project SPEDiT in manufacturing companies have shown how important the coordination of modeling methodology, modeling language and modeling tool is in order to gain acceptance by developers in the industrial environment.

Beyond these aspects, the introduction of model-based system engineering (MBSE) requires attention to the changes that need to be made to the development processes and, in turn, to the organizational structure and qualification of the people involved.

The actual goal of the introduction of MBSE, to make the increasing complexity controllable, must not be lost sight of. The models of the technical system must have a benefit for the communication between the people from different departments, not only for the system developers who are involved in the model creation. Nor should one think only of the development engineers from the various disciplines of mechanics, electronics or software, but should take into account that co-workers from sales, purchasing, production, logistics, quality and finance are involved in the design of the technical system. They all may not need to be able to create or modify the models, but they need to read and understand the information that is important to them in order to take it into account in their product development tasks.

If this aspect is to be considered when implementing MBSE, it shows the extent to which change needs to be shaped in companies. This has an impact on many people who have to be qualified to work with the models and for whom specific views of the system have to be designed. At the same time, the well-rehearsed processes of product development in the various areas of the company are changing as a result.

Some of the aspects addressed in this chapter go beyond the scope of the solution described here but must be considered when introducing MBSE in industry. Since the initial situation in the companies is very different, the description of the solution in this article reaches its limits.

3.2 Introduction of a Running Example

To illustrate SpesML concepts and the tool implementation in the SpesML project, we have chosen the *window lifter system* (WLS) as a running example. The system has the task of handling the opening and closing of the side windows in a car. This is a functionality with which usually everyone had already contact in real life and thus should be able to easily understand the basic aspects. In addition, the system is simple enough to be used as introduction example but has still several possibilities to extend it to show all important concepts and features of SpesML. The general functionality of this example is based on the description of a fictitious WLS by Houdek and Peach (2002), which we filtered and adapted to have a perfectly fitting running example.

We have only selected the system specifications that are most useful for the demonstration purpose, which means that the following system description is not exhaustive regarding a real WLS. Furthermore, it is not relevant whether our example system is perfectly realistic, because it should only demonstrate how specific SpesML concepts and the SpesML plugin can and should be used.

The following sections will introduce our running example of a WLS by first describing the elements it consists of, then explaining the functionality of the system, and finally presenting the resulting requirements that we used to develop the example model.

3.2.1 Hardware Setup

Our system consists of a standard car with four side doors, where each door has a window that can be opened and closed. The vertical movement of these four windows is achieved by four electric motors, one for each window. The movement of the windows can be controlled via switches that are integrated into the armrests of the doors. The two rear doors and the front passenger door have exactly one window lifter switch each. The driver door has four window lifter switches to enable a global control possibility, one for each window, and further three buttons for the child-proof lock, one for each other passenger window. Figure 3-1 gives an overview over the described hardware setup of the WLS.
A brightness sensor exists in the car that provides the current brightness of the environment. Every door has an own control unit, e.g., an ECU, that is connected to the respective window lifter motor to control the motor and receive the motor's feedback. In addition, a central control unit exists that is connected to all four door control units and is responsible for all the tasks that can be decided and performed centrally for all the doors and their windows. The central control unit is connected to the brightness sensor and all other input devices via Ethernet, except for the other four control units, because all control units are connected among each other via a bus system, e.g., a CAN bus. The four door control units are connected with their respective window motor via USB.
There are small sensors that detect whether a window has reached one of its end positions. When something is blocking the window movement in general, it is detected by a higher resistance of the respective window lifter motor. To reduce the complexity of the example model, we have decided to model this motor feedback simply as status feedback with four modes: whether the window is in its top position, in its bottom position, somewhere in between, or blocked.

3.2.2 Functionality

The functionality can be best explained by first introducing the basic behavior of the windows and their switches. Afterwards, we describe safety features that make the basic behavior safer before we finally show the comfort features that should support the driver and the passengers.

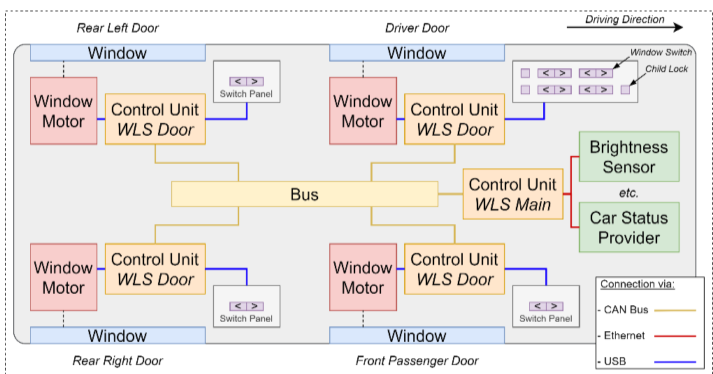


Figure 3-1: Showcase of the hardware for the WLS

Basic Functionality. Every door has a switch for its own window. If this switch is pushed downwards, the connected window will move down (opening), and if the switch is pulled upwards, the window will move up (closing). If the switch is not pushed or pulled anymore, it returns to its neutral position. The described window movements that are aligned with the positioning of the switch will continue as long as the switch is active, i.e., pushed or pulled. The window movement will stop when the switch is released, when the window is in its corresponding end position or when a safety feature overrides the movement.
The driver can control these window movements not only for the driver window but for all windows, which is why the driver door panel has three additional window switches.
Safety Functionality. As pinch protection, the window motor will immediately stop when an obstacle is detected for the current window movement and then open the window completely. A similar feature could be implemented for crash situations, but since the functionality and the implementation would be like the pinch protection, we have ignored it in this example system for the sake of complexity reduction.
Passengers should not be allowed to control the windows in a way that disturbs the driver, which is why the global movement commands by the driver switches can always override any action from other switches.
The driver can disable the window control from other passengers by activating a child-proof lock in the driver's door panel. The child-proof lock can be activated for each other passenger window separately. If the child-proof lock was activated for a window, all switch inputs by this passenger are disabled until the child-proof lock button is pressed again.
The window motors need a certain voltage to ensure a safe and complete opening or closing process. Therefore, the window motors, and thereby the window movement, is automatically disabled if the battery voltage is below a certain threshold, e.g., 10V.
Comfort Functionality. To add some comfort, it is not needed to hold the switch permanently in an active position to trigger a corresponding window movement. As soon as the switch is pushed or pulled for at least 0.5 seconds, the window will continue with the current movement until it reaches its end position or any other obstacle, even if the switch is released. If the switch is pushed or pulled again during this time after the first release, the automatic mode is stopped.
As another comfort feature, all windows are automatically closed as soon as the car enters a tunnel and if nothing else prevents or overrides it. This is detected by a rapid brightness change via the brightness sensor.
Finally, if the car is being closed and locked, all windows are automatically closed as well.

3.2.3 Requirements

For the modeling process we needed to have specific requirements that can be used in the requirement view and can be traced to model elements of the other views. Table 3-1 shows the selection that we have chosen based on the previously given system description of our WLS.

Table 3-1: Chosen requirements for our running example.

ID	Requirement
WL-1	The system shall allow the driver and the passengers to (fully or partially) open and close the side windows of the car.

WL-2	The system shall allow the driver to open and close the front and rear, left and side windows.
WL-3	The system shall allow the passengers to only open and close their own window.
WL-4	The system shall have a child protection feature that disallows operating the passenger windows for the passengers.
WL-5	The system shall automatically close all windows when a tunnel is entered.
WL-6	The system shall detect when window movement is blocked. In this case, the closing of the window shall be stopped, and the window shall be lowered completely.
WL-7	The system shall continuously move a window up or down as long as the corresponding button on the operating panel is pushed.
WL-8	The system shall fully open or close a window when the corresponding button on the operating panel is pushed continuously for at least 0.5 seconds.
WL-9	The system shall not drive the window motors to raise a window when the window is already fully closed.
WL-10	The system shall not drive the window motors to lower a window when the window is already fully opened.
WL-11	The system shall not operate the window motors when the battery voltage of the vehicle is below 10V.

References

[18] Houdek, F., Baech, B.: Das Türsteuergerät – eine Beispielspezifikation. https://wwwbroy.in.tum.de/lehre/vorlesungen/ase/ss05/iese-002_02.pdf, TUM, 2002.

Wolfgang Böhm
Manfred Broy
Henning Femmer
Maximilian Junker

4 SPES Methodology

The SpesML project builds upon decades of previous works on the SPES methodology. While not all previous work on SPES is essential to SpesML, SpesML shares the basic principles with SPES, the underlying modeling theory, as well as the core concepts of the SPES Methodology. Therefore, these three topics will be introduced in the following pages, including the formal foundations of SpesML, although the theoretical background is not necessary for modeling in SpesML.

This chapter can be read front to back as a brief primer for readers novice to SPES or used as a quick reference for central terms while reading the remainder of the book.

4.1 SPES Principles

The SPES framework introduces a set of fundamental modeling principles. These principles aim at establishing specific ways of thinking to be applied when performing modeling activities. We call the following concepts "principles" since they are not specific to a certain type of model, a certain modeling domain, or type of system, but instead hold for all modeling activities within SPES.

These principles are not to be understood as isolated concepts but instead are mostly related to each other. For example, the principle of decomposition is applicable model elements in all views and within all layers of granularity (system, subsystem, sub-subsystem, etc.). On the other hand, identifying subsystems is itself an activity of decomposition. Similarly, traces across different views tend to have a more precise semantic with a notion of an interface that is universally applied in views of different viewpoints.

In many cases the principles are supported by the underlying mathematical modeling theory which enables the principles in the first place. As an example, the modeling theory has a precise notion of composition, which allows to decompose a system element into parts and which ensures that the parts can be safely composed.

4.1.1 Principle 1: Interfaces

The concept of an interface is pervasive in SPES. It is the foundation for other principles, such as decomposition or refinement. Interfaces in SPES have two aspects: The *syntactic interface* describes which kinds of messages, signals or physical interactions can be sent or received via which channels. The *semantic interface* (also called *interface behavior*) describes how a system element reacts on its inputs by producing outputs.

SPES lays out a *Universal Interface Model (UIM)* (see chapter 4.3.2) based on a formal theory about interfaces [19]. This UIM is uniformly applied to all viewpoints, implying that functional, logical and technical interface abstractions are based on the UIM. The only exception is the

technical view, in which we do not model interface behavior, since we anticipate that the variance of this viewpoint in the various disciplines of different domains could make the modelling process complex and time consuming, error prone to keep consistent and thus not as worthwhile as in the other viewpoints.
The universal application of the UIM to all viewpoints allows for meaningful relationships between system elements of different views, fostering reuse and promoting consistent models. As an example, SPES allows to use white-box functions of the functional view to be used as building blocks inside the logical architecture by employing wrappers. Therefore, the functional architecture is no longer decoupled from the logical architecture but instead provides its basic building blocks. This is only possible because of the uniform interface concept.
System elements which are no further decomposed (called *atomic*) can be endowed with a state machine to describe the behavior of the system element. The state machines in SPES again are based on the same concept of interfaces. From the state machines an interface behavior can be extracted and a whole architecture comprising of several elements with state machines can be simulated.

4.1.2 Principle 2: Refinement

Refinement is an essential operation in any development process. In general, refining a model means adding information to the model that may add but never violate properties of the original model. A model may be refined due to several reasons. For example, new information is available that has not been available before (refinement removes uncertainty) or information is added that was not of interest for the refined model (refinement bridges *Layers of Abstraction*).

A specific type of refinement is *interface refinement*. Interface refinement addresses the refinement of information related to the syntactic interface or the interface behavior. We further differentiate between *refinement in space* and *refinement in time.* Refinement in space refers to a situation where the original and the refined interface exist in one snapshot or baseline of the same model (i.e., the interfaces exist side-by-side and the refinement describes a relation between them). This may be the case when an interface in the functional view is refined by an interface in the logical view. Both, the models of the functional and of the logical viewpoint are still valid and continue to be part of the overall model. Compared to this, refinement in time refers to a situation where the same interface is refined from one version of the model to a later version.

Refinements may manifest themselves by operations in the model such as detailing data types or adding channels, both for internal or external interfaces.

4.1.3 Principle 3: Abstraction

Related to the concept of the refinement is the concept of *abstraction.* Often, a system can be modelled differently depending how much details we add to the model. This applies especially to the modeling of physical aspects of the system. The physical reality contains much detail, most of which is irrelevant for modeling the system the functional or logical view. Choosing a suitable abstraction for the model is key to obtain a model that is both meaningful and practical. The modeling theory underlying SPES allows to choose abstractions most suitable for the task at hand. For example, in cases when real-time aspects are important for the description of a system, those can be included. If timing is not a core issue, time can be abstracted in the behavior models.

4.1.4 Principle 4: Decomposition

Decomposition plays an important role as a lever to master complexity in nearly all engineering activities. Instead of solving a difficult problem as a whole, we decompose the problem into simpler sub-problems, solve each of those and compose the resulting solutions to obtain a solution of the original problem. Focus and the UIM guarantee that this composition is semantically correct.

In SPES, the principle of decomposition is applied in several areas. For example, complex requirements may be decomposed into sub-requirements, which are usually more precise and can usually be realized and verified independently from one another. Often the step of decomposing requirements into sub-requirements is accompanied with a refinement step (see refinement principle in subchapter 4.1.5). Decomposition is also applied to architecture elements. For example, a black-box function can be decomposed into a set of interacting white-box functions. Similar, logical and technical components are decomposed into sub-components. By subsequently decomposing system elements we obtain a *decomposition hierarchy*.

The individual sub-elements resulting from a decomposition of a system element can be supplied with a specification. This refers to the specification of their syntactic interface as well as their behavior. As long as certain boundary conditions are met, their specifications can be composed to form a specification of the compound. We may then verify whether the composed specification conforms to the requirements formulated for the compound system element.

A special case of decomposition is the identification of subsystems (see principle *Layers of Granularity* below). In this case, a decomposition is aligned to a change of scope of the SuD. The subsystem is treated as a new system in the next layer.

4.1.5 Principle 5: Views

Stakeholders of the system and its description have different interests. Therefore, the resulting concerns must be considered in the system description - for example the functionality of the system, its logical structure, or its technical realization. To separate the different interests, concerns are represented in terms of one or more models with suitable model elements in a suitable representation [25].

The set of models in a system description related to a stakeholder's concern is called a *view*. In other words: A view is a subset of the set of all models used in the system description. For the system engineer, for example, this can be a functional model representing a chain of effects, for the electrical engineer it can be technical model representing electrical circuits, and for a software engineer it can be a task architecture representing software components. The contents expressed in the various views can overlap - since they each express what is of "interest" to the specific stakeholder. However, in the SPES methodology the concrete views are defined in a way that model elements do not overlap between viewpoints.

A view always describes the SuD according to a viewpoint, which is a specification for constructing, interpreting and analyzing the view to address the concerns framed by that viewpoint. Please note that for each view in a system description there exists exactly one viewpoint that specifies this view. The viewpoint is an approach to how to describe a system description while the corresponding view contains the descriptions of a concrete system. SPES defines four concrete viewpoints, the details will be given in the subsequent chapters of this book.

- Requirements Viewpoint
- Functional Viewpoint
- Logical Viewpoint
- Technical Viewpoint

In addition to the views representing stakeholder concerns, the system descriptions may also include correspondences (mappings and relations) and correspondence rules that relate models of different views. The organization and structure of system descriptions into views provides a mechanism for the separation of concerns among the stakeholders, while maintaining the scope of the whole system that is fundamental to the notion of architecture.

4.1.6 Principle 6: Granularity Layers

The concept of granularity layers allows to recursively apply the SPES concepts and viewpoints to selected architecture elements (subsystems), which leads to a nesting of architecture descriptions. It is important to point out the difference between decomposition and granularity layers. While decomposition relates to a breakdown of a system element into sub-elements, all of which are part of the same system description, identifying an element as a subsystem and thus initiating a new layer of granularity starts a new system description with the selected element as the scope. Where multiple subsystems are derived from an SuD, all these derived subsystems are part of the same granularity layer. For each of them, the syntactic and semantic interface of the subsystem must be consistent to the interface of the corresponding system element in the SuD. Please note that the new system descriptions may or may not be based on the SPES methodology.

As, in general, only selected elements of an architecture will become subsystems, the set of all subsystems will not form a complete decomposition of the SuD. It is a means to decouple the engineering processes and divide them into a number of individual fine-grained engineering processes, complemented by certain activities to support the integration of the various engineering artifacts. This enables, for example, component reuse and the integration of a supplier relation into the engineering process.

4.1.7 Principle 7: Traceability

Model elements are related to each other as well within a single view as across views and granularity layer boundaries. These relations can be very different in nature. Examples include relations describing the fulfillment of a functional requirement by a function in the functional view or the realization of a function by a component of the logical view. Although there are very different types of relationships, they share the common goal of making dependencies between model elements explicit and establish a trace that connects model elements from requirements to technical components. A typical use-case connected with those *traceability links* is to evaluate the impact of changes.

Trace relationships between model elements of a SuD from different views are n:m in general. This means, that a model element in one view is related to m different model elements in the other view, and vice versa a model element of this other view may be related to n different model elements of the first view.

Trace relationships can have a very precise formal semantics or also only a loose meaning. SpesML suggests a practical compromise between generality and the possibility of enabling

semantically expressive tracing relationships between model elements. The concrete usage of traceability in SpesML is defined in chapter 6.7.

4.2 Underlying Modeling Theory of SpesML

In this chapter we briefly outline the formal modeling theory called Focus [19], which underpins SPES and SpesML. This theory provides the semantic basis for nearly all modeling elements in SpesML, their interfaces and behavior. The aim is to provide the reader with a high-level understanding of the theory, which can be understood with basic mathematical background. Of course, this can barely scratch the surface. A full treatment of the material can be found in [21] and [22].

Please note again that modelers working with SpesML do not have to know the theoretical underpinning of the Focus theory. It suffices to know that SpesML allows to model in SysML guaranteeing the principles of Focus.

4.2.1 Interfaces in the Focus Theory

Interfaces provide a description of how a system element interacts with its environment without providing details on how the observed behavior of the system element is realized internally. Interfaces contain all relevant information regarding the interaction between the system element and its environment. Interfaces are key concepts in engineering as they enable the decomposition of systems into parts which cooperate via their interfaces to provide the system's behavior.

We consider two aspects of interfaces. The *syntactic interface* describes the general form in which a system element can interact with its environment. Depending on the type of the system we are considering (e.g. software systems or mechanical systems) the syntactic interface represents different aspects of the system (e.g. data channels for software or physical interactions for mechanical systems). In the modeling theory we abstractly define syntactic interfaces as a set of input channels and a set of output channels. Each channel has a name and an associated sort, specifying which type of messages can be passed on the channel.

The *semantic interface* or *interface behavior* describes which effects a system element realizes when it is provided with certain stimuli. For digital systems the interface behavior consists of outputs, given a set of provided inputs (if a system element takes inputs at all). For mechanical systems the stimuli and effects may also be expressed by physical quantities.

Formally, we specify a syntactic interface for a system element by providing a set I of input channel names and a set O of output channel names. Channel names for I and O must be unique. Each channel name has an associated sort M (for "messages") that specifies the types of messages that the system element can receive or provide via the channel. The syntactic interface consisting of channel names I and O is then written as $(I \blacktriangleright O)$.

4.2.2 Streams

Our goal is to model interactive systems that interact with each other by asynchronously passing messages (representing digital or physical interactions). Asynchronous here means that a system sending a message is not waiting for another system to receive the message (i.e. to synchronize with the receiver).

To represent this kind of interactions, we use the concept of *timed streams.* For a sort M of messages, the sort TStream M is the sort of streams containing elements of M. The carrier set of this sort are the finite and infinite sequences which again consist of sequences of M-elements. We write this as $(M^*)^{*|\omega}$. This is a simple model of discrete time, where each element of a timed stream represents a time interval and contains the sequence of messages received in this interval.
A number of operators and functions are defined on streams:

$_\,\&\,_ : \mathrm{M} \times T\mathrm{Stream\ M} \rightarrow T\mathrm{Stream\ M}$	(Prepend an element to a stream)
$\mathrm{ft} : T\mathrm{Stream}\ M \rightarrow \mathrm{M}$	(First element of the stream)
$\mathrm{rt} : T\mathrm{Stream}\ M \rightarrow T\mathrm{Stream\ M}$	(Rest of the stream)
$_\,\hat{}\,_ : T\mathrm{Stream\ M} \times T\mathrm{Stream\ M} \rightarrow T\mathrm{Stream\ M}$	(Concatenate a finite stream with a (finite or infinite) stream)

4.2.3 Stream Processing Functions

The interface behavior of a system element is given by the output messages that the system element sends on its output channel in reaction to the input messages it receives on the input channels. This dynamic is modeled by *stream processing functions*. Intuitively a stream processing functions maps streams conforming to its input channels to streams conforming to its output channels. We can write this formally by introducing *channel histories*. Given a set of channels C a channel history assigns a stream to each channel. The assignment is such that the streams assigned to the channels conform to the sorts associated with the channels. Formally, a channel history is a function

$$C \rightarrow (M^*)^{*|\omega}$$

We also write $\vec{C}$ for the set of channel histories for C. Stream processing functions are then functions that map input channel histories to output channel histories: If I and O are sets of input channels and output channels respectively, a stream processing function is a function

$$f: \vec{I} \rightarrow \vec{O}$$

Stream processing functions model interactive systems that process input messages and asynchronously produce output messages.

4.2.4 Composition and Feedback

In most cases it is not feasible to specify the behavior of the whole system as a single stream processing function. Instead, the system is decomposed into more fine granular system elements that cooperate to deliver the system's functions. In order to obtain a specification for the system from the specification of its elements, we need composition. There are different forms of composition which, taken together, form a general composition operator.
Parallel Composition models the situation with two independent system elements which do not interact and have disjunct output channels. We assume two stream processing functions f_1 and f_2 and define the parallel composition of both.

$$f_1: \overrightarrow{I_1} \to \overrightarrow{O_1}$$
$$f_2: \overrightarrow{I_2} \to \overrightarrow{O_2}$$

We assume the output channels are disjunct: $O_1 \cap O_2 = \emptyset$. Then the parallel composition $f_1||f_2: \vec{I} \to \vec{O}$ with $I = I_1 \cup I_2$ and $O = O_1 \cup O_2$ is given by

$$(f_1||f_2)(x) = f_1(x|_{I_1}) \uplus f_2(x|_{I_2})$$

Here, $\uplus$ refers to the union of channel histories with disjunct channel names.

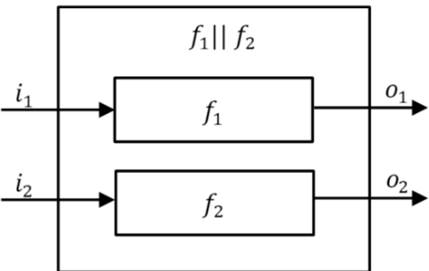


Figure 4-1: Parallel Composition

Sequential Composition models the situation where the output of a system element is the input for another system element and hence the system elements form a pipeline. Let again be two functions given where $I_2 = O_1$. The sequential composition $f_1 \circ f_2: \vec{I}_1 \to \vec{O}_2$ is given by

$$(f_1 \circ f_2)(x) = f_2(f_1(x))$$

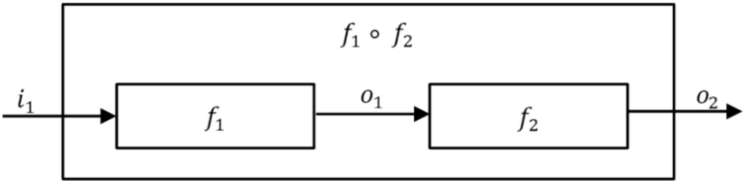


Figure 4-2: Sequential Composition

Feedback models the situation where an output channel of a (possible composed) system element is fed back into one of its own input channels. We assume a stream processing function $f: \vec{I} \to \vec{O}$. We are now in the situation where we want to feedback a channel $b_o \in O$ into one of the function's inputs $b_i \in I$. Let $I' = I \setminus \{b_i\}$. The function which results from f by feedback of channel b_o into channel b_i is written as

$$[b_i \leftarrow b_o : f]: \vec{I'} \rightarrow \vec{O}$$

and has the following definition:

$$[b_i \leftarrow b_o : f](x) = \mathrm{lfp}(\lambda y. f(x \uplus (b_i \mapsto y(b_o)))$$

Here, lfp refers to the least-fix point of the function that feeds back the output on channel b_o into channel b_i. This fix-point is guaranteed to exist under certain conditions which are detailed out in [22]. One relevant assumption that we apply in SpesML and our tool implementation is *strong causality*, meaning that the cause precedes all possible effects. In the example in Figure 4-3, strong causality implies that the output b_o, determined at time unit *t*, only depends on inputs b_i received in time units before *t*. This adequate assumption for real systems prevents having zero-time feedback loops and thereby simplifies various problems, e.g., when simulating f_1.

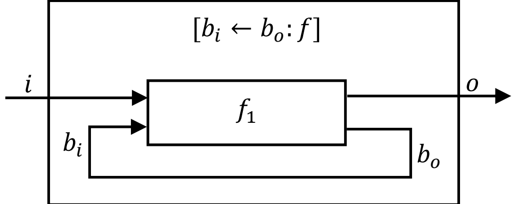


Figure 4-3: Feedback of output to input channel

Using the three forms of composition we can obtain a general form of composition with feedback.

4.2.5 State Machines

Above we discussed the interface view on system elements. However, in many situations it is more natural to describe the behavior a system element in terms of its internal states and state transitions. This kind of view on behavior can be well represented by *state machines*. There are various types of state machines depending on which aspects should be modeled. One simple example are Mealy Machines, which are state machines that can process input and produce output as a reaction.

For a set of input actions I and output actions O, such a Mealy Machine is represented by

Σ, the *state-space* of the machine
$\Lambda \subseteq \Sigma$, the set of *initial states*
$\Delta: \Sigma \times I \rightarrow \Sigma \times O$, the *transition function* of the state machine

As we will outline in later chapters, we will also employ more complex kinds of state machines that are, for example, able to capture timing behavior. When using state machines, the output actions O at the transition function then define the output streams, based on the input streams (i.e. the input actions for the state machines) as well as the internal state of the state machine.

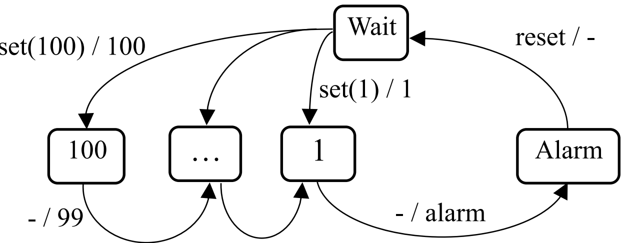


Figure 4-4: Example of a Mealy State Machine

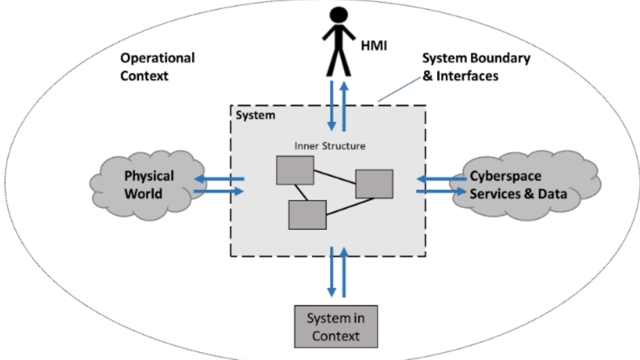
4.3 Core Concepts of SPES

4.3.1 Generic System Model

In SPES, a *system model* is a conceptual ("generic") model of systems and their properties. It describes what constitutes a system as the result of a conceptualization. System models define the components of systems, their structure, essential properties, and other aspects that have to be considered during development. Among other things, system models define what requirements refer to (subject of discourse). In SPES, the comprehensive system model consists of (see Figure 4-5):

- An *operational context* that represents what is outside the system but influences or is influenced by the system at runtime. The system's behavior, essential system properties, and other aspects that have to be considered during development are represented by this interaction. This constitutes the black box view onto the system at a given level of abstraction.
- A *syntactic interface* that clearly separates the system from its operational context and describes the options of interaction between the system and its operational context (indicated by arrows at the interface)
- An *inner structure* of interrelated and communicating elements (architecture), which are themselves systems that may be described by state machines.

Relations between the models at the same or different grades of refinement and levels of abstraction yield consistency of system specifications. The system specifications are



documented by sets of artefacts which have to be validated and verified to guarantee consistency. Hereby the grade of formalization determines automatic validation and verification options.

4.3.2 Universal Interface Model

The *Universal Interface Model* (UIM) is the foundation of interfaces and behavior across all views. As such, a SpesML modeler will never explicitly instantiate elements of the UIM, instead elements of the functional, logical and technical views will be created. However, as the UIM is the basis for all interfaces in each viewpoint models, the UIM will be applied implicitly all the time. The following section describes the concepts of the UIM. The description uses a mathematical language to describe the concepts. While it is certainly helpful to have an understanding of the underlying mathematical theory, the actual modeling with SpesML is done on a higher level of abstraction.
A system element in SpesML models the behavior of (a part of) a system according to the general SPES System Model. A system element is called *atomic* if it has no sub-elements. Otherwise, it is called *composed*. Atomic system elements are not further decomposed and directly have behavior descriptions linked to them (e.g. by interface assertions or state machine models). Figure 4-6 depicts the relation between system elements, channels, and data types.

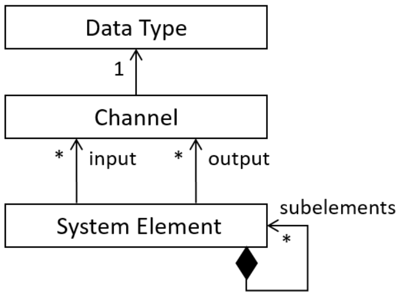


Figure 4-6: Relation between system elements, channels, and data types.

We apply the theory described above: Each system element is defined by a syntactic and a semantic interface. An example for a system element in the logical view is a logical component. System elements interact via sending and receiving typed messages via their input and output channels. Only messages of a channel's data type may be communicated via that channel.

Syntactic Interface

According to chapter 4.2 the *syntactic interface* $(I \blacktriangleright O)$ of a system element is defined by a set of input channels I and a set of output channels O. A syntactic interface can be given in terms of sub-interfaces $(I_1 \blacktriangleright O_1)$, …, $(I_n \blacktriangleright O_n)$, where the sets of input channels respectively the sets of output channels are disjunct.
Figure 4-7 schematically depicts the system element *F*. The syntactic interface of F is $(c1, c2 \blacktriangleright c3, c4, c5)$. The channel $c1$ is of type T (c1:T) and so on. The picture abstracts from the definition of the type.

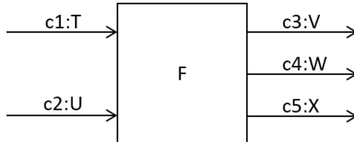


Figure 4-7: A system element and its syntactic interface.

Semantic Interfaces

The semantic interface (or interface behavior) of a system element with the syntactic interface *(I ► O)* is given by mapping input *communication histories* to sets of output communication histories. We represent the *black box* or *interface behavior* of a system element by a set-valued function (we also speak of the semantic interface of the system):

$$F: \vec{I} \rightarrow \wp(\vec{O})$$

For each possible input $x \in \vec{I}$ on the channels contained in *I*, the function specifies all possible outputs *F(x)* for the input. Thus, the function *F* represents an underspecified or non-deterministic behavior of a system element with the syntactic interface *(I ► O)*.

Causality

A semantic interface $F: \vec{I} \rightarrow \wp(\vec{O})$ is called *weakly causal* (*strongly causal*), if the output up to time-point $t+1$ only depends on the input received until $t+1$ (until t in case of strongly causal behavior).
If a semantic interface is not weakly causal, then it cannot be realized by any system element. This holds because semantic interfaces that are not weakly causal can be interpreted to model behaviors that can react to events (receiving of messages) that happen in the future. Stated differently, if a semantic interface is not weakly causal, then the system element can already react in a time unit t to an input that it receives in a later time unit t' with $t' > t$ (for at least one input communication history).
Strongly causal semantic interfaces are even more restricted. Every strongly causal semantic interface is also weakly causal. For strongly causal interface behaviors, the output in a time unit only depends on the messages received before that time unit. Thus, without even receiving a message in a time unit, the component can already determine its possible outputs for the time unit *t*, which is based on the input received so far. Strongly causal semantic interfaces are useful in the context of the composition of system elements. The composition of a strongly causal system element with another system elements guarantees nice properties in the sense that the composition is guaranteed to be a well-defined system element.
Every weakly causal semantic interface can be transformed to a semantic interface that is strongly causal modulo some of its output channels by

- delaying the messages communicated on the output channels and
- adding initial outputs to the channels (initial outputs may be empty sequences)

It has been shown that every specification of a semantic interface can be refined into a specification that is strongly causal. Interface specifications of systems that are to be composed result in system specifications for the composite system by simple conjunction. In case that some of the channels that are used for feedback to connect the composed components should be no longer visible, they can be hidden by existential quantification.
If system interface specifications of the subsystems are incomplete and describe only partial properties of their behaviors, this results in system specifications of the composed system which are incomplete. One step into completeness is a refinement of the system interface specifications of the subsystems to be composed by adding strong causality which, in general, leads to stronger specifications such that for the composed systems stronger specifications are achieved. Refinement of systems by strong causality is always possible, but it may lead to interface specifications that are not implementable since the specifications may become contradictory. Specifications that contradict the principle of strong causality are refined to false. Therefore, the refinement by strong causality is a test whether the specification is sound.

Describing Semantic Interfaces

A semantic interface can be described by an interface assertion involving the input and output channels of a system element. Equivalently, in case the behavior is weakly causal, it can be described by a state machine, e.g. a Mealy Machine. System elements, their syntactic interface, and their interface behavior can be composed to form again a system element with composed syntactic and semantic interfaces.

4.3.3 Interface Modeling of Physical and Mechanical Systems

The engineering of cyber physical systems (i.e. systems consisting of physical and digital parts) is a very complex task that usually requires extensive expertise in the individual engineering disciplines (mechanical engineering, electrical engineering, computer science). The development of components from these disciplines also requires specific tools and processes. The SPES methodology supports the engineering of such systems at a certain level of abstraction and allows the integration of specific development processes via the mechanism of granularity layers which will be discussed in more detail later in this book.
In CPSs, the software components control and monitor the physical components. The SPES methodology was developed specifically for those systems where the focus is on the control of physical subsystems by software subsystems. Therefore, we choose some interface abstraction for the modeling of physical parts (see Figure 4-8 A lower layer of abstraction of the state transitions can be achieved by ordinary differential equations (ODE) in so-called hybrid programs [26]. However, the modeling of such differential equations might be very complex for many physical systems. Therefore, a suitable abstraction of the behavior of the physical part is required.
In the logical architecture of systems (logical view), we decompose a system into components that communicate via data streams, applying the UIM described above. Following our design approach, the architecture comprises components that model physical systems and components that correspond to software services. Some of these software services control the physical system using actuators and sensors that convert the digital models into analog (physical) models and vice versa. This control can be very tight or very loose.

Controller

Control Layer

in x, b: Stream Data
out y, a: Stream Data

Interface Assertion

Phys. Subsystem

Physical Device

in a: Stream Data
out b: Stream Data

Interface Assertion

x: X
y: Y
Control Layer
a: A
b: B
Physical Device

Figure 4-8: Controller and physical subsystems as relations on streams

In SpesML we choose state machines as abstract discrete models of physical systems [22] that are controlled by actuators and sensors. In our particular simple version of theses state machines in each time interval on each input channel or each output channel there is either a message from the set of predefined messages for that channel or the input or output is empty which is represented by a special symbol ε for *no message.*

As a result, we model the behavior of physical components in the logical architecture by state machines where we describe the particular physical state of the system in some abstraction. Those discrete models are in contrast to classical control theory where systems are modeled typically by partial differential equations.

A discrete state machine first models the component representing the physical device (physical component) independent from the (digital) control component, which again can be modeled via separate state machines. Applying the UIM, the interface between the physical and the control component is given by the data streams between the control component and the actuators and sensors of the physical component. Which states of the physical component can be monitored via this interface depends on whether coupling of the software control application is more tight or more loose.

For modeling the control component, we use state machines with input and output that define a relation between input and output streams or input and output histories which represent the abstract interface behavior of the physical system, as it is seen from the control component which just looks at the data streams of signals going through the sensors and actuators. The overall behavior of the system can be modeled as a composition of the behaviors of the physical and the control component (see Figure 4-9).

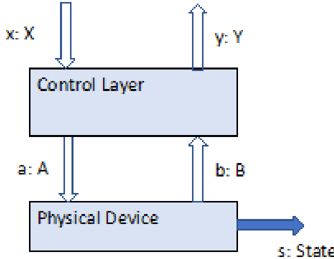


Figure 4-9: Architecture of a control layer composed with a physical device and showing the stream of states

For the discrete model of the physical components we define an attributed state space where each state is represented by a record of state attributes with a given type. The attributes and their types should be chosen to reflect all relevant aspects of the physical components in the form of the states of their elements and also relevant aspects of the behavior of the physical components. In our window lifter running example introduced in chapter 3, the state machine might look as follows:
Consider the physical window as the physical device to be modeled. A state consists of two attributes:

mode: {stopped, goin_up, goin_down, alarm}

p: [0:100]

Here p stands for position and represents the position of the window. The mode indicates the actual movement of the window, the position indicates how far the window is closed (p = 100 holds if the window is closed, p = 0 holds if the window is fully open). Alarm models the stopping of the window movement in case something is trapped (pinch guard). The state transitions can be defined by a simple table:

Table 4-1: State transitions of the window lifter with inputs and outputs

mode	p	input	mode’	p’	output
≠ alarm		stop	stopped	= p	stopped
stopped		ε	stopped	= p	stopped
goin_up | stopped		close	goin_up	= p	mov_up
goin_down | stopped		open	goin_down	= p	mov_down
goin_up	= 100	ε | close	stopped	= 100	closed
goin_up	< 100	ε | close	goin_up	> p	mov_up
goin_up			alarm	= p	alarm
goin_down	= 0	ε | open	stopped	= 0	open
goin_down	> 0	ε | open	goin_down	< p	mov_down

alarm	> 0		alarm	< p	alarm
alarm	= 0		stopped	= 0	open

References

[19] [Broy and Stolen 2001] Broy, M., Stolen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement. Springer 2001.

[20] [Broy 2010] Broy, M.: A Logical Basis for Component-Oriented Software and Systems Engineering. In: The Computer Journal vol. 53 no. 10, pp. 1758 -1782, 2010.

[21] [Broy 2019] Broy, M.: Logische und Methodische Grundlagen der Programm- und Systementwicklung. Springer, 2019.

[22] [Broy 2022] Broy, M.: Modeling Cyber-Physical Systems. To appear, 2022.

[23] [Broy 2023] Broy, M.: Logische und Methodische Grundlagen der Entwicklung Verteilter Systeme. Springer, 2023.

[24] [Böhm et al 2021] Böhm, W., Broy, M., Klein, C., Pohl, K., Rumpe, B., Schröck, S.: Model-Based Engineering of Collaborative Embedded Systems. Springer, 2021

[25] [ISO 42010:2022] ISO/IEC/IEEE 42010:2022: Software, systems and enterprise – Architecture description. 2022.

[26] [Platzer 2013] Platzer, A.: Lecture Notes on Foundations of Cyber-Physical Systems. Carnegy Mellon University, 2013.

Florian Drux
Henning Femmer
Maximilian Junker
Mathias Pfeiffer
Bernhard Rumpe
David Schmalzing

5 SpesML – A Modeling Language for the SPES Methodology

5.1 From Methodology to Language

The SPES projects define a methodology that enables efficient, secure development of even safety-critical systems. This methodology first provides a basic understanding of systems and their semantics. Furthermore, different views of the systems and dependencies between these views are defined. Finally, a development process is recommended, which allows the creation and further development of models based on the mentioned semantics, views, and dependencies. The methodology remains intentionally syntax-agnostic, i.e., a concrete syntax for the models is not introduced. This has the consequence that the language, i.e., the set of the valid models, is only abstractly defined. Thus multiple expressions in different concrete syntaxes are possible. Providing a language with concrete syntax, clear semantics and accompanying assistance systems is the declared goal of SpesML. This concrete language drastically lowers the hurdle to create, manage and use SpesML models. It also makes it possible to provide a set of tools such as editors, simulations and analyses.

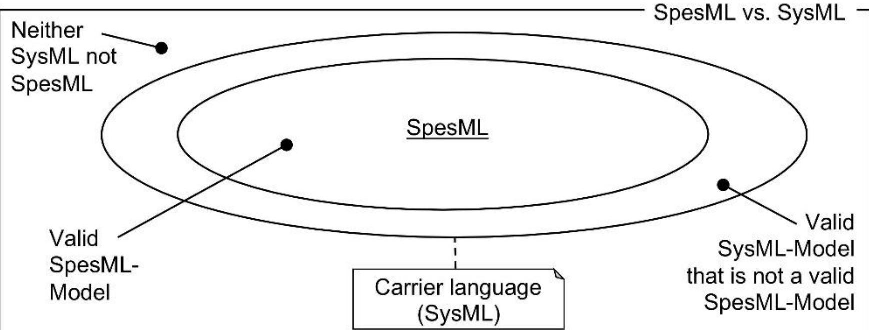


Figure 5-1: SpesML defines a language. A language thereby is the set of valid models. Its carrier language is SysML, i.e., the set of all SysML models. The rules and hints of SPES define a strict subset of SysML.

There are two options for creating the language and particularly its syntax. It is conceivable to introduce a completely new syntax for the abstract SPES language, designed from ground up. This would however not correspond to the requirement to lower the entry threshold, but would instead even introduce new hurdles. Therefore the SpesML is built on top of a carrier language. The defacto standard language in systems engineering, SysML, was chosen as the carrier. Figure 5-1 makes the definition of SpesML on the carrier language SysML more tangible. In effect, the rules of the SPES methodology are applied to SysML models. Figuratively, SysML defines the

universe of models that are at all possible but not yet necessarily valid. In this universe, SpesML defines a real subset, the set of valid SpesML models. In doing so, this approach follows the paradigms of abstraction and simultaneous restriction known from general purpose languages (GPLs) to domain specific languages (DSLs) or assemlber languages towards modern programming languages. The abstractions and restrictions are made possible by the domain-specific nature of the modeling language. This means only solutions to problems of the target domain must be expressible. The domain here being systems engineering. This makes the language easier to use as there are fewer syntactically different solutions to semantically express the same thing. Models become clearer and errors fewer.

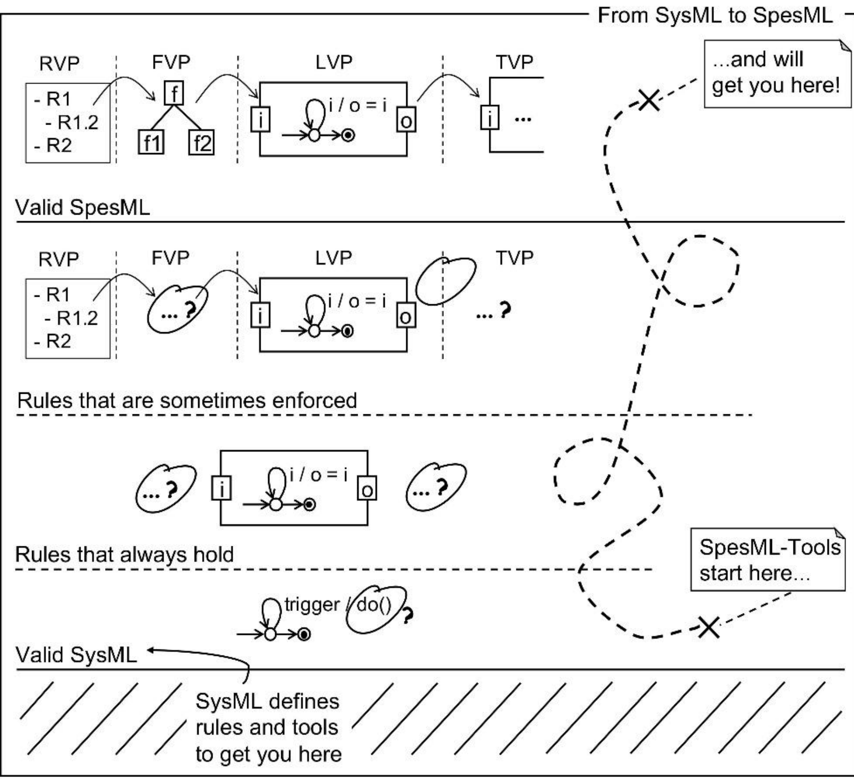


Figure 5-2: A vizualization of the journey from SysML to SpesML models. Starting from a valid SysML model in the right bottom corner, the rules and recomendations of the assistance system guide the development in such a way as to reach a valid SpesML model in the top right corner.

It is now imperative to define the language SpesML in a well-defined and precise way. Without such a well-defined language, assistance systems are not able to determine the validity of the models and are subsequently unable to provide feedback to the user. But a binary answer is not good enough. Good assistance systems should provide guidance to the users. An invalid model should iteratively be turned into a valid model with the help of the SPES methodology. However, the user should not have to know all the subtleties of SPES by heart, but should use it intuitively though the assistance system. Therefore, in addition to the distinction between invalid and valid, levels of validity must also be considered and, in particular, instructions and specifications must

be defined as to how the next higher level and, finally, a fully valid model can be achieved. Thus, guidance through the process is needed. Based on the rules induced in the SPES methodology, SpesML also defines new, non-strict recommendations for this purpose, which warn against entering a path that decisively increases the complexity of the development. Thus, the goal of SpesML is also to sharpen SPES in such a way that clearer paths become visible in order to achieve better results more quickly. Following [29] and based on the three levels of the SPES methodology (system model, dependency model, process model), four levels of model validity can be distinguished. The floor is formed by valid SysML models. Above this floor are three hierarchy levels of model validity. These levels allow an empirical, engineering-like approach to modeling and permit measurements of validity and thus quality of the models. A graphical overview of the validity hierarchies is provided in Figure 5-2.

In the first hierarchy-level, always-applicable rules are examined. These are typically syntactic in nature. For example, in SpesML only already defined blocks may be instantiated in Internal Block Diagrams (IBDs). In addition, the reference- and type-correct use of the textual language parts (see following subchapters) is enforced here. In the second level of the hierarchy, relationships between model elements are examined. These rules typically do not always apply, but are enforced only on certain model elements depending on their semantics. Examples are the correct use of the four views of the SPES methodology including other inter-element dependencies. The last hierarchy level finally deals with inter-model dependencies and the process of model evolution itself. These rules typically apply only at certain checkpoints during the developmenent process or even at the end of the process. A typical example is the complete tracing of all requirements and checking the completeness of all views and layers of granularity. The collection of all rules and recomendations thus describes a schematic path towards correct models. This path is not always a straight line however and the process of reaching a valid model might include re-evaluating previous steps. Tightly meshed rules should prevent the process from running in the wrong direction for too long, reducing the potential economic damage.

In summary, SpesML implements the methodology of the SPES projects by means of a concrete language, rules and recomendations to produce valid models in said language, and an assistance system bundling it for the engineer. SpesML is multi-layered and -faceted however. And this is reflected in this book. To this end, this introductory subchapter is intended to provide an overview of the importance of a well-defined modeling language and its central characteracter in SpesML. Chapter 5 will continue by introducing the language elements necessary for expressing detailed interface and behavior descriptions of system components. Chapter 6 will then build on those elements and presents a mapping of the SPES methodology to the language elements developed in Chapter 5. Finally, Chapter 7 summarizes the development of an assistance system bundling language elements, methodological considerations, and the resulting rules and recomendations that enable creation, management, and analysis of SpesML models. The rules and recommendations hinted at throughout this section are introduced and explained in Chapter 7. The reader has the opportunity to find and reference the any rule there and in tabular form in the appendix.

5.2 Data Types

Many software development methods use data models to describe the structure of a system or the data contained in a system. For example, data types are used in object-oriented development,

where complex data types are defined via classes that describe the structure of and relationships between objects. Such classes are part of UML as a modeling element of class diagrams. SysML has a similar but slightly modified element in the form of blocks.
In SpesML, data types are primarily used to describe the structure and relationship of data. On the other hand, a system's structure can be described by an internal block diagram in the form of the decomposition of the system into subsystems. In SpesML, data types are an important part of the specification of constants, variables, and functions. In addition, data types form the basis for formulating predicates in the context of objects.
Data types are furthermore an essential part of a system's syntactic interface description, as each channel has a type. A channel's type defines a contract about the structure of messages sent between interacting components. Without these contracts in place, two problems arise. From an interior perspective, the system must handle any eventuality or make guesses about an input's structure, which is not feasible for complex systems handling multiple inputs and managing complex behavior specifications. Similarly, from an exterior perspective, a system without clear interface contracts struggles to be re-used as potential applicants cannot be sure to trust the system to handle the inputs they throw at it, nor can they trust the output it produces.
Furthermore, SpesML defines the behavior of systems via input-output-relations over a set of channels. To model these relations, one has to know the permissible elements, i.e., the channel's types. Only then can these relations have a clear semantic and be verified to be used correctly. This argument circles back to the previous paragraph about systems not trusting their inputs nor providing a structure for their outputs. More pragmatically speaking, clearly and concisely modeling such a system becomes impossible. Specifying the relation of input to output for any potential input is clearly not feasible. Reducing to a set of "common" or "expected" values soft-defines types already, without the exterior knowing about it. Requiring an outsider to "look into" a system to understand its expected input or produced output is not scalable and goes against the common practice of black-box reuse.
Thus, SpesML's core goals of precisely and unambiguously modeling systems, as well as evaluating the validity of said models, relies on clear and well-defined type systems. Tol reinforce these goals, SpesML is statically typed. Static typing enables the check for correct usage at model design time and reduces the potential cost for erroneous specifications.

5.2.1 Primitive Types

Primitive types denote the types that are included in the language. These are the most basic types from which all other types are constructed or derived. SpesML's primitive types are inherited from SysML's primitive value types: *Boolean*, *Integer*, *Real*, and *String*. Each of those primitive types, as any type, defines a set of values. The semantics are common across most fields of mathematics. The *Boolean* consists of the two truth values *true* and *false*, i.e., $Boolean = \{true, false\}$. The *Integer* represents the set of whole numbers; examples include *-5*, *0*, and *1000*. However, due to implementation limitations, literals in this set are limited to 32-bit values, following common practices. The *Real* is an uncountable and dense type, representing the set of real numbers. Examples include finite and infinite fractions: *1.2,* respectively π. Again, due to implementation limitations, the actual set of literals is limited, here to floating-point arithmetic with single precision. The *String* finally is the set of all character lists, most often conveniently

noted in quotation marks and without any delimiter: *"This String is 33 characters long"*. Contrary to *Real*, *String* is countable.
The implementation details lead to a change in the behavior of the given types (a subset of values cannot be represented). In SpesML, this is reflected by the types *int*, *float*, and *boolean*. The types *int* and *float* represent *Integer* and *Real* with the aforementioned limitations, respectively. The type *boolean* is identical to *Boolean* and exists solely for consistency reasons. In expressions, only the types *boolean*, *int*, *float*, and *String* can be used. For defining value types, *Boolean*, *Integer*, and *Real* are available.

5.2.2 Enumerations

An enumeration is a user-defined data type representing a group of constant values (literals). An enumeration has a name and consists of one-to-many literals. Enumerations enable users to define sets of possible valuations in varying sizes, adding to the predefined, fixed-size primitive types. Furthermore, users can name enumerations to match the problem at hand, i.e., enumerations can be domain specific.

5.2.3 Value Types

A value type combines multiple types into a single type. An alternative name would be composite type. Mathematically speaking, value types are sets of tuples where each tuple element is again a type. A value type is composed other types, even other value types.
Value types are closely related to classes of UML class diagrams. A value type has a name and consists of a set of properties, the latter defining the structure and state space of the value type. Properties combine attributes association of UML class diagrams to a single element. Each property has a type and a name, just like an attribute. Furthermore, a property, like an association, may have a cardinality. Thus, a property also represents a binary relationship between two value types, but this relationship is unidirectional. And, unlike an association with additional role names, a property has only a single name.
Following SysML, there are several modifiers that can further customize a property. For a property's visibility, the characteristics public, protected, package, and private are available. A public visibility represents a generally unrestricted access, a property with protected visibility can still be accessed in subtypes, a package visibility further restricts access to the same package, and private allows access to the property only in the defining value type. Furthermore, a property may be read only. Such a property can be read anywhere in its visibility, but can only be modified in the defining value type or one of its subtypes. Thus, this modifier enables to distinguish the visibility of read and write access to a property.

5.3 Expressions

Graphical notations are especially suited to give a quick overview of the system being modeled. However, graphical notations abstract much from detail. A textual notation based on mathematic descriptions is particularly suitable for expressing conditions and complex system properties.
Therefore, SpesML provides an intuitive yet expressive textual expression language and a seamless integration with the graphical modeling elements. This unifies both the advantages of

graphical modeling in terms of ease-of-use, overview, and visualization capabilities, as well as those of textual modeling w.r.t. conciseness, preciseness, and expressiveness.
An expression computes a single strongly typed value and consists of operands, operators, and executable functions, which is comparable to expressions in common general-purpose languages, such as, for example, Java [30], or C++ [31]. Operands are the programming objects an operation can be performed on.

$$\text{switchPosition} \ == \ \text{POS.OFF} \ || \ \text{output.current} \ > \ 0$$

The above example is a *Boolean* expression, i.e., an expression producing a *boolean* value. It produces the logical disjunction, i.e., logical OR (||) between its left- and right-hand sides. On the left-hand side, the expression refers to the switchPosition toggle and checks whether the switch is in the OFF position. On the right-hand side, it is checked if the output current is greater than 0.

5.3.1 Operands

Operands are what an expression operates on. Operands in expressions can be:

- Basic literals such as 1 and ” hello ” represent values like numbers and strings of characters.
- Enumeration literals such as POS.OFF represents values of a (usually small) list of user-defined values.
- Contextually available values such as value properties, ports and their channels, and local variables.
- Expressions themselves can be composed of other expression, allowing for complex formulas. For example, in $(1 + 2) * 4$, not just the literals are operands, but the expression $(1 + 2)$ as well.

5.3.2 Operators

Operators perform operations on their operands. Operators define the number and types of their arguments, as well as their return type. Any type-invalid use of an operator or not resolvable reference will lead to validation failure. Type-correct use of operators is usage on operands with types as described in this section. Operators are classified into arithmetic, assignment, comparison, and logical operators, which are listed in further detail below.

5.3.3 Evaluation Order

Operators evaluate in a specific order that is determined by their precedence and by associativity. The operator with the higher precedence evaluates first. For example, multiplication has higher precedence than addition.
If two operators have the same precedence, then the operators are applied in order of associativity, which is left to right. For example, multiplication and division have the same precedence. The associativity is left to right; therefore, the operator on the left evaluates first.

5.3.4 Implicit Casting

We check for the types to be compatible in a given context (e.g., the corresponding operator) such that the values are not used in an undefined way, e.g., ”Apple” % ”Cherry”. Most, if not all, meaningful expressions should be accepted, and not too many special cases should be

defined, as this would affect the maintainability of the type checker. Thus, we enabled implicit casting between types.
In short, given multiple expressions of primitive types, the least restrictive type is chosen (e.g., int is more restrictive than float). The type of the more restrictive expression is implicitly changed (implicit casting) to a less restrictive type (e.g., above 1 is changed to 1.0, and after that 1.0 + 1.9 is calculated). Precisely, for primitive types int can be implicitly cast to float.

5.3.5 Well-formedness

As described, expressions are structured such that automatic well-formedness checks can be run on them. For this cause, the well-formedness relation needs to be defined, which is done in the sections below; Everything not explicitly allowed is not considered well-formed. For operands and operands, the following holds.
Operands are well-formed, if and only if they have a type, usually due to being associated with a value. Let switchPosition be a value defined in the context. Example: SwItChPoSiTiOn will raise a validation error because the switchPosition is, in this example, spelled incorrectly, as such it is not referencing a value defined in the context. switchPosition will not raise a validation error because the switchPosition is spelled correctly, as such it is a valid reference to a value defined in the context.
Operator usage is well-formed, if and only if they operate on operands with types of their specified input domains. In the following, let < be defined analogously to the corresponding operator in mathematics. Example: 5 < "six" will raise a validation error because the operator < is not defined for the types int on the left and String on the right. This is not a type-correct usage of the <-operator. 5 < 6 will not raise a validation error because the operator < is defined for the types int on the left and int on the right. This is a type-correct usage of the <-operator.

5.3.6 Arithmetic Operators

Arithmetic operators perform mathematical operations. They apply to numbers and return a number. The type of the expression is float if any of the operands have type float . The type of the expression is int if none of the operands has type float .

Operator	Name	Example	Precedence
+	Addition	a + b	10
-	Subtraction	a – b	10
*	Multiplication	a * b	11
/	Division	a / b	11
%	Modulo	a % b	11

5.3.7 Assignment Operators

An assignment operator assigns a value to a variable or port. The variable or port to which the value is assigned is specified on the left-hand side, the value on the right-hand side. The variable or port and assigned value must have compatible types. a = 2 assigns the value 2 to a variable or port called a. Variations of the assignment operator abbreviate combinations of assignment and other operators.

Operator	Example	Precedence	Abbreviation for
=	a = b	1	
+=	a += b	1	a = a + b
-=	a -= b	1	a = a - b
*=	a *= b	1	a = a * b
/=	a /= b	1	a = a / b
%=	a %= b	1	a = a % b

5.3.8 Comparison Operators

Comparison operators compare two values. They apply to primitives and return Boolean values.

Operator	Name	Example	Precedence
==	Equal to	a == b	7
!=	Not equal to	a != b	7
>	Greater than	a > b	8
<	Less than	a < b	8
>=	Greater than or equal	a >= b	8
<=	Less than or equal	a <= b	8

5.3.9 Logical Operators

Logical operators operate on Boolean variables or values, e.g., to concatenate or negate constraints. They solely apply to and return Boolean values.

Operator	Name	Example	Precedence
&&	Logical and	a && b	6
||	Logical or	a || b	5
!	Logical not	!a	12

5.4 Functions

During the modeling of a system, one may have to design a large number of expressions. These expressions may be very similar or even identical to each other. Assume one models a state machine which, no matter its state, checks that the input is in a specific range. For example, multiple transitions could have the same guard:

$$\text{port.a} > 2 \;\&\&\; \text{port.b} \,\%\, 3 == 2$$

If this guard is not met, the state machine does not change its state. Thus, each transition of the state machine contains the same subset of expressions. Assuming one may to change the sets of constraints, then one has to modify each transition individually. This is not just redundant work, additionally, forgetting to change a single transition potentially leads to failures whose sources

are hard to track down. These problems can arise inside a single expression as well, as sub-expression may be repeated, e.g.:

$$-p / 4 + sqrt((p / 4) * (p / 4) - q)$$

This expression is a wrongly implemented version of one of the p,q-formula, which calculates the result of a quadratic equation. The sub-expression p/4 is used multiple times throughout the formula. Additionally, this sub-expression not the correct subexpression for the p,q-formula, the correct sub-expression would be p/2, leading to the required change of the same expression in multiple occurrences. As such the SpesML offers a sub-language that allows for extracting sets of common (sub-) expressions into a single definition: Functions.
Functions "stand in" for expressions, allowing to use the same function in stead of the expression the functions stands for. A function has to be defined to be used. A function definition has a name and the expression the name stands for. As a function may be used in a multitude of contexts, e.g., in different parts of one expression, in different transitions of the same state machine, or even in transitions of different state machines, the function definition needs to abstract from its context. Assume, for example, the expression in the beginning:

$$\mathrm{port.a} > 2 \,\&\&\, \mathrm{port.b} \,\%\, 3 == 2$$

the port, and by extension a, and b, are available from the context of the state machine, which itself has the corresponding ports. As, however, the function standing in for the expression may be used in another state machine without the port, accessing the port (or rather the context) directly is not allowed in a function definition. Instead of accessing the context, the function definition abstracts from the context by means of parameters. A parameter can stand in for an expression in a function, just like a function itself can stand in for an expression. The difference is, that the parameter stands for a single value, e.g.:

$$\mathrm{c} > 2 \,\&\&\, \mathrm{d} \,\%\, 3 == 2$$

Above is the same expression as before, however, this expression is part of a function definition, where the parameters c and d stand in for the port attributes a, and b, respectively, which are themselves unavailable to the function definition. The corresponding function definition may look like this:

$$func(int\ c, int\ d) = c > 2 \,\&\&\, d \,\%\, 3 == 2;$$

The name of the function is "func". After the name, the parameters follow within parentheses in a comma-separated list. The parameters have a name to refer to them inside the function definition (c and d) and a type each. The type is used to check the expression defined within the function for well-formedness and is the type the usages of the name have within the expression. Using the defined function, one can replace the original expression with the expression

$$func(\mathrm{port.a}, \mathrm{port.b})$$

This expression is a call to a function. The value of port .a is bound to the parameter c. Likewise, the value of port .b is bound to the parameter d. Then, the value of the function call is the value of the expression in the function definition using the bound values, thus being equivalent to the original expression before it was replaced. Therefore, the expression can be reused multiple times while being written only once.
As an example of reuse, let us first assume that we have another transition in the state machine which has the same constraint in addition to further constraints. In this case we do not have to rewrite the full expression, but can rather reuse the function

$$func(\mathrm{port.a}\,, \mathrm{port.b}\,)\ \&\&\ \mathrm{port.b}\ <\ 100$$

This is possible as function calls are expressions on their own. Furthermore, one may reuse the function by using different arguments; Assume for example, one wants another port .e to have a value that has a remainder of 2 for a division with 3, while we do not need this constraint for port .b. In this case replacing the second parameter to the call reuses the function for a different port.
To further increase well-formedness checking, one may add the expected return type of the function to the beginning of the function definition. E.g.:

$$boolean\ func(int\ c\ , int\ d\ ) = c\ >\ 2\ \&\&\ d\ \%\ 3\ ==\ 2\ ;$$

This allows for potential checks that the value of the function definitions expression is of the expected type.

5.4.1 Overloading

Functions may be overloaded. A function is overloaded if there are multiple function definitions with the same name. In this case, any call to a function uses the function with the corresponding attributes; Assume the following function declarations:

$$float\ abs\ (double\ i)\ =\ if\ i\ <\ 0\ then\ -i\ else\ i\ ;$$
$$int\ abs\ (int\ i)\ =\ if\ i\ <\ 0\ then\ -i\ else\ i\ ;$$

in the above case, a call abs(1.0) would use the first function, because the input is a float and the first function expects a float, and thus the function call returns a float. However, a call to abs(1) would, with the same reasoning, call the second function and thus return an int. In the case of primitive types as parameters, implicit casting is used if required. E.g. if the function abs is called with a byte parameter, as no abs(byte) function is available, then abs(int) is used instead of abs(float), as the byte can be cast to int with less casting (int is more restrictive than float). If more than one function applies for a given call, e.g., because their names and the types of the parameters are the same, then the functions are not well-formed; For each function call, there must be exactly one function that applies. This is the case since each expression (here the function call) can only result in one value.

5.4.2 Recursion

While functions themselves are a stand-in for expressions, they do allow for expressions that are not possible without them. That is, they enable recursion; Assume the following function declaration:

$$int\ fibonacci\ (int\ n) = if\ n\ ==\ 0\ then\ 0$$
$$else\ \big(if\ n\ ==\ 1\ then\ 1\ else\ fibonacci\ \ (n - 1) + \ fibonacci\ (n - 2)\big);$$

The function calculates the nth Fibonacci number for any natural number n. Note that the function's expression contains calls to the very same function. Such a recursive function, in most cases, cannot be replaced by an expression without a function call, therefore, allowing for calculations not possible without functions.
As a comparison, recursion allows for calculations similar to while-/ for -loops in imperative programming languages; They allow for "arbitrarily long" calculations; While most other expressions are guaranteed to finish in a finite amount of time, recursive functions calls are more powerful, they can run calculations that may not be guaranteed to finish.

5.4.3 Package and Modules

We have function definitions. What happens if we have a bigger model? Maybe we want to import collections of functions of another model. Maybe we want to import collections of functions of multiple models to reuse. What if some of those functions have the same name? This hinders reuse, as one may not always know what a specific function does. Additionally, the model becomes ambiguous, which in turn results in problems regarding well-formedness checking. One could mitigate the issue by writing longer, more likely to be unique names. However, the issue, while less prevalent, still prevails. Additionally, very long function names may severely reduce the readability of the expressions. In this option's stead, functions offer a qualified name. The function's name of its definition itself is one word, this is the unqualified name. The qualified name, however, consists of multiple words concatenated with dots in between. E.g.:

$$\text{mainPackage.subPackage.module.function}$$

The qualified name is always unambiguous, allowing one to identify a function. As the qualified name can be used for identification, we allow for the unqualified name to be ambiguous. Structuring the function collections in modules and packages then allows the use of the short unqualified name in most cases, such that one has the readability advantage of the short names, but also the advantage of unambiguous identifiability of the long names. A rough overview:
- Every function is defined in a module.
- Every module lies in a package.
- A package may lie in another package; this is not recursive.

Thus, functions, modules, and packages form a tree, with packages being in the root and consecutive nodes, the modules being on the second to last layer, and functions being the leaves. Modules and packages have (unqualified) names. The qualified name of a function is the

concatenation of the unqualified names from the package root of the tree to the function. While packages do not contain any information in addition to their name and the set of modules and sub-packages, modules have to have additional information to allow for the use of functions using their short, unqualified name.

```
package a.b.c;
import a.b.c.module1;
import a.b.module2;

montifun module3 {
  int function1( ) = function2( ) + 4 ;
}
```

Each module starts with a package declaration, followed by arbitrarily many import statements, and the module definition. In the example, the package of the module is a.b.c. The module imports the two modules a.b.c.module1 and a.b.module2, making their functions available in the current module, thus, the functions in this module may use the imported functions by using their unqualified or qualified names. The name of the module is module3. Its fully qualified name is a.b.c.module3. The body of the module is defined after the modules name, which is written after the keyword montifun1 It is enclosed by curly brackets and contains the functions' definitions. In the example, the function definition of function1 uses function function2, which has not been defined in this module, but rather in one of the imported modules.

5.5 Statecharts

Statecharts are a means to model a system's state space and behavior via states and transitions between these states[27]. A statechart models how a system element reacts to a given input, i.e., what output it produces. Using statecharts, system engineers can model high-level specifications or define low-level system functionality. Accordingly, statecharts specify the behavior of composed systems or define the behavior of atomic system elements, i.e., leave nodes in the system composition hierarchy.
Statecharts are one of many manifestations of the same fundamental concept of state-based behavior descriptions. The different manifestations can be used for various purposes, such as specifying the state space of an object or recognizing sequences of messages. In SpesML, a statechart models a system's behavior and can be made executable. That is, they model the output of a system in relation to its input and serve as the link between a system's state and the behavior of its components.
While statecharts in their original definition were intended to define a system's structure and behavior, statecharts in the SpesML are primarily used to specify system behavior. The structure of a system instead emerges from the composition of its system elements, which is essentially a composition of statecharts. Nonetheless, statecharts extend finite state machines with hierarchy, which can be used to structure the system's state space.
The essential elements of a statechart are states, transitions, and actions. The states define the system's state space, the transitions describe the change of a system's state, and actions describe a system's behavior through operational code.

5.5.1 States

The states of a statechart define a system's state space through abstract names. A state is a situation in which a system awaits an event or performs an internal activity[28]. A system can be in some state, called its current state, and transition from its current state to other states. Typically, a system remains in a state for some time.
A state has a name that is unique within the statechart. Furthermore, a state may contain an entry action, an exit action, and a do-activity. Entry and exit actions execute when the system enters, respectively, exits the state. The do-activity executes when the system remains in the same state for some time. An excerpt exemplifying the notation of a state is shown in Figure 5-3.

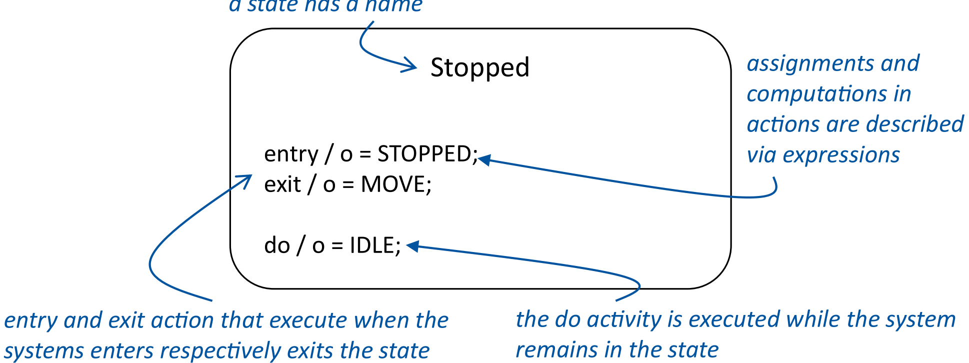


Figure 5-3: A state has a unique name and may contain an entry action, an exit action, and a do-activity.

A statechart has at least one initial state. In this state, the system begins its lifecycle. A statechart may have multiple initial states, representing different forms of system initialization. However, statecharts in the SpesML do not have a final state. Instead, systems run indefinitely; their execution never stops.
A system's states do not solely define its state space. Instead, a system description may further introduce system variables that extend the state space. Variables are more flexible than states and can be used to describe large state spaces easily. The actual state of a system results from the current state and current variable assignment. In the following, we abstract from this distinction and include the current variable assignment with the term current state.
The behavior of a system is defined on the basis of its state space. That is, a system's behavior is not only defined with respect to its current input but also with respect to its current state. As a system changes its states dependent on the received inputs, its state abstractly reflects the history of received messages. Thus, induced in the state space of the system, the behavior of a system also depends on previously received messages.

5.5.2 Transitions

A transition defines the change of the system's state and the system may exhibit behavior (i.e., output a message) in reaction to a state change. The description of a transition contains the transition's source state, target state, guard, stimulus, and reaction. An excerpt exemplifying the notation of a transition is shown in Figure 5-4.

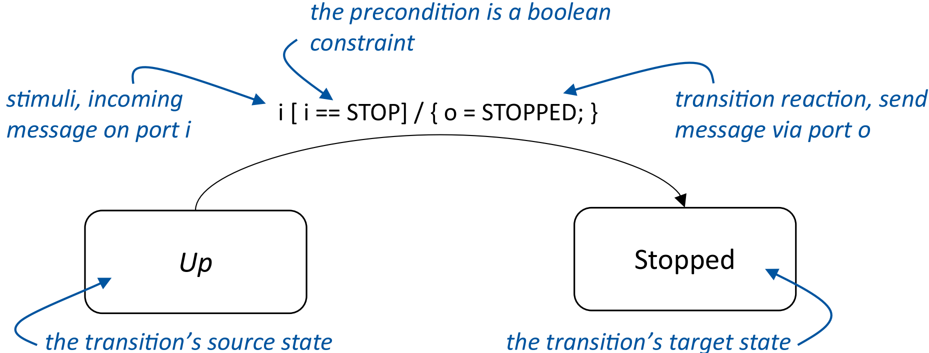


Figure 5-4: A transition leads from a source state to a target state and has a stimuli, a guard, and a reaction.

Transitions may execute. When a transition executes, it executes its reaction after exiting the source state and before entering the target state. A transition may only execute if the system's current state is equal to the transition's source state. The guard of a transition further constrains the execution of the transition. A guard can restrict the execution of the transition to certain properties of incoming messages or variable assignments. Subject to these constraints, the occurrence of a transition's stimulus triggers execution of the transition.

We call a transition enabled if its stimulus occurs while the system is in the transition's source state and if the transition's guard holds. If there is any situation in which multiple transitions are enabled, then the statechart is nondeterministic.

In SpesML, the stimulus is either the progress of time or the event of an incoming message. Which stimuli are available depends on the specification's time variant. In timed specifications, a transition may execute in reaction to the progress of time. In asynchronous specifications, a transition may execute in reaction to an incoming message. Thus, in synchronous specifications, each transition represents the progress on time, whereas in timed asynchronous specifications, multiple transitions may execute between two points in time.

5.5.3 Actions

Actions complete the behavior description of a statechart. They describe the system's behavior in reaction to a stimulus, subject to the system's current state. That is, they describe the behavior a statechart exhibits in reaction to the execution of a transition or the behavior the statechart exhibits when entering, exiting, or remaining in a state.

Entry and exit actions are used to describe the common behavior of entering, respectively, exiting a state, such as signaling state changes, writing log outputs, and initializing and resetting variables. Entry and exit actions do not enable the description of any behavior that cannot be described by transition reactions alone. Rather, they are executed in the context of a transition's execution and serve as a shorthand notation for adding the same calculations to all incoming and outgoing transitions. Hence, entry and exit actions can be moved to outgoing, respectively, incoming transitions.

In the SpesML, actions are procedural behavior descriptions. They consist of statements, which are either assignments or control structures. Assignments store values in variables or send messages via ports. Control structures govern the execution of statements, which otherwise execute in sequence. Computations in statements are described in detail by expressions (see Section 5.3). Expressions are also used to specify the guards of transitions.

5.5.4 Hierarchy

Statecharts can be hierarchically structured; that is, a state can have sub-states, or rather, a sub-statechart. The division into sub-states can be used to structure a system's state space and prevent state explosion. A hierarchical subdivided state can contain common parts of entry and exit actions or serve as a source or target of common outgoing, respectively, incoming transitions. Furthermore, state hierarchy may be used methodologically, e.g., by first defining an abstract state and refining it later on dividing it into sub-states.
A hierarchically subdivided state, like any other state, has a name and may have an exit action, an entry action, and a do-activity. Transitions can cross state hierarchy levels, leading from any super-state to any sub-state and vice versa. When a transition executes, the system exits and enters any state along the hierarchy, sequentially executing exit and entry actions.
In the SpesML, hierarchical subdivided states are considered complete. That is, a system in a hierarchical subdivided state is always in one of its sub-states. Thus, if a system enters a hierarchical subdivided state, then it enters one of its sub-states.
A sub-state can be an initial state. Such a state is not an initial state to the whole statechart but an initial state of its super-state. If a transition target's a hierarchically subdivided state, then the system enters its initial sub-state once the transition executes. However, if the transition directly targets a sub-state, then the system would instead change to that state.

5.6 Application of the Universal Interface Model

This section explains how the universal interface model (UIM) concepts are implemented in the SpesML workbench based on SysML. Hence, the following sections first provide a general mapping of the UIM concepts to SysML. Afterwards, a section shows the SpesML metamodel for the UIM, which is based on SysML.

5.6.1 System Element

System elements (representing functions, logical components, or technical components, respectively) are represented as SysML blocks and parts. In the universal interface model (see Section 3.3), we did not distinguish between the definitions of system elements and their usage. As SysML makes this distinction, there is no 1:1 mapping of the concept of system element.
A SysML *block* defines a type of system element. With this, it defines the syntactic interface and the behavior of its usages. Each block owns at least one proxy port. Each proxy port defines a sub-interface of the system elements modeled by the block (see later sections).
In SysML, a *part property* represents the usage of a block in a certain context. A block is a composed system element if it owns at least one part property. Then, it is called *composed*. Otherwise, it models atomic system elements and is called *atomic*. The part properties model the sub-elements of the system elements. Each part has a name and an associated block as its

type. The name of the part represents the name of the sub-element. A part property represents a system element in a specific role (e.g., the window-lifter-controller or the back-window-controller of a car). Communication relationships (via SysML connectors) are defined on the level of parts. If two distinct system elements have the same interface and behavior, this is represented by one block and two distinct part properties. Each part property is an instance of the block in a different role.

5.6.2 Syntactic Interface

In the universal interface model, the syntactic interface of a system element consists of named, typed input and output channels. While we did not stress this, a syntactic interface can be split into sub-interfaces with input and output channels. The union of all sub-interfaces gives the syntactic interface.

Regarding the mapping to SysML, sub-interfaces of system elements are modeled with SysML *proxy ports*. Each proxy port is typed with a SysML *interface block* that describes the sub-interface's input and output channels.

By default, all the channels of a sub-interface defined by a proxy port are output channels. To represent the inverse of the syntactic sub-interface, the attribute "*Is Conjugated"* of the proxy port is set to *true*. A ~ symbol visually displays this as a prefix to the name of the proxy port. Thus, if the conjugated property of a proxy port is set to true, then it models a sub-interface solely containing input channels.

5.6.3 Channels

A channel in the universal interface model is represented by a SysML *flow property* owned by an interface block. The name of the flow property defines the channel's name. The type of the flow property models the channel's type. In SpesML, the direction of all flow properties must be set to *out*. Whether a channel is used as an input or an output channel by a system element is determined by the proxy port that is typed with the interface block containing the channel. In SpesML, each interface block can contain an arbitrary number of flow properties.

5.6.4 Data Types

Recall that the universal interface model defines a syntactic interface containing channels with an associated type (or sort). SysML value types represent these types.

5.6.5 Composition

System elements in SysML are composed by defining a block for the composed system element and adding the system elements as part properties. This amounts roughly to parallel composition as defined in the theory (see Section 3.3).

In order to allow the system elements to interact or to define a feedback channel, their proxy ports need to be connected with SysML *connectors*. Depending on which ports are connected, this represents the concepts of sequential composition, feedback, and channel hiding in the universal interface model.

Some rules need to be respected when connecting proxy ports. Two ports may only be connected via a connector if they have the same types and their directions match. The directions of any two ports match if one of the following conditions is satisfied:

1. One port is owned by the block, and the other port is owned by a part, and both ports are conjugated.
2. One port is owned by the block, and the other port is owned by a part, and both ports are not conjugated.
3. Both ports are owned by parts, and one of the ports is conjugated, and the other port is not conjugated.

The first condition corresponds to the case where messages are transmitted from an input port of the block to an input port of a part. The second condition corresponds to the case where messages are transmitted from an output port of a part to an output port of a block. The third condition corresponds to the case where messages are transmitted from an output port of a part to an input port of a part.

5.6.6 Example

Below, we show an example of the SysML model elements used for mapping the universal interface model. The example shows a strongly simplified version of the running example Window Lifter. The used SysML model elements are annotated in the diagrams. The interface blocks typing the proxy ports are only displayed by name in the IBD (see Figure 4-1). A definition is visualized in a BDD (see Figure 4-2), including their flow properties and the used value type.

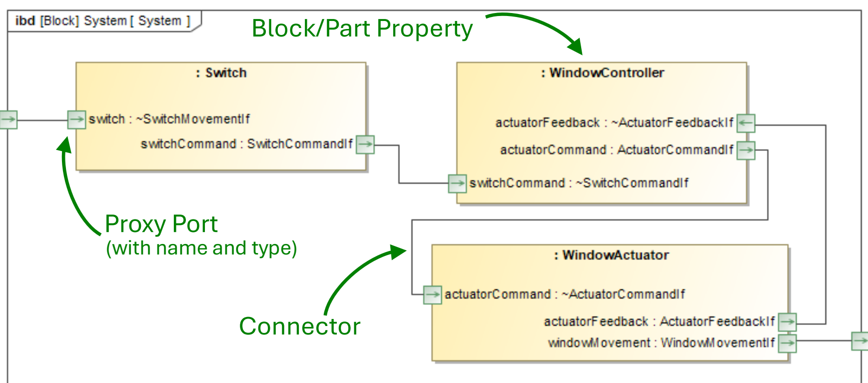


Figure 5-5: Simple Example for the used SysML model elements

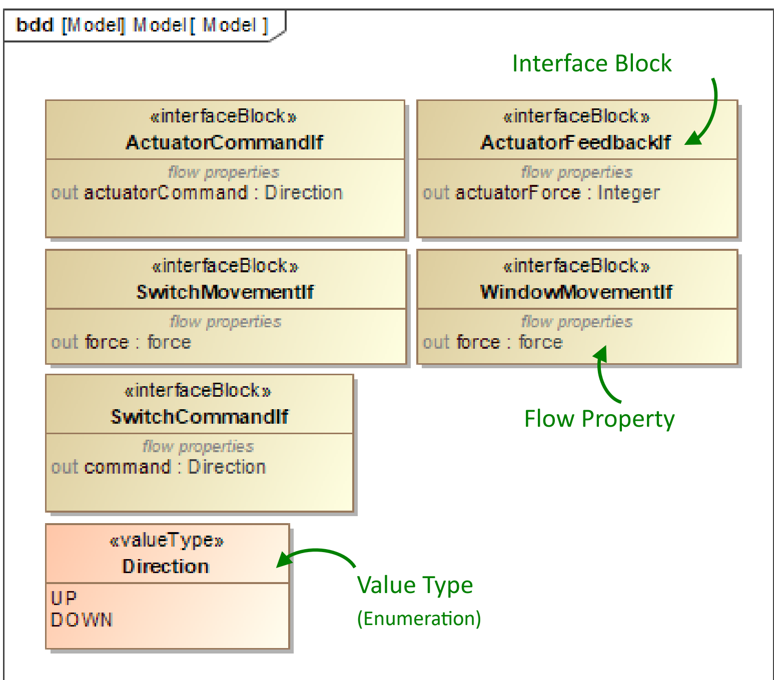


Figure 5-6: Used Interface Blocks and Data Types for the Example

5.6.7 The Spes ML Metamodel for the UIM

In the last section, we outlined the general mapping from the universal interface model to SysML. For SpesML, we do not use the SysML elements directly but instead build a profile that is based on SysML. Building a profile consists of defining model element types (called stereotypes in SysML) that are derived from SysML elements and may add additional properties. SpesML introduces model element types specific to the different views. For example, in SpesML, there is not just a single element type *Interface*. Instead, there are element types: *Functional Interface*, *Logical Interface*, and *Technical Interface*. Figure 5-7Figure 5-7 shows an overview of the SpesML element types for the universal interface model, however, without showing the element types for all viewpoints. In the following, we explain the model elements one by one.

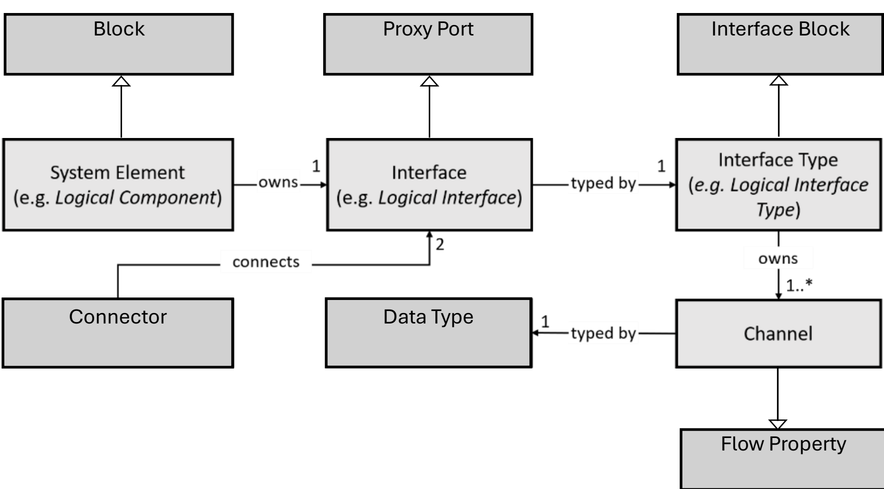


Figure 5-7: Overview of the model elements of SpesML. Elements in brown color are SysML elements, and elements in grey are elements derived from SysML elements.

Interface: A system element (such as a logical component from the logical viewpoint, a black-box or white-box function from the functional viewpoint, or a technical component from the technical viewpoint) owns one or more interfaces. In the different viewpoints, the interfaces are respectively named *Logical Interface*, *Functional Interface*, and *Technical Interface*. An interface can be either a source interface (messages are flowing out of the owning system element) or a target interface (messages are flowing into the owning system element). To mark an interface as a target interface, the interface must be conjugated.

Interface Type: An interface is always typed. That means it is associated with an interface type. Again, in the different viewpoints, an interface type is respectively termed "*Logical Interface Type*", "*Functional Interface Type*", and "*Technical Interface Type*". An interface type must always be named and may only contain channels as sub-elements.

Channel: The interface type describes the input and output of a system by means of channels. Therefore, each interface type owns one or more channels. Channels are always named and represent unidirectional communication from the source interface to the target interface. Although a channel has a direction, this direction must always be set to "out". This restriction makes it possible to clearly see which information is flowing in which direction. In order for a channel to represent an input, the property "Is Conjugated" needs to be set at the interface.

Data Type: Each channel is typed with a data type, which describes what kind of messages are exchanged via this channel. Note that data types must not be confused with interface types. While interface types only type interfaces, data types only type channels (which in turn are owned by interface types).

Connector: Two interfaces may be connected using a connector, which describes that messages can flow between the interfaces.

Strong Causality with respect to some output channels can be introduced by delaying the outputs of an interface. An interface can be delayed by adding a value to the *Initial Sequences*

for Delay property. The Initial Sequences for Delay property can only be given a value for interfaces where the conjugated property is set to false, i.e., only output channels can be delayed. If the property is given a value in an interface, then all channels represented by the port are delayed.

Composition: System elements are composed by connecting their interfaces with connectors. Two interfaces may only be connected via a connector if they have the same types and their directions match. See the rules in the previous section for when the directions of interfaces match.

5.6.8 Well-formedness Rules

The UIM also defines a broad set of general well-formedness rules (WFR).

General Rules:

WFR-1: All SpesML Elements that may have a name must have a name.

- Reason: In combination with textual specification languages, referenceable model elements should have names. If a model element is not supposed to be referenceable, then it usually should not have a name either.

Enumerations and Types:

WFR-2: Enumerations must not have attributes.

- Reason: The purpose of enumerations within SpesML is to create a data type for listing elements. Attributes in an enumeration come with an unclear interpretation within the model.

WFR-3: Enumerations must have at least one enumeration literal

- Reason: Enumerations without enumeration literals are useless.

WFR-4: All attributes of value types must have a type, which is a value type, an enumeration type, or a primitive type.

- Reason: We use a strong type system to minimize the number of runtime errors already at design time.

Flow Properties:

WFR-5: Interface blocks must not contain any elements different from flow properties.

- Reason: By definition, an interface block only consists of flow properties in the SpesML.

WFR-6: All flow properties must have a type, which is a value type, an enumeration type, or a primitive type.

- Reason: See WFR-4.

WFR-7: Flow properties must have the direction out.

- Reason: The actual direction of the channel represented by the flow property is defined by the value of the 'Is conjugated' property of the proxy that is typed with the interface block defining the flow property. Thus, the value of the direction is irrelevant. As we cannot disable the direction property, the default value is defined to be out.

Ports:

WFR-8: All proxy ports must have a type, which is an interface block.

- Reason: This is to increase the model quality. An untyped proxy port does not have a specified syntactic interface.

WFR-9: Only blocks that do not own parts may contain value properties.

- Reason: Value properties represent the internal state of a system element. However, if a system element is composed of other system elements (and thus, the block owns parts), its state is given by the states of its sub-elements.

WFR-10: Every block must own at least one proxy port.

- Reason: A block without a proxy port has no interfaces and, therefore, represents a system element that is not interacting with the environment or other system elements.

WFR-11: Every value property of every block must have a name and a type, which is a value type, an enumeration type, or a primitive type.

- Reason: A value property can only be referenced from a state machine when it has a name. To verify that expressions in state machines are correct, the value property needs a type.

WFR-15: Every part of every block must have a type, which is a block.

- Reason: See WFR-4.

WFR-20: Initial sequences of ports of blocks may only be defined for ports that are not conjugated.

- Reason: By convention, initial sequences may only be defined for output channels.

WFR-27: All incoming ports of a part must be connected.

- Reason: This prevents using parts without providing input.

WFR-28: All out ports of a part should be connected.

- Reason: This reduces the risk of forgetting to use the outputs of a part.

References

[27] OMG (2011). Unified Modeling Language (OMG UML), superstructure.

[28] B. Rumpe (2016): Modeling with UML: Language, Concepts, Methods. Springer International.

[29] Schätz, B., Pretschner, A., Huber, F., & Philipps, J.: Model-based development of embedded systems. In *International Conference on Object-Oriented Information Systems*, pp. 298-311. Springer Berlin Heidelberg, 2002.

[30] Oracle America, Inc (2023) JSR-395 Java SE 20.

[31] ISO/IEC (2020) ISO/IEC 14882:2020 Programming languages - C++.

Andreas Bayha
Sebastian Bergemann
Wolfgang Böhm
Jan Philipps
Andreas Vogelsang
Sebastian Voss

6 Usage of the Language Constructs in the SPES Methodology

As described in Chapter 3, systems modeling in the SpesML methodology is based on a division of architectural models into views and granularity levels. View groups models to address specific concerns framed in viewpoints. granularity layers are used to treat selected system components as subsystems, each with their own modeling activities according to SpesML. The models of different views in different granularity layers are connected through tracing relationships. In addition, well-formedness rules abstractly define the set of valid SpesML models.
This chapter describes the four SpesML viewpoints - requirements, functional, logical and technical viewpoint - in detail (Sections 6.2 to 6.5). Common to the the functional, logical and technical viewpoints is that they each include a viewpoint-specific model of the system context. Context modeling is described in Section 6.1. The modeling of subsystems for system development at different granularity layers is described in Section 6.6. The chapter concludes with Section 6.7, which describes the way context models, viewpoints and subsystems are brought together through tracing relationships.

6.1 The system and its context

A SPES system model (see Section 4.3) also defines the environment in which it operates. The environment is part of the system definition, meaning the system is always embedded in its so called "*context*" and communicates and interacts with specific context elements.
Elements of this context can be of various types. In SpesML, each element that is identified as a context element must have a direct interface with the System under Development (SuD), we call this a relation of first grade. This interaction is defined through an interface that follows all principles of the universal interface model (UIM) (compare LINK: Modeling Theory). This enables for an indirect access to further context elements that are connected through first grade elements indirectly.
In general, we distinguish between two different types of context:

1. **Knowledge Context:** This type of context may contain standards, regulations (e.g. ISO26262, DO-178B) and other forms of documentation relevant for the SuD. This type of context influences the system at design time.
2. **Operational Context:** An operational context may contain various elements that interact with the system during runtime.

In the following, we focus on the operational context and especially on its elements and relations to the SuD.

6.1.1 Elements of the operational context

The operational context consists of the following system elements

- System under Development (SuD): The system that is currently being engineering, resp. in focus of an engineering step.
- Context Elements: System elements as described below.
- Interfaces between context elements and the SuD.

In SpesML each element is modeled by applying the UIM, i.e. the elements are treated as a black-box that have a syntactic and a semantic interface. When modeling the operational context on system level, the following modeling guidelines must be followed:

- A context model of each view shall contain exactly one SuD.
- The SuD on different views remains the same, meaning the SuD does not change, for instance, with respect to the scope of the SuD.
- Each context element is modeled via the UIM and shall have at least one channel to the SuD, being a context element of 1st grade.
- Each context element shall be of one of the following types:
  - **Human Actors as Context Elements:** These elements describe human interaction with the system across a human-computer interface
  - **External Systems Context Elements:** These elements describe system-to-system communication
  - **Physical Context Elements:** One specific External System Context Element could be a Physical Context Element. These elements may represent the impact of physical processes onto the system (e.g. hydraulics, gravitational forces, …). Such elements, however, are external system context elements as they also apply the UIM. They can be modelled in the same way as physical components that are part of the SuD. Such elements have further properties on their interface description (cf. chapter 4.3).

Surely, there are a lot of elements that do not have relations (even not related relations through other context elements) with the SuD. These elements do not at all interact with the SuD and thus are not context elements of 1st grade, meaning being outside of the context border and thus not considered for context (and SuD) modeling.
In general, context elements can be directly connected to the SuD or transitively, meaning through composition with 1st grade context element. We call these elements context elements of 2nd grade. The operational context conceptually even includes 2nd grade elements through the consequent application of the UIM. Thus, 2nd grade element are influence the SuD behavior through the interfaces of the 1st grade elements. In the SpesML project we explicitly focus in 1st grade relations. Thus, the identification of context elements of 1st grade is a first abstraction in systems modeling, helping to better understand the necessary system interfaces. Figure 6-1 illustrates the various types of context elements.

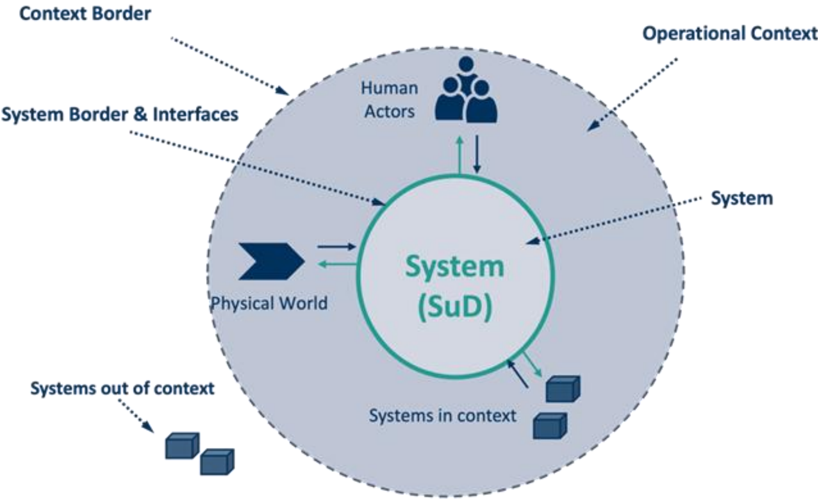


Figure 6-1: Context and System

6.1.2 Dimensions in context modeling

There is a horizontal and a vertical dimension of modeling context elements:

- *Horizontal:* As SpesML offers different viewpoints (and their corresponding views) on the system, this may result in different views on the context as well. However, the same SuD is considered in the functional, the logical and the technical models on system level.
- *Vertical:* If the scope of the SuD changes, by using the concept of granularity layers (see Section 4.1), this consequently, results in a different operational context for the new SuD, meaning different context elements.

In the following, we describe both context dimensions in detail.

6.1.2.1 Horizontal Context Dimension in SPES

Each viewpoint in the SPES framework allows to model various concerns a stakeholder may have (compare chapter 4.1).
This means that without changing the SuD, each viewpoint provides a set of predefined models and model elements to model the SuD and, as a consequence, also its context elements, at a certain level of abstraction. Figure 6-2 illustrates a 1st grade element in the functional view, namely the Context Function A.

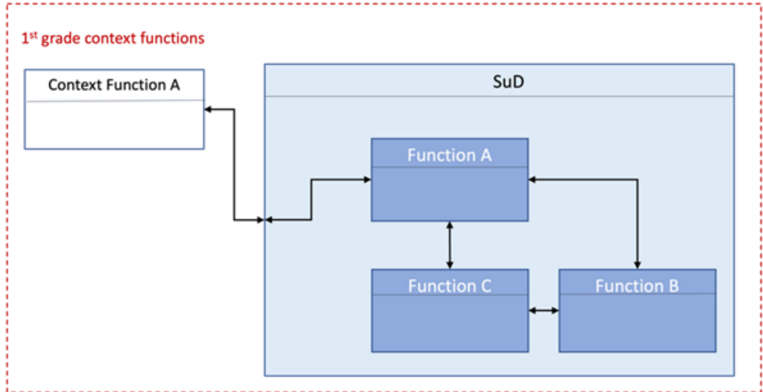


Figure 6-2: Context elements of 1st grade

In the following, we provide an overview of the SPES viewpoints and their dedicated context elements that appear, without changing the system scope, i.e. there is a clear notion of what is part of the SuD and what is considered the context of that system.

Functional Context. The context described in the functional view focuses on functions (i.e., chunks of behavior), representing a functional perspective of the SuD context, rather than entities. The system boundary and the system context are specified in the *Functional Context Diagram*. This diagram contains one specific element that represents the entire SuD and additional elements modeling functional context elements, named *context functions* outside the SuD. These context functions describe the functional interactions between the SuD and the context.

Figure 6-3 illustrates different context functions in the functional view, here representing the two different types of context elements, a Human Actor and an External system or an External System Element. As discussed above, the behavior of context functions is modeled by applying the UIM (chapter 4.3) i.e. the elements have a syntactic interface and an interface behavior. The black-box models of context elements include all necessary syntactical and semantical information. A more detailed modeling of the functional view of a SuD can be found in Section 6.3.

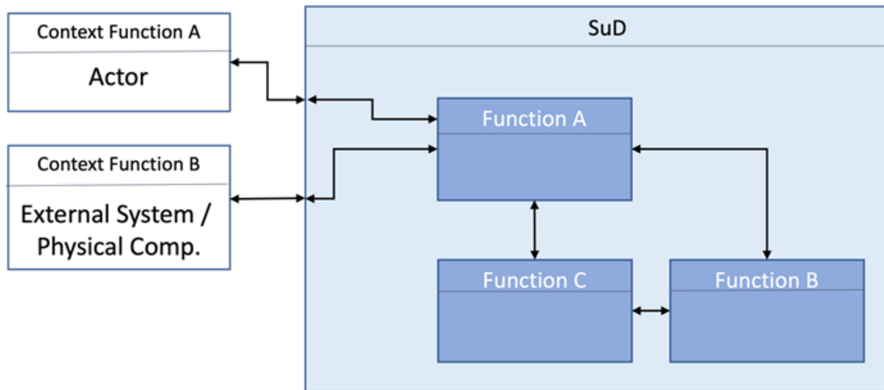


Figure 6-3: Example of Context functions interaction with the SuD

Context in the Logical View. The logical context focusses on modeling the environment in which the SuD shall operate on a logical level. Therefore, context elements in the logical view are called *context systems.* The system boundary and the system context are specified in the *Logical Context Diagram*. Again, context systems can be of various types, e.g. human actors, that interact with the SuD and external systems or physical context elements. One main objective in modeling the logical context is also to specify the logical interface of the SuD according to the abstraction of the logical view (Section 6.4). Context systems of whatever type are not "stand-alone", moreover, have a relation to context functions or may emerge in the logical context view (e.g. Context System Y). These elements and their relations to other viewpoints are described in Section 6.7.

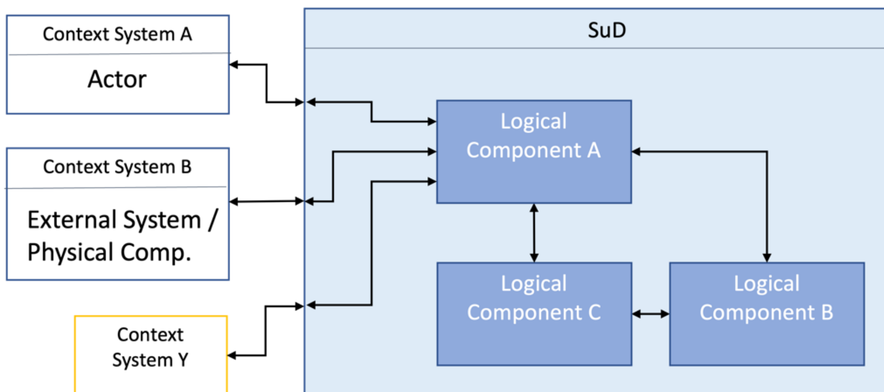


Figure 6-4: Example of context systems in the logical view

Figure 6-4 illustrates three different context systems in the logical view interacting with the SuD in a relationship of 1st grade. Again, comparably to the functional context, the behavior of

context systems in the logical view is modeled by applying the UIM, i.e. context systems are having a syntactic and a semantic interface.
Technical View. Similar to the functional and logical view, a context can be modeled within the technical view. The technical context focusses on modeling the current SuD in relation to technical relevant context elements. The technical system boundary and the technical context are specified in the Technical Context Diagram. We call elements of the technical context technical context systems. In general technical context systems are comparable to context systems from the logical viewpoint with specifically defined relations between them (compare Section 6.7). Moreover, also in the technical view there may emerge technical context systems with no further relations to context elements from the logical view, e.g. technical context system X in Figure 6-5 that may be represent technical components that may be implicit in the communication channel of the logical view.

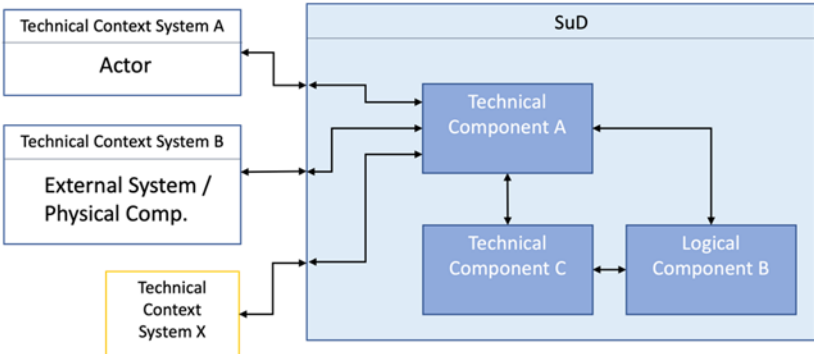


Figure 6-5: Example of technical context system

Relations between Context Elements. Model elements in different views are in relations to each other. (see Section 6.7), meaning that parts of a model element in one view are “implemented” or “realized” by elements of another view. This holds not only for elements of the SuD, but also for context elements, as these elements represent the context of the same SuD just on a different level of abstraction.
Figure 6-6 illustrates the relations between context elements of the different views, representing system design recommendations in SpesML: In the horizontal dimension a context function from the functional view (e.g. *Context Function A*, is related to a context system, namely *Context System A*. The tracing relations between context functions and context systems are analog to the relations between systems functions and logical components. This may result in a situation, where a *Context Function A* may be realized by more than one *Context Systems* in the logical context.

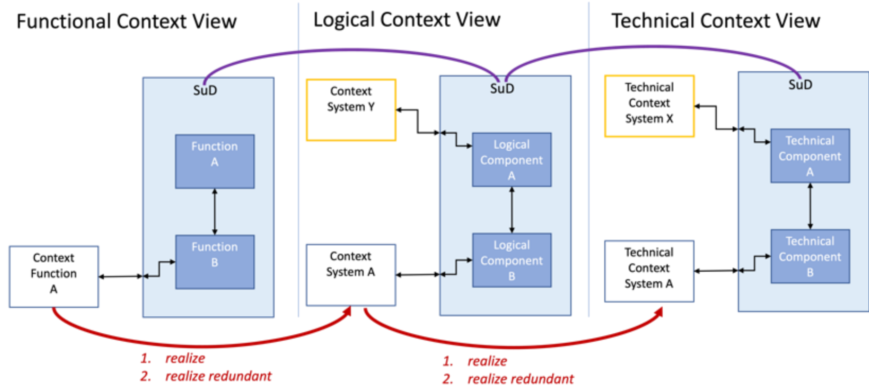


Figure 6-6: Relations between context elements (horizontal dimension)

We can make use of the same principles not only because context elements as well as SuD elements rely on the UIM. Context systems, thus, *realize* context functions or if context systems provide redundancy relations, we can make use of a *realize redundant* relationship. The argumentation is comparable to elements relations of the SuD between functional and logical view.

Note that – as a design recommendation – we strongly recommend modeling the structural context in the logical view in such a way that there is a n:1 relation between context functions and context systems, corresponding to a white-box decomposition in the functional view.

Furthermore, there may be context systems in the logical view that do not have a relation to context functions (e.g. Context Systems Y in Figure 6-4). These context systems may be added due to implementation or technical reasons. Also in the technical view there may such technical context systems with no relation to logical context systems (e.g. technical context systems X) for the same reason.

A context system (e.g. Context System A) can be further traced to technical context elements of the technical view. We make use of the same tracing relations, as already know from SuD elements, namely either *realize* or *realized redundant* respectively. At the highest level of abstraction, we again recommend a 1:1 relationship between context systems and technical context systems. Vertical dimensions in SPES

Across granularity layers, e.g. from a system level to subsystem level, the scope of the SuD changes. In SpesML, we allow the technical components in the technical view to be viewed as independent subsystems which are modeled on the next granularity layer.

In general, any technical component from the system level (1st granularity layer) can be further engineered on the next level of granularity, namely the sub-system level (2nd granularity layer). The technical component from system level becomes the new SuD on sub-system level. The engineering can follow any process and methodology, including SpesML. The new SuD consequently provides a new scope, including new context elements, that may – but not necessarily must – result from: First, previous context elements of the system level (with a 1st grade relation). Second, technical components that have a direct relation to the technical

component that is now the new SuD on sub-system level and third, new technical context systems.

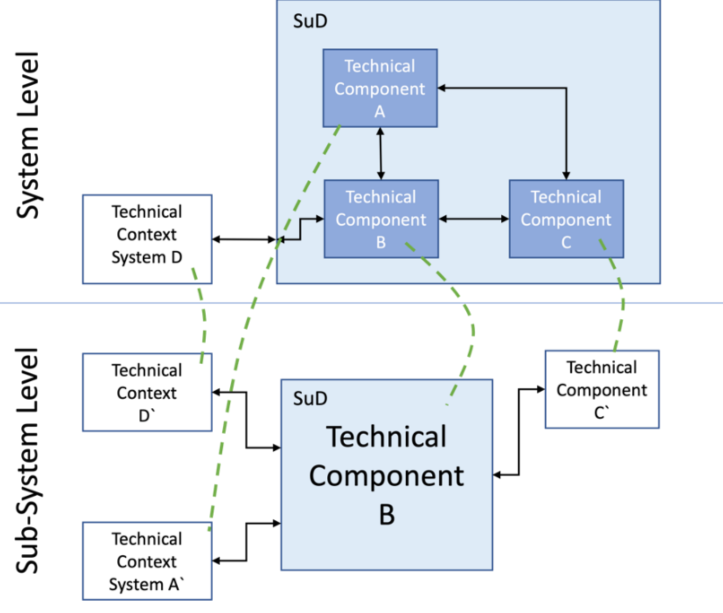


Figure 6-7: Relations between context elements in the vertical dimension

Figure 6-7 illustrates such a new view of *Technical Component B* of the SuD to be further engineered on the next granularity layer. In such a situation, *Technical Component B* has a new scope on the next granularity layer, meaning that this component becomes the new SuD.
From a context point of view, with a change of scope, meaning with a new SuD, the context elements change as well as the operational context is always defined with a reference to the SuD. In general, all technical components from the super-system with a communication channel to the Technical Component B become context elements on subsystem level (e.g. *Technical Component A* becomes a *Technical Context System A`)* on the level of the subsystem). In addition, all 1st grade components of the super-system context with a direct communication channel to *Technical Component B* are in the context of the subsystem (e.g. *Technical Component D* becoming *Technical Component D`).*
The new SuD can be modeled according to the SpesML methodology on the new granularity layer modeling all views defined in SpesML. In this case we obviously have relations between the context elements of the different views and relations between the views of the super-system and the views of the subsystem. Figure 6-8: Context relations across views and granularity layers illustrates this as an example. We use the same relation types, namely *realize* and *realize redundant* between context elements of different views, as described in Section 6.2. For instance, *Context System A* is realized by *Technical Context System A* (compare Figure 6-8).

Surely, relations to the functional view are possible on all granularity levels (see Section 6.7 for details).

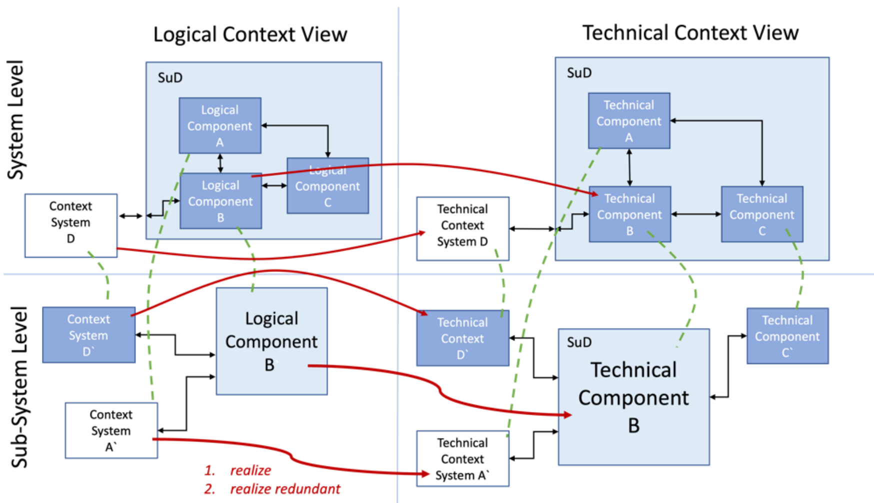


Figure 6-8: Context relations across views and granularity layers

Furthermore, different engineering approaches are supported when applying SpesML. A top-down engineering approach may be used for the development of a new system, whereas a bottom-up approach is applied when integrating existing (re-used) components into an overall system architecture. In the re-use case, it is sometimes not known at design time of the subsystem in which context it will be used when it is integrated into a super-system. Using assume/guarantee descriptions on the interfaces of such subsystems will allow to re-use these subsystems in various contexts, where the assumption reflects context behavior expectations and guarantees provide information of the behavior of the subsystem if the behavior assumptions hold.

6.2 Requirements Viewpoint

Requirements are an over-arching topic for SPES architectural models. Requirements – at different levels of precision – link the needs of stakeholders and the obligations arising from the development context with the architectural elements of the functional, logical, and technical viewpoints.
It is common to differentiate requirements according on the aspects they address. In SpesML we roughly distinguish between:

- Capability requirements, concerning stakeholders' demands for the system to be able to perform a certain functionality.
- Functional requirements, concerning the input/output behavior of functions and components.

- Quality requirements, concerning software and product quality characteristics such as the well-known “ilities”.
- Constraints, concerning side conditions to be observed during development

Some of these categories can be further refined. For instance, there are different kinds of constraints, depending on whether it is a physical entity to be constrained (e.g., its weight), architectural design decisions (e.g., notational conventions), or the development process (e.g., the need to for certain analyses or tests).
Thus, requirements categories form a hierarchical taxonomy, as shown in Figure 6-9.

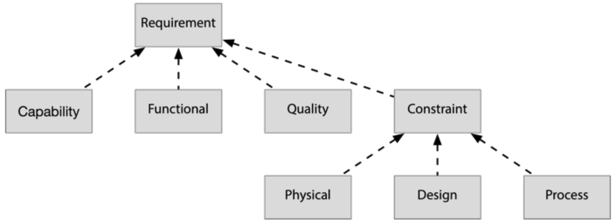


Figure 6-9: Requirements categories in SpesML.

The requirements taxonomy might seem a bit overwhelming at first. Indeed, for systems with limited size and complexity it is reasonable to just use the generic “requirement” category. More complex situations benefit from the higher expressibility and the stronger well-formedness conditions gained when differentiating categories.
In contrast with the elements of the functional, logical or technical viewpoints, requirements as such are not given a formal semantics. in SpesML. One reason is that for most categories, the Universal Interface Model - on which the formalization of the other viewpoints is based - is not a suitable mathematical basis. The second reason is that even for functional requirements, where such a formalization is intuitively possible, formalized requirements have little use in SpesML. Instead, the system functions of the functional viewpoint can be seen as formalized behavior descriptions of a group of related functional requirements. In practice, this gives a better understanding of the intended functionality than isolated partial behavior specifications.
Requirements are connected to architectural elements or to other requirements through various tracing relations, again addressing different aspects, for instance, whether an architectural element is intended to *satisfy* a requirement or whether a requirement is *derived* from another requirement.
Although requirements themselves in SpesML have, for the reasons discussed above, no mathematical semantics in the underlying Universal Interface Model (UIM, see Section 4.3), it is still possible to derive methodically useful properties at the level of the traceability graph consisting of linked requirements and architectural elements. For instance, it is possible to discover unimplemented stakeholder requirements or unmotivated derived requirements; it is also possible to construct lists of verification obligations arising from the graph structure.

6.2.1 Model Elements

Given the absence of a mathematical semantics for requirements in SpesML, requirements are represented, similar to the requirements of SysML, as objects holding textual statements. Obviously, the representation may be more restricted than natural language alone. In increasing degree of formality, one might envision:

- *Guidelines*: Even for free-form natural language, it is recommended to follow a style guide, such as the INCOSE recommendations [INCOSE, 2019]. Most development organizations will curate a collection of such guidelines and make them obligatory for their development projects.
- *Constrained content*: For requirements allocated to, for instance, a system function in the functional view, it is desirable that the requirement references and constrains phenomena in the scope of the model element it is allocated to, such as ports or internal state variables.
- *Sentence templates*: Strongly restricted sentence patterns, for instance in the style of EARS [Mavin et al., 2009].

Such restrictions imply verification obligations. For instance:

- Do the requirements obey the project's guidelines?
- Do the entities mentioned in the requirements statement belong to the element to which the requirement is allocated?
- Do the requirements fit into the templates?
- Is the chosen template a good fit for the requirements category?

Clearly, the more formal the representation, the higher the potential for automated consistency and correctness checks. While within the SpesML project there are no plans to deeply integrate such checks, it is relatively easy to apply modern NLP toolkits to the task of consistency checks.

Requirements attributes. In addition to the textual requirements statement itself, requirements in SpesML have metadata in the form of attributes, such as

- an *identifier*, as a unique reference to the requirements.
- the requirements *category* (see above).
- the requirements *source* (e.g., a stakeholder or a normative document).
- a *rationale*, as an explanation of the need of the requirement. This is particularly of interest if a requirement does not directly or indirectly originate from a stakeholder, but rather from a process activity (e.g., a safety analysis) or from an architectural decision.
- a *status* field, indicating the degree of maturity of the requirement. Typical values are *proposed*, *accepted*, *reviewed*, *implemented,* and *tested*.
- a *safety* level, such as an ASIL (Automotive Safety Integrity Level) attribute for automotive systems. Other domains have similar attributes, such as SIL (Safety Integrity Level, IEC 61508) or DAL (Design Assurance Level, ARP 4554).

Note that since we treat safety not as a requirements category, but rather as an attribute, the classification in safety-related and non-safety-related requirements is orthogonal to the taxonomy introduced in this section.

This allows easy extension of the requirements landscape to also indicate security, environmental or legal requirements, and offers more flexibility when it is not a priori clear whether there will not be any overlap between for instance safety and security, or security and legal requirements.

6.2.2 Tracing

Requirements are not isolated model elements; they are typically related to other requirements and other model elements through tracing relationships. The following table lists the tracing relationships used in SpesML. The right-most column of the table shows the model elements connected by the requirement ("source/destination"):

- *Req*: Requirements
- *Arch*: Main architectural elements of the views, i.e., functions in the functional view or components in the logical or technical views.
- *Any*: Either requirements or main architectural elements.

Table 6-1: Requirements tracing relations in SpesML.

Relationship	***Description***	***Tracing***
Containment	Decomposition into sub-requirements. This is not really a tracing relationship, but instead part of the requirements meta model. It is frequently used for the co-evolution of architecture and requirements (see below).	Req/Req
Derive	Derivation of one or more requirements from a source requirement. This is related to the concept of refinement in Focus.	Req/Req
Satisfy	Relate requirement with architectural model elements that are intended to satisfy the requirement.	Arch/Req
Trace	General-purpose relationship between a requirement and any other model element. The semantics of trace include no real constraints and therefore are quite weak.	Any/Req
Verify	Relate requirements with test or simulation scenarios or other model elements.	Any/Req
Require	Needed for A/C-like arguments. An architectural model element can *require* requirement to make explicit assumptions needed to *satisfy*other requirements.	Arch/Req
Matches	In order to link assumptions of one model element with guarantees of another. Semantically, the *Matches* relation is identical to *Derive*; the difference is purely methodological.	Req/Req

The most frequently used relationships are *Satisfy* and *Derive*. The *Satisfy* relationship is used to state that an architectural element should satisfy a requirement. Thus, a *Satisfy* relationship gives rise to the verification obligation of checking whether this in indeed the case. There are multiple ways to discharge such an obligation; in industry, functional requirements are typically verified by tests and constraints are verified by reviews. In some development process descriptions, the *Satisfy* relation is referred to as an *allocation* of requirements to architectural elements.

The *Derive* relationship is used to add detail or to include certain design decisions. In the absence of formal requirements notations, *derive* does not have a formal semantics, but

intuitively, if a requirement *R-m* is derived from a requirement *R-n*, we expect that when *R-m* is valid for a given system, then so is *R-n* (note that this is a more liberal demand than the universal "*R-m* implies *R-n*"). Since in general it is not possible to automatically verify this expectation, the *Derive* relationship gives rise to a verification obligation that needs to be discharged during development (typically through a review).
It is a common pattern that an architectural element satisfies a requirement, which itself is derived from a more high-level requirement, as shown in Figure 6-10.
Of course, there may be a longer chain of derived requirements, up to an initial stakeholder requirement.

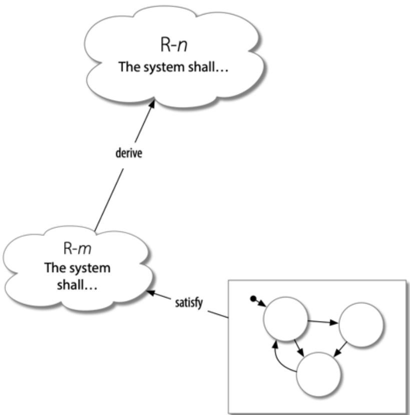


Figure 6-10: Satisfy and Derives relationships.

6.2.3 Requirements Fulfillment

Above we have seen that SpesML requirements have an attribute for requirements status information. It is good practice to be able to represent the two-level verification obligation typically needed for requirements:

- *Review of requirements*: Do we accept the requirement and agree with it? This obligation is quite straightforward and easily supported through requirements quality criteria such as those of INCOSE [2017].
- *Verification of the system against requirements*: Does the system - or a part of the system - fulfill the requirement? Note that this is related to the use of *Satisfy* links. The question is whether indeed the architectural element at the source of the link indeed fulfills the requirement at the end of the *Satisfy* link.

An immediate question when considering *Satisfy* links and validity of requirements is whether the source of the *Satisfy* link stems from the (real) system to be developed or from the model of the system to be developed.

For SysML, the interpretation is that it is the real system - or a part of the real system - that must satisfy the requirement. Typically, requirements satisfaction is demonstrated through tests. Consequently, in SysML, *Verifies* links are used to link test cases with requirements.
For SpesML, the situation is slightly different. Testing of systems is highly dependent on the nature of the (real) system as well as on test environments, test techniques and test tools. Testing of systems is out of scope of the SpesML project with its focus on architectural models.
On the other hand, there are some verification goals that can be fulfilled considering architectural models only. One example is the verification of design constraints (see the taxonomy in Section 6.2.1), which pose demands on the architectural model. Obviously, design constraints can be verified on the model itself. In SpesML, a number of common design constraints have been added as well-formedness rules (WFRs); they can be verified in the SpesML tool.
Another example is the use of simulation of an architectural model with executable component specifications as a specific form of prototyping in order to get early feedback on the feasibility of the chosen architecture to satisfy requirements.
Using simulation is straightforward to get confidence into the feasibility of an architecture to satisfy capability requirements (again, see the taxonomy in Section 6.2.1) – capabilities are typically existential properties ("the system shall be able to...") which only need examples for demonstration.
Functional specifications, however, are typically universal properties ("whenever the following simulation holds, the system shall…"). Here we need some coverage argument as an explanation why the chosen simulation scenarios are sufficient to demonstrate fulfillment of the requirement.
The question of fulfillment of requirements of the other categories should not be neglected in a systems development project; however, this is not in the scope of the SpesML project.
Representation of Requirements Fulfillment. Common to the capability and functional requirements categories is that their fulfillment is demonstrated primarily through test or simulation. For SpesML, the main question here is whether test simulation/test setups, scenarios and results should be included or referenced in the architectural models. For functional requirements - typically being universal properties -, in addition a coverage argument may need to be documented.
The *Verify r*elationship is the main relationship to demonstrate fulfillment of requirements. It can be used to to link requirements to test or simulation information that is used to verify this requirement. Fulfillment also needs to consider which system or model element is responsible for ensuring a requirement; this aspect is denoted by the *Satisfy* relationship.
In summary, the use of a test setup to demonstrate that an architectural element (a logical or technical component, or a system or white-box function) satisfies a requirement *R* is as shown in Figure 6-11.

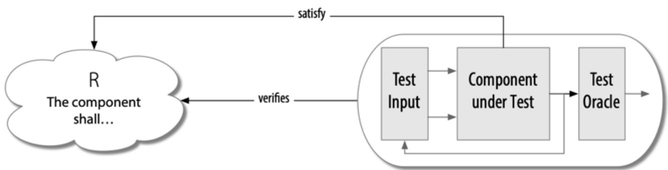


Figure 6-11: Use of simulation setups to demonstrate requirements fulfillment

6.2.4 Methodology

As in most professional activities, for system development, certain activities are recurring and can be abstracted into "patterns". This section gives a small selection of examples.

Co-Evolution of Requirements and Architecture. Frequently, requirements and architecture co-evolve in the sense that not only are architectural decisions based on requirements, but also that vice versa requirements decomposition and derivation may depend on architecture. This "zigzagging" between requirements and architecture is also recognized by development standards (see, for instance, ISO 26262, Part 10, Clause 7).

It can be represented in SpesML through the common decomposition of requirements and architecture, as shown in Figure 6-12.

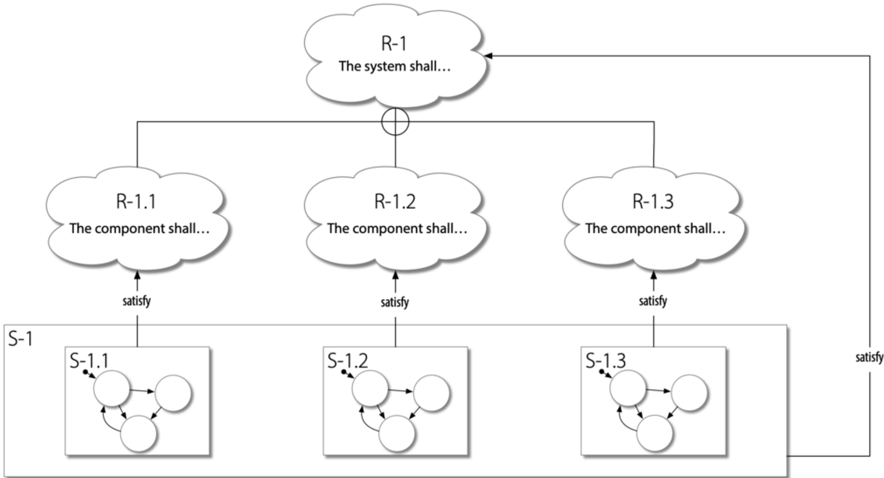


Figure 6-12: Decomposition of requirements and architecture.

Note that requirements are decomposed through the *Containment* relationship, which is not a proper trace relation. Instead, containment is encoded into the requirements model itself. It is used to divide a requirement into sub-requirements which can then be traced, allocated and verified independently.
When using this co-evolution pattern, one would expect the following conditions to hold:

- Taken together, the sub-requirements must imply the parent requirement.
- If the parent requirement is satisfied by a function (in the sense of an architectural element) *F* or by a logical or a technical component *E*, the sub-requirements must be satisfied by a sub-function of *F* (in the sense of the functional hierarchy) or by sub-components of *E*, respectively.

Development against assumed requirements. Subsystem development frequently occurs in parallel with, or earlier than, development of the main system. Typical examples for such concurrent development are:

- Early development of subsystems or components within an organization before requirements are finalized and agreed upon; in fact, in some cases, it is precisely the experience of such early development sub-projects that is needed to finalize requirements and architecture.
- Development of "Baukasten" subsystem or components within an organization with the intent of using these elements in the construction of product systems for customers.
- Development of off-the-shelf subsystems or components by suppliers.

In these cases, development happens against *assumed* requirements, as there are no stable system requirements yet available. This means that it is not certain that the independently developed subsystem or component does indeed fulfill the properties required by the system. Moreover, the same argument goes in the other direction: A subsystem or component can have requirements on the integration context which may or may not be satisfied by the system to be developed.
In SpesML, the *Require* and *Matches* relationships are intended to represent such development situations:

- The *Require* relationship is used to state expectation on the integration context of a subsystem (from the point of view of the subsystem), as well as expectation on the properties to be fulfilled by the subsystem (from the point of view of the system).
- The *Matches* relationship is then used to describe which of the properties satisfied by the system imply that the integration demands of the subsystem are fulfilled, as well as which of the properties satisfied by the subsystem imply that the expectations of the system on the subsystem are fulfilled.

As an example, given a supplier intent on developing a subsystem that generates status information. When starting development, it is not yet determined at which frequency, let only with which precise formats and interface technologies and protocols, the status information has to be produced. Still, the supplier continuous working and develops such a system. During development, it turns out that there are some demands on whatever system the subsystem is to be integrated in - most prominently, electrical power with certain characteristics must be available. Thus, subsystem development proceeds against *assumed* requirements (which are

then guaranteed or *satisfied* by the subsystem) and also for an *assumed* integration context (which is *required* by the subsystem).
Conversely, the system developer discovers a need to have a subsystem to provide status messages with a certain minimum update rate. Clearly, the subsystem will have power demands; hence, a certain power budget is guaranteed by the system.
When integrating the subsystem of our example, the integrator has to ensure that the guaranteed performance of the subsystem -i.e., the frequency of status delivery- matches the demands of the system under development. Conversely, the integrator also has to ensure that the expectations of the subsystem -i.e., its power demands- can be satisfied by the system under development; see Figure 6-13 for a simple example.

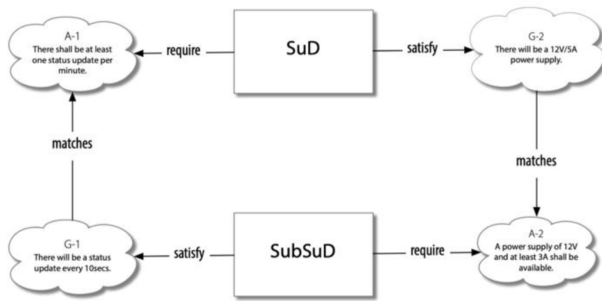


Figure 6-13: Require and matches relationships.

While in general it is not possible to automatically verify such pairings of assumed (*Require*) and guaranteed (*Satisfy*) properties, it is typically feasible to manually review them. The *Require*/*Matches* relationships help to identify and track these review obligations.
Note that in general the *Require*/*Matches* combinations may be circular: Without supplied power, it should come as no surprise if the subsystem fails to deliver status information. On the other hand, perhaps the system will only route power to the subsystem if the status information implies the subsystem is indeed alive?
Such circles must be broken by rephrasing one of the requirements in a way such that at least one component fulfills its guarantee in advance; only if then its assumed requirements are not fulfilled by the other component may the first component renege on its guarantees. Clearly, this approach suitable only for behavioral properties. However, in practice this will rarely be a problem. Circularities are unlikely to occur in design constraints, and assumptions related to quality requirements typically involve resource guarantees allocated through a top-down budgeting process.
Note that the *Matches* relationship is closely related to *Derive*, in that they state that if the source requirement is valid, the destination requirement of the trace should be valid, too. As with *Derive*, *Matches* gives rise to a verification obligation, namely, to verify that *Required* requirements of the subsystem are indeed fulfilled by *Satisfied* requirements of the main system, and vice versa.

Note that this discussion only considered the two requirements tracing relationships. In SpesML, there is also a methodology for subsystem development, which is closely related to independent development against assumed requirements; see Section 6.6 for details.

6.3 Functional Viewpoint

The functional viewpoint (FVP) describes the desired functionality of a system on a system level (i.e., primarily from the perspective of external actors such as users or external systems).
In the center of the functional viewpoint is the concept of *system functions*. A system function describes a coherent set of interactions between a system and its external actors or external systems. The granularity of a system function is comparable to that of a *Use Case* [33]. A system with several system functions is called a *multifunctional system*. In general, system functions describe system behavior that is largely independent of each other, however, system functions may also influence each other in specific cases. In the functional viewpoint, these influences are modeled by so-called *mode channels* between system functions.
The functional viewpoint then consists of three model types:

- *Functional Black-box Model*: Contains all system functions and structures them in a hierarchy.
- *Functional White-box Model*: Describes the decomposition of one system function into a set of related white-box functions.
- *Mode Model*: Describes operational states of the system. These operational states can be referenced in mode channels that describe dependencies between system functions.

In the following, we describe the three model types and how we represent them using the *Universal Interface Model* (see Section 4.3).

6.3.1 Model Elements

Functional Context: The System and its Context in the Functional Viewpoint. Just like all other viewpoints, the functional viewpoint refers to a specific system scope, i.e., there is a clear notion of what is part of the *system currently under development* and what is considered the context of that system. The context described in the functional viewpoint focuses on functions (i.e., chunks of behavior) in the context rather than entities. The system boundary and the system context are specified in the *Functional Context*. This diagram contains one specific *Function* that represents the entire System-under-Development (SUD) and additional functions around the SUD that describe functions provided by the context.
Figure 6-14 shows a Functional Context Model with the main system function *manageSystem* and the surrounding context functions that influence or are influenced by the system.

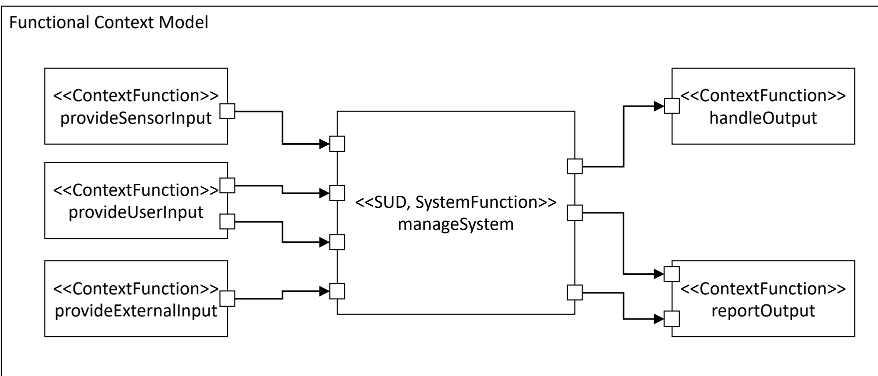


Figure 6-14: Functional Context

System Functions. A system function describes a coherent set of interactions between a system and its external actors or external systems. For this purpose, it uses the modeling elements of the Universal Interface Model (see Section 4.3). That means a system function is described by a specific modeling element called *Function*. Each system function has a syntactic interface defined by a *Functional Interface*, i.e., a set of input and output ports with associated types. Figure 6-15 shows a system function *doSomething* with its associated interface consisting of three input and two output ports.

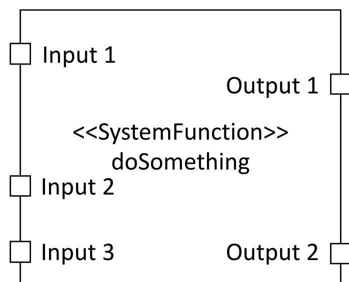


Figure 6-15; System Function

System Function Hierarchy. A system can have several system functions. If the number of system functions is small (e.g., less than 10 system functions), we can simply list the system functions and model their interfaces. However, if a system offers a large number and variety of system functions (like an entire car), it may be beneficial to structure the system functions hierarchically with respect to some *functional domains*. To model this hierarchy, a functional domain can be described as a composition of system functions. Technically, the functional domain is, again, a system function with an interface, which is defined by the composition of

the contained system functions. Figure 6-16 shows a system function hierarchy with several system functions organized in functional domains.

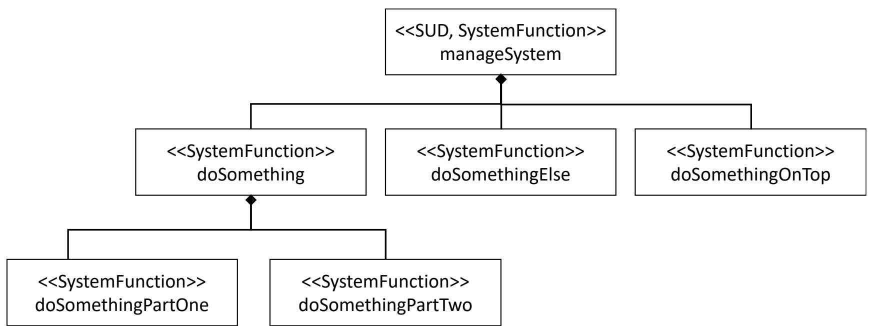


Figure 6-16: System Function Hierarchy

Functional Black-Box Model. To model this composition, a system function can have *parts*, which again represent system functions. The *ports* of the *parts* can be connected. Such connections between system functions describe functional dependencies between system functions (e.g., a specific state of one system function affects the behavior of another system function). These dependencies should be considered with care since you should specify system functions as independently from each other as possible.
Remember that system functions describe interactions between a system and its external actors or external systems. That means, a decomposition into system functions must not result in *internal* functions, which have no interface to the context of the system. The system function that represents the *root* of the system function hierarchy specifies the interface of the entire system from a functional point of view (i.e., by a vocabulary and on a granularity level that is close to the functional system requirements).
Figure 6-17 shows a Functional Black-box Model. In this example, the system contains three system functions (*doSomething*, *doSomethingElse*, and *doSomethingOnTop*). There are mode channels between the functions (*Mode 1* and *Mode 2*) that are used to describe that the *doSomething* functions is influenced by the vakue of the two modes, which are determined by the *doSomethingElse* and *doSomethingOnTop* functions.

White Box Functions. A white-box function describes a unit of behavior that is necessary to realize a system function. For this purpose, it uses the modeling elements of the Universal Interface Model (see Section 4.6). That means a white-box function is described by a specific modeling element with an associated syntactic interface defined by *ports*. The behavior of a white-box function can be specified by any type of behavior specification that relates input streams received on the input ports to output streams produced on the output ports (e.g., state machines, see Section 5.5).

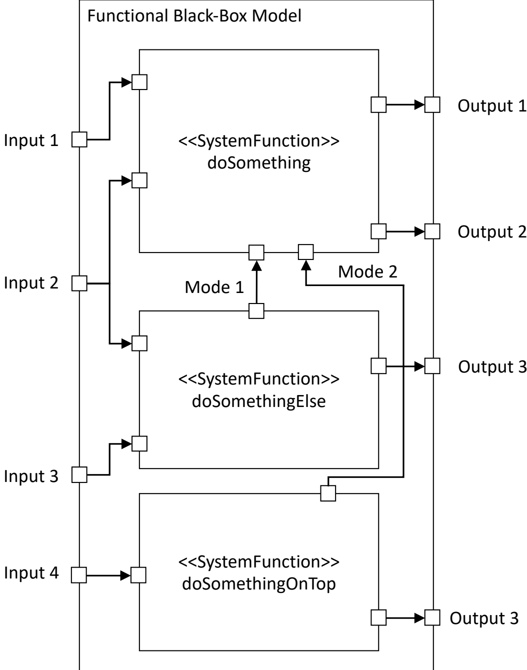


Figure 6-17: Functional Black-box Model

Functional White Box Model of a System Function. A system function is realized by a set of communicating white-box functions that may again be decomposed into smaller white-box functions. To model this hierarchy, a system function is related (via a *refinement relation*) to exactly one *functional white-box model* that describes a composition of white-box functions. To model this composition, a white-box function can have *parts*, which again represent white-box functions. The *ports* of the *parts* can be connected. The resulting *chain* of white-box functions for a system function is sometimes also called a causal chain. In contrast to system functions, most white-box functions will only have *internal* ports, i.e., output ports are connected to input ports of other white-box functions. Only a few white-box functions will have input or output ports that are *not* connected to other white-box functions. These *external* ports must reflect the syntactic interface of the corresponding system function.
Figure 6-18 shows the functional white-box model of the system function *doSomethingElse*. It defines four white-box functions that specify the necessary steps to realize the system function.

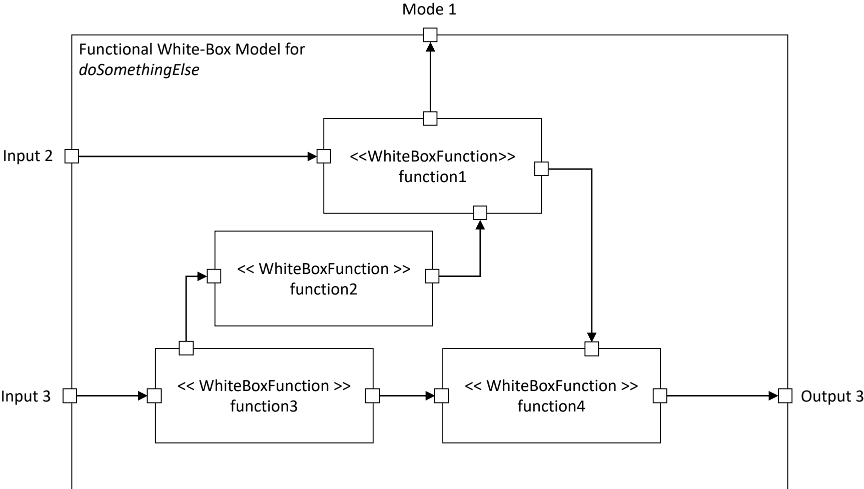


Figure 6-18: Functional White-Box Model

Mode Model. As described above, system functions can influence the behavior of other system functions. In the functional black-box model, we describe these influences by ports and channels between two system functions. The messages sent over these channels often describe certain operational states (a.k.a. *system modes*) that influence the behavior of other system functions. Therefore, channels between system functions in the functional black-box model are called *mode channels*.

In addition to the functional black-box model, we can also describe a *mode model* that focuses exclusively on the modes and possible transitions between mode values. A mode model is modeled by a state machine where the states represent values that can be sent over a mode channel in the functional black-box model.

In our example, we had two mode channels (*Mode 1* and *Mode 2*) between the system functions. The port associated with the *Mode 1* channel has a type that is defined as follows (see Section 4.2):

Mode 1 = {Mode1_Active, Mode1_Inactive}

The port associated with the *Mode 2* channel has a type that is defined as follows (see Section 4.2):

Mode 2 = {Mode2_Low, Mode2_High}

From this information, we can create a mode model that contains a state for each of the values. In addition to the states, we can also specify transitions between the states to define allowed mode sequences (see Figure 6-19). In the example, we specify that only specific mode sequences are allowed to be transmitted over mode channels with the specific mode types. The system

functions that have an outgoing mode channel with such types must adhere to the allowed mode transitions (e.g., *doSomethingOnTop* must not define behavior that sends a sequence like <..., *Mode2_Low*, *Mode2_High*,...> over the mode channel *Mode 2* after a message *Mode1_Inactive* has been sent over the mode channel *Mode 1*). Thus, the mode model defines a kind of mode protocol that all functions must adhere to.

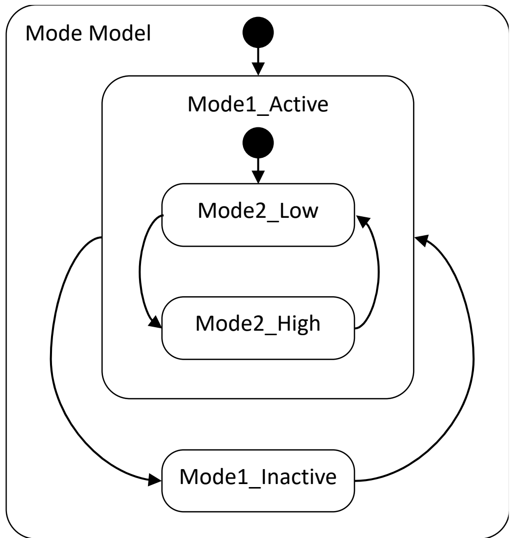


Figure 6-19: Mode model based on the data type of the mode channel.

6.3.2 Tracing

Detailed information about trace links between the functional viewpoint and other viewpoints can be found in Chapter 6.7.

In general, system functions structure and formalize the functional requirements of a system. Thus, system functions *satisfy* functional requirements. An engineer may decompose system functions into a set of white-box functions to assign smaller units of functionality to logical components that realize the functionality specified by a white-box function. Figure 6-20 illustrates these relations.

6.3.3 Well-Formedness Rules

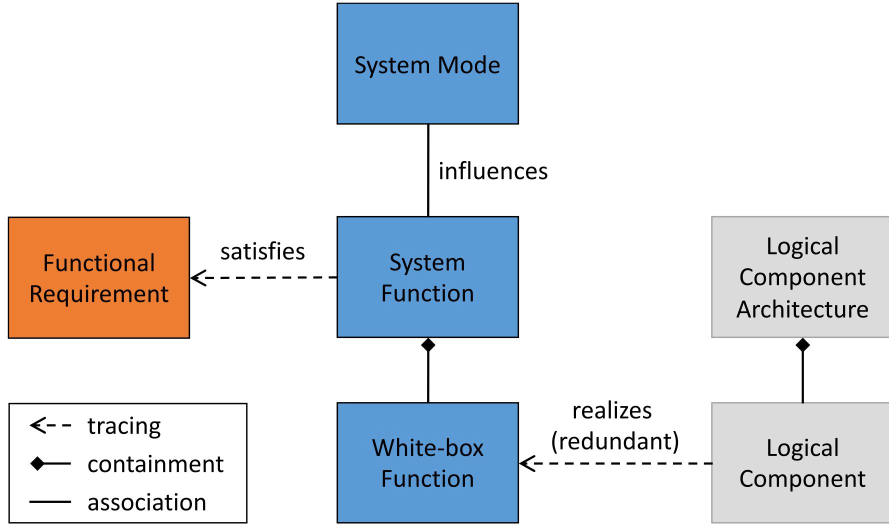


Figure 6-20: Elements of the Functional VP and their relations to elements of other viewpoints

The functional viewpoint incorporates all the general well-formedness rules (WFR) that were introduced for the whole SpesML concept due to, e.g., the UIM. In addition, the following rules were identified specifically for the functional viewpoint.

Functional Context

WFR-F1: The functional context must include exactly one system function that represents the SUD.

- *In the functional view of a system, there is one element that represents the SUD. This element is a system function that represents the composition of all system functions of the system. All context functions must interact with this top-level system function.*
- *Reason: The functional context shall define what is considered part of the system and what is considered part of the context. It does not make sense to have more than one element summarizing all functions of a system.*

System Functions

WFR-F2: System functions must have at least one output port connected to an output port of the overall SUD or at least one input port connected to an input port of the overall SUD.

- *Reason: A system function describes functionality that is observable by an external actor. A system function without any input or output port connected to the outer system interface would describe internal system behavior not observable by*

an external actor (e.g., function 2 in Figure 6-18 *describes only internal behavior and is thus not a system function).*

WFR-F3: The name of a system function should be a verb+noun combination.

- *Reason:* A system function models behavior, so the name should begin with a verb (e.g., *controlWindows*, *doSomething*)

System Function Hierarchies

WFR-F4: There exists exactly one root system function (the SUD).

- *In the functional view of a system, there is one element that represents the SUD. This element is a system function that represents the composition of all system functions of the system and, thus, is the root of the system function hierarchy.*
- *Reason: It does not make sense to have more than one element summarizing all functions of a system.*

White-Box Functions

WFR-F5: Each functional white-box model must be related to exactly one system function.

- *Reason: A white-box model is the refinement of a system function. If a white-box model does not have any relation to a system function, it is not clear for which purpose the white-box functions exist.*

WFR-F6: Each system function must be related to at most one functional white-box model.

- *Reason: A white-box model is the refinement of a system function. If there are several white-box models related to a system function, it is not clear which one describes the system function.*

WFR-F7: The interface of a system function (i.e., inputs and outputs) must be reflected in its related functional white-box model.

- Each input and output port of the system function must be a free input or output port in the functional white-box model. Free means that ports are not connected to other ports. Free ports emerge from the composition of all white-box functions in a white-box model. For example, the system function *doSomethingElse* in Figure 6-17 has two inputs (*Input 1* and *Input 2*) and one output (*Output 1*). The corresponding white-box model (Figure 6-18) reflects these ports at its outer interface.
- *Reason: A white-box model is the refinement of a system function. Therefore, the interface of the system function must correspond to the outer interface of the white-box model.*

WFR-F8: Each white-box function must be related to at most one logical component that implements this white-box function.

- *A white-box function that is related to more than one logical component must be broken down into smaller white-box functions that can be related to single logical components.*

- *Reason: If a white-box function is related to more than one logical component, it is not clear which part of the white-box function shall be realized by which logical component.*

Mode Model

WFR-F9: There is at most one mode model per system.

- *Reason: The purpose of the mode model is to capture all relevant modes of the system. It does not make sense to cover them in several mode models since this may hide dependencies between modes of several mode models.*

WFR-F10: Each value that can be sent over a mode channel must have a corresponding state in the mode model.

- A mode channel is a channel between system functions in the functional black-box model. The type of a mode channel defines a mode of the system. Each mode must be represented as a state in the mode model.
- *Reason: The mode model is an orthogonal view of the system behavior with a focus on the operational modes and their transitions.*

WFR-F11: Transitions in the mode model do not have any guards or actions.
Reason: A mode model defines potential sequences of modes, for which it is only relevant which mode may follow on which other mode.

6.4 Logical Viewpoint

The *logical view* describes how the system under development (SUD) is structured to achieve the behavior which is specified in the *functional view* independent of the technical realization. To do so, the logical view describes the logical system context, the decomposition of the SUD in logical components and subcomponents and their respective interface behavior. This sub-chapter describes the basic concepts and models for the logical view of a SUD.

6.4.1 Model Elements

The central modeling concept of the logical viewpoint are the so-called *logical components*. They strictly follow the universal interface model (REF auf UIM Kapitel) including syntactic and semantic interfaces and the concepts of causality, composition and decomposition. In accordance with this, the logical components have *logical interfaces* that communicate with logical interfaces of other components via *channels*. Every logical component is modeled with their behavior as a function that maps the inputs of the logical interfaces to the respective outputs. This behavior function is modeled in SpesML using *state machines* or decomposing the component into a distributed network of subcomponents, which again can have an internal structure.
The following sections will explain in more detail how these model elements can be used to model the logical view on the system under development.

6.4.2 Logical Context

The *logical context* models the environment in which the SUD shall operate on a logical level. Context elements can be actors that interact with the SUD or external systems. Besides its importance for simulating the system behavior, one other objective for modeling the logical context is also to specify the logical interface of the SUD to the operational context on a logical (i.e. technology independent) level. To do so, the logical context model refines the context model of the functional context from the functional view. For this refinement, we recommend to map every context function completely to one logical context component. This logical context component can then refine the context function – usually such a refinement concerns the interfaces and interface types at the system boundary to the system context.
As for the functional and technical view, the logical context defines the scope of what is inside the system boundary and what is outside. In the SpesML, the logical components for the SUD and the element of the context are connected in the logical context diagram as can be seen in Figure 6-21.

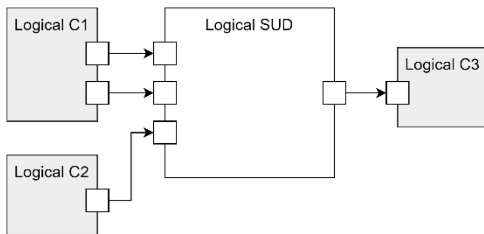


Figure 6-21: An example for a logical system context model. The external components C1, C2 *and* C3 *are displayed in grey.*

Interface Behavior of Logical Components. As already mentioned, also the models of the logical viewpoint are based on the concepts of the universal interface model. This means, that interfaces of logical components consist of a syntactical and a semantical interface. The syntactical interfaces of logical components are modeled using *logical interfaces* and all logical interfaces need to be assigned a *logical interface type*. This interface type again is defined as a set of channels. This conceptual metamodel for logical interfaces can be seen in Figure 6-22. The semantical interface of logical components can be modelled by means of state machines or by decomposition into a network of sub-components (see Section 6.4.3) Such state machines operate based on the component's input interface values and produce the component's output interface values. To model this, the guards and actions cannot only contain comparison and assignment operations, but also use executable functions. More information on SpesML state machines can be found in the respective chapter 5.5.

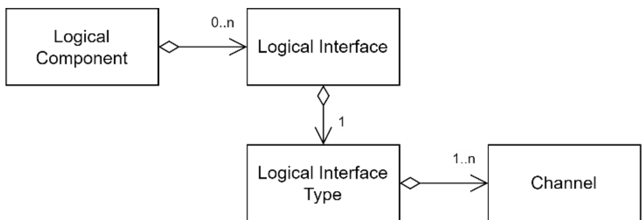


Figure 6-22: Conceptual Metamodel for syntactical interfaces of logical components.

6.4.3 Decomposition of Logical Components

To structure the logical system architecture and to reduce complexity, all logical components can be decomposed into subcomponents via a sub-component refinement. This means that the behavior of a decomposed component results from the composed behavior of the subcomponents. In SpesML, decomposition is modeled by means of dedicated diagrams in which subcomponents are instantiated and connected as can be seen in Figure 6-23.

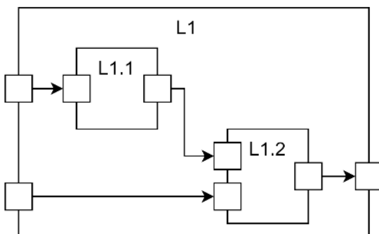


Figure 6-23: Example for the decomposition of a logical component into subcomponents: The logical component L1 *is decomposed into* L1.1 *and* L1.2.

Decomposition of logical components also involves the syntactic and semantic interface of components and subcomponents: Here, all input interfaces of the component need to be connected to an input interface of a subcomponent and all output interfaces of a component need to be connected to an output interface of a subcomponent. More detailed information can be found in Chapter 4.3 on the universal interface model.
This decomposition of logical components to sub-components can be recursively applied to all logical components and sub-components, until eventually all sub-(sub-)components got assigned a behavioral description either in form of another decomposition or by a state machines (atomic components).

6.4.4 Tracing

The logical components are traced to the white box functions they realize from the functional view. For this tracing, it is recommended to trace *n* white box functions of the FV to exactly *one*

logical component, such that no white box function is implemented by several logical components.
There are also traces to the technical viewpoint which models the technical realization of the logical components. Also here, it is recommended to trace every *individual* logical component to exactly *one* technical component. More detailed information on these concepts regarding tracing can be found in Section 6.7.

6.4.5 Modeling the Transition to the Technical Viewpoint

The logical view can also explicitly model the *transition* from the logical to the technical viewpoint, as is introduced in the section on tracing in more detail. In particular, the identification of subsystems in the technical view can be modelled here, by decomposing logical components into different subcomponents according to the engineering disciplines they can be realized in. The engineering disciplines we suggest to differentiate in SpesML are *Software*, *Mechanical*, *Electronic* and *Mechatronic*, but in general also other disciplines which are distinguished in the technical view would be feasible.
This distinct mapping to single engineering disciplines for logical components or subcomponents, is recommended, since these components will play different roles for the respective technical components in the technical view: In case a component is implemented in hardware, the logical components behavior might be used for simulation purposes in later stages, while for the components which are realized in software, the logical behavior might be an input for code generation. For all of these purposes, the logical behavior needs to be clearly defined for each technical component in all technical disciplines. In all such cases, the SpesML methodology strongly suggests to decompose or regroup the logical architecture such that a clear assignment to engineering disciplines and a clear tracing to the respective technical components is made possible.

6.4.6 Well-formedness Rules

The well-formedness rules for the logical viewpoint all result from the well-formedness rules of the universal interface model. In this Section we summarize the application of these rules to the logical viewpoint.

Logical Components and Interfaces

WFR-L1: Every logical component and logical interface must have a name.

- Reason: To be used in textual specification languages, elements need to be able to be referenced via a name.

WFR-L2: Every logical component must have at least one logical interface.

- Reason: A logical component without any interface would not have any observable behavior and hence would not be relevant for the system.

Logical Interface Behavior

WFR-L3: All logical components and sub-components need to have an interface behavior model in form of decomposition or a state machine.

- Reason: To enable the verification of the system via simulation or analytic techniques, the behavior of all components needs to be defined.

WFR-L4: State machines must only refer to interfaces defined by the logical component in which they are defined.

- Reason: Only the interfaces which are defined by the component itself are accessible and have a defined semantics in the respective scope.

WFR-L5: State machines must not assign multiple values or sequences to logical output interfaces at the same time.

- Reason: If multiple values or value sequences would be assigned at the same point in time, the order of these values could not be determined unambiguous.

Composition/Decomposition

WFR-L6: All logical input interfaces of all logical components must be connected to other logical interfaces.

- Reason: It would be undefined, which values an unconnected interface would obtain and hence, the behavior of the logical component could not be analyzed or simulated.

WFR-L7: Connected logical interfaces need to have the same type.

- Reason: It is undefined, how input values of a different type than declared by the receiving component would need to be handled.

WFR-L8: Every logical input interface needs to be connected to exactly one logical output port.

- Reason: In case an input interface would be connected to multiple outputs, the order of incoming messages would be undefined in case several messages are received at the same point in time.

6.5 Technical Viewpoint

The *technical viewpoint* and its instances, the *technical views*, are mostly concerned with the question of how to get from the platform-independent models of the logical viewpoint (see Section 6.4) to platform-specific models [9]. In other words, it is modeled how the specifications of a logical view can be implemented using concrete technical realizations. In the following, all models of the technical viewpoint are described.

Core concept. The main idea behind the technical viewpoint is to map the logical components of the logical viewpoint to technical components, which are platform specific and usually modeled for a specific technical discipline. In general, we see a generic technical model on the first granularity layer representing the overall system, which can then be refined to more specialized models on the next granularity layer(s). A good example for such a refinement is the so-called Software Execution Subsystem (SES), which is described in detail in Section 6.5.2.

The presented approach for a technical viewpoint targets various scenarios that may emerge in systems engineering, e.g.:

- Integration of technical components from various engineering disciplines such as integration of a mechatronic component, which includes mechanics, electronics, and software. Such a

scenario can be supported by specific technical components and their dedicated interfaces to other technical components.
- Engineering a new system architecture in terms of hardware topology, e.g., with a more centralized software part. This requires an overview of computation and communication resources that are intended for the new system.

We also envision combinations and variations of these scenarios, which are enabled by the proposed approach.

Behavior modeling. SpesML does not provide behavior models for technical views since we are not able to describe the behavior of all possible technical components in detail. It is possible to use dedicated external techniques if such a detailed description is needed. However, we provide the input for such behavior modeling through our technical viewpoint concepts that are described in the following. These concepts can also enable further development steps like deployment and scheduling analysis. In contrast to other SpesML viewpoints, the technical viewpoint only considers the syntactical interface of the Universal Interface Model (UIM), which is explained in Chapter 4.3.

Section outline. In Section 6.5.1 we will first cover the general model elements of the technical viewpoint on the first granularity level before describing in Section 6.5.2 the more specific models and elements of the SES in detail. Figure 6-24 illustrates the elements and their relations within the technical viewpoint. Finally, we point out the tracing aspects of the technical viewpoint in Section 6.5.3 as well as specific well-formedness rules in Section 6.5.4.

6.5.1 Model Elements

The basic model of the technical viewpoint is the *technical component architecture*. It represents the existing technical components of a system including their domain affiliation and how they are connected. This architecture is most important on the system layer of granularity, but it can also be used on all lower layers to describe the technical architecture of each subsystem. Figure 6-25 shows an abstract example of how a technical architecture can look like across granularity layers, beginning with the technical context.

Technical context. Like in the functional and logical view, a context can be modeled within the technical view as well. With such a *technical context* it is possible to place the current system under development (SUD) in relation to connected external systems and *technical actors*. The technical context contains the current SUD and exactly all technical systems and actors that are directly connected with the SUD via at least one technical interface. This way it is possible to display where the system inputs are coming from and where the system outputs are going to without explicitly modeling all the external systems. More about contexts in SpesML can be found in Section 6.1

Components. The technical architecture of a system is described using components. These components represent the system either in a generic way or more specific using models of the respective technical engineering disciplines. The generic component is called *technical component* and can be used for everything in any discipline. If the content of a component can be assigned completely to one specific discipline, a more specific component type can be chosen to represent this. Referring to this, available disciplines are mechanical engineering with the *mechanical component*, electrical engineering with the *electrical/electronic component* (in the following denoted as E/E component), and software engineering with the *software execution*

component. The software execution component is a container for all the model elements of the SES, which is explained in Section 6.5.2. The generic technical component can also be called *mechatronic component* because it represents the combination of all three of these specific disciplines - mechanical, electrical, and software engineering.

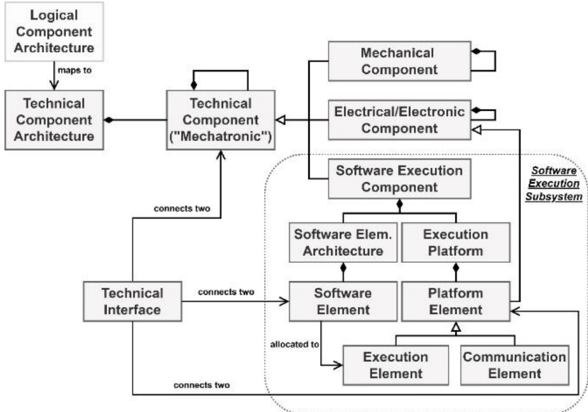


Figure 6-24:Overview over the SpesML technical viewpoint concept

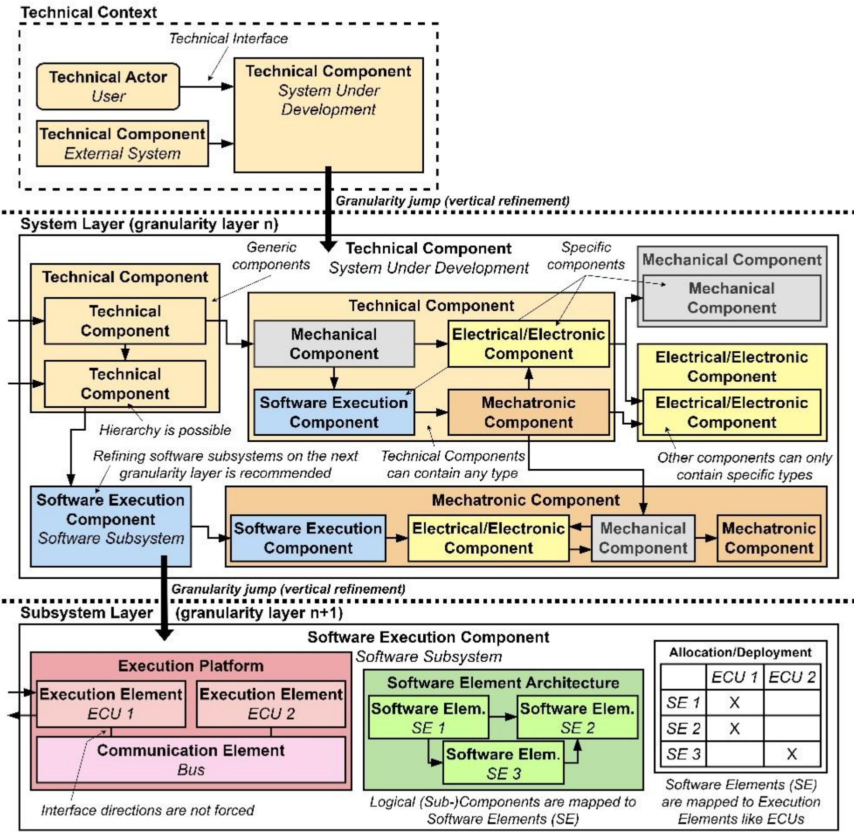


Figure 6-25: An example of how the technical elements can be composed including different granularity layers. Colors indicate components from different technical disciplines.

Interfaces. All the previous components can interact with other components via syntactic interfaces that are modeled from an implementation point of view, for instance representing the type of cable or the protocol used for this connection. For the technical architecture on system level, there is just a single generic interface type called *technical interface*. It can connect every component type with every other one.

Decomposition. Components may be decomposed into further subcomponents within the same granularity layer. The purpose of such hierarchical decompositions is usually to gain a better overview, especially for large, complex systems. It is possible to decompose a technical component into any combination of any other subcomponents. The other domain-specific components can only be decomposed in subcomponents of their own type, i.e., a mechanical component can only contain more mechanical components, and an E/E component only more E/E components. An exception is the software execution component. It represents the SES and is decomposed on the next granularity level by a set of dedicated models (see Section 6.5.2) including software element architectures and execution platforms. Note that it is not possible to

decompose a software execution component into other component types like mechatronic, mechanical, and E/E components.

Design recommendations. As the purpose and abstraction of specific engineering disciplines are highly heterogeneous, their models differ accordingly. Whether a component is further refined as an independent subsystem on the next granularity layer depends on the specific development project. It is always recommended to refine the SES and its software execution component on the next granularity layer since very different kinds of models are used to represent the software element architecture, execution platform architecture and their corresponding allocation.

6.5.2 The Software Execution Subsystem

When a software execution component is refined, a dedicated set of models can be used to describe it that do not strictly follow the principles of the UIM. This is why it is strongly suggested to create these models on the next granularity layer as subsystem, therefore conceptually speaking of it as *Software Execution Subsystem (SES)*. The term "execution" was added to its name because it is a subsystem that not only models the software of the system but also the execution aspects of the software including the hardware that is needed for that. Figure 6-26 shows how the SES is divided into dedicated models. In the following, this section provides an overview of these possible models and their relations.

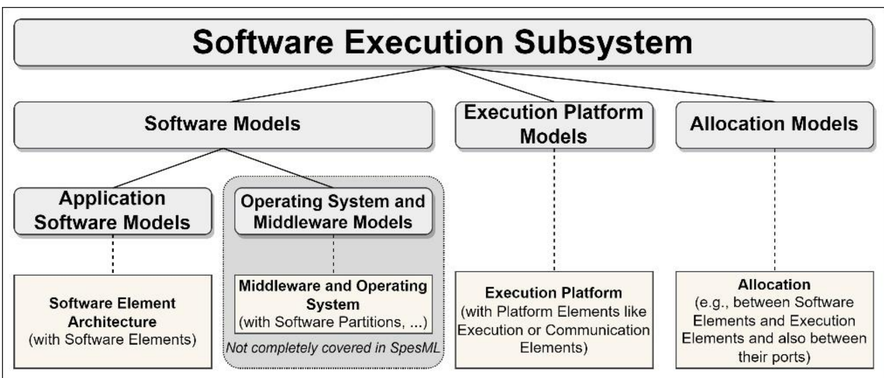


Figure 6-26: Overview over the models of the Software Execution Subsystem (SES).

As part of the SES, the *execution platform models* represent everything that is needed to execute the software of the system. Its main model is the execution platform, which describes the hardware in terms of platform elements such as execution or communication resources. Another part of the SES are the *software models* representing the pure software part of the system. The software models have both dynamic parts, represented by dedicated *application software models* like software element architectures, as well as static parts, represented by *operating system and middleware models*. Finally, relations between software architecture and execution platform are described by potential *allocation models* where software artifacts are allocated to execution resources.

In SpesML, a *software execution component* is a container for all model elements of the corresponding SES concept and can contain *software element architectures* (for the application software), *execution platforms* (for the execution hardware topology) and *allocations* between them. Model elements for the operating systems and middlewares, like partitions, were not realized in the tool implementation of SpesML since they heavily depend on various individual needs and can hardly be specified in a generic way. In the following sections, these models are described in more detail.

Application Software Models. Application software models represent the dynamic part of the software models in an SES. For instance, application software can be described by a set of executable software elements (e.g., AUTOSAR runnables), which processes information from its inputs to its outputs, and a bus message catalogue, which contains all messages representing the technical dataflow between these software elements. In SpesML, a *software element architecture*, which is part of an SES, is used to model the structure of such *software elements*, which can communicate via *messages*:

- *Software element*: Software elements are entities of the technical software architecture. If subcomponents of the logical architecture should be realized in software, they can be mapped to these software elements. The mapping of software-related logical subcomponent to software elements is *n:m*, although it is recommended to map individual logical subcomponents to exactly one software elements. A software element defines which kind of incoming information is processed to create which kind of outgoing information. Messages model the kind of information that is received and/or sent by interfaces of these software elements. The software elements will be deployed onto the execution units of an execution platform (see below).
- *Message*: Software element interfaces can be connected to model an information flow between software elements via messages. An interface input can only be connected to exactly one interface output of the same message type. An interface output can be connected to multiple interface inputs of the same message type. Messages contain information that is sent between software elements to process it there. SpesML uses *technical interfaces* to define such interfaces and specify the messages that can be sent through them. Following the syntactical UIM, each technical interface can have multiple channels. Each channel has a type that defines which type of information can be sent via the channel and the interface. Note that the behavior of interfaces is not explicitly modeled. The same technical interfaces are also used as connection possibility between all other technical elements besides software elements as described previously.

Software elements can have properties that are useful for further development steps like deployment optimization. As demonstration, we introduced in SpesML the software element properties *flash memory usage*, *RAM usage*, and *needed ASIL safety level*. They can easily be expanded and adjusted for other projects as well. If such properties are defined for every software element and if the execution elements have their own corresponding properties, it is possible to use solvers to find satisfying deployment allocation solutions.

Operating System and Middleware Models. Conceptually, SpesML also supports operating system and middleware models as part of the software models in an SES. These models include models representing specific middleware aspects and operating system models and thus are representing the static parts of the software models. For instance, such models comprise of

models representing certain middleware aspects, including, for instance, virtualization and potential OS-related models. Software partitions are one example of artifacts that can be described here. Operating system and middleware models are usually modeled as black-box models that describe the syntax of software models.

Execution Platform Models. The *execution platform* enables the description of an execution hardware topology in SESs. It is denoted it as execution platform, since it captures exactly those parts of the system that are necessary to execute the software elements. Therefore, we want to clearly distinguish this architecture from the more general hardware architecture that could also contain, e.g., mechanical components. The previously described software element architectures of the application software models do not define where their software elements are executed. Therefore, execution platforms are needed to specify possible hardware resources for software execution to know in combination with the deployment allocations where the software can and will be executed. The most prominent hardware components hereby are execution units like Electrical Control Units (ECUs) that can execute software elements. However, also technical communication units such as bus systems are highly relevant for the software execution in distributed technical architectures. Therefore, execution platforms can contain *execution* as well as *communication elements* to represent these two entity types. Although we were focused on these two types for the SpesML project, the metamodel is easily expandable with further types, like e.g., external memory or watchdogs. Therefore, we are talking in general of *platform elements* when describing the content of execution platforms, and execution as well as communication elements are more specific types among them (see Figure 6-24).

For gaining an added value for systems engineering, an important aspect is to capture more than the structure of the hardware topology alone. The specification of relevant attributes of all technical entities is the key to enable verification and analysis of the technical architecture. For execution elements, this might entail to capture the available memory and computational power. For communication elements, the usual technical aspects are bandwidth, real time capabilities and similar quality of service attributes. As a demonstration within the SpesML project, we added *flash memory capacity*, *RAM capacity*, and *guaranteed ASIL safety level* as possible properties to execution elements. They correspond to our example properties of software elements, which were introduced above. Again, these properties can easily be expanded and adjusted to fit any other project environment. All these entities and attributes are finally relevant for the allocation models described in the next section.

The connection between multiple platform elements of an execution platform can be modeled via the generic *technical interface*. Actuators and sensors are often modeled outside the execution platform as part of mechatronic components, but it might be required that they are connected to some of the internal elements of an execution platform like execution elements. This is why execution platforms, and their contained elements, can also be connected to other technical components, again via the generic technical interface.

Allocation Models. A software allocation model describes a possible mapping of software elements of the software element architecture to execution elements of the execution platform. Thereby, it is an integrated platform-specific solution model for the SES. Such an *allocation* is necessary to identify the respective execution and communication resources for given software architecture artifacts, e.g., an allocation of a set of software elements to certain execution elements to describe the software deployment. This allocation enables further system

engineering activities, e.g., scheduling analysis of software elements. Finding an optimal or at least satisfying deployment allocation can be done via, e.g., SMT solvers, as shown in the SPES-based modeling tool AutoFOCUS 3[1] [Eder et al. 2020]. The basis for this is given by the properties that can be set for all software and platform elements, as mentioned in previous sections. The following example illustrates this. For instance, if all software elements have a *flash memory usage* property and all execution elements a *flash memory capacity* property, it is easy to define a rule that checks for all allocations of software elements to execution elements if the sum of flash memory usage across all software elements that are allocated to one execution element is not higher than the flash memory capacity of this specific execution element. Not only can such a rule be used to check manually made allocations, but it can also be used in algorithms to generate all possible allocation solutions that fulfill all such allocation rules for, e.g., deployments.

All allocations of software elements to execution elements in the software deployment model must be of cardinality *n:1*. This means, that each software element can only be deployed to exactly one execution element, while each execution element can execute *n* software elements. SpesML provides two different possibilities to describe this allocation of software elements to execution elements. The reason is that SysML has the concept of type definitions, called blocks, and their instances, called parts. For example, an execution element representing a special ECU type needs to be defined once as a block, but this block can then be instantiated several times in the diagrams as parts to model that for this special ECU type there exist several instances in the actual system. To support this concept, two allocations for software elements to execution elements exist in SpesML: a *software element instance to execution element instance mapping* for allocations between instances and a *software element type to execution element type mapping* for allocations between types.

Furthermore, SpesML supports the allocation of software element interfaces to execution element ports. This *software element interface to execution element port mapping* specifies which interfaces of software elements communicate via which ports of execution elements. For instance, it is possible that several software element interfaces need to be mapped to an execution element port, because the latter is a bus port on which the messages of multiple software element interfaces are transported.

When integrating operating system and middleware models, e.g., software partitions, it is important to note that this consequently influences the allocation models as well. Models from the application software, e.g., software elements, can conceptually be allocated to models of the operating system and middleware, e.g., software partitions, which in turn will be allocated to execution platform models. This means there are *software element to partition element mappings* as well as *partition element to execution element mappings*.

6.5.3 Tracing

As mentioned beforehand, logical components of a logical view are mapped to technical components of a technical view. This relationship between logical and technical architectural

[1] https://af3.fortiss.org/

components is *n:m* in general if the technical architecture is designed without the logical architecture in mind.

However, our overall approach is to provide meaningful tracing relationships between the model elements of the views. Therefore, it is suggested to develop an initial architecture in the technical view that has the same component structure as the logical architecture, which yields to a tracing relation where every individual logical component is traced to exactly one technical component on this layer of abstraction. It is possible that the technical view needs some components that are required only for purely technical reasons and hence do not have any counterpart in the logical view at all.

It is also recommended to structure the logical view in terms of grouping logical subcomponents that are realized in the same technical engineering domain, e.g., to group all logical subcomponents that are realized in software. This enables a possibly meaningful tracing relation where, e.g., individual logical components with software-related subcomponents are traced to exactly one software execution component in the technical view. Note that although we recommend such a strict relationship between logical and technical components, the interface refinement has a large impact on this detail of tracing between the logical and technical architecture.

Other relationships are possible, but we strongly encourage to manifest engineering decision in terms of a proper logical systems architecture that enables a relationship like described above. More details about tracing can be found in Section 6.7.

6.5.4 Well-Formedness Rules

The technical viewpoint incorporates all the general well-formedness rules (WFR) that were introduced for the whole SpesML concept due to, e.g., the UIM. In addition, the following rules were identified specifically for the technical viewpoint - mainly due to its deployment allocations and software element tracing. They were all realized as separate validation rules in the SpesML plugin as well (see Section 7.4).

WFR-T1: Interface deployment must be aligned with software element to execution element deployment.

- Software element interfaces can only be deployed to ports of execution elements to which the software element as owner of the interface is deployed itself. It is not allowed to deploy interfaces to ports of other execution elements to which the software element(s) have no deployment connection.
- *Reason*: It does not make sense to separate interfaces and their software elements between different physical elements in the real world.

WFR-T2: Each software element must be allocated to exactly one execution element.

- It is not allowed that a software element is without deployment allocation to an execution element. Also, there must not be a software element that is allocated to two or more execution elements.
- *Reason*: Every software element needs execution hardware to become part of a system. Also, every instance of a software element can only be executed on one execution component.

WFR-T3: Each software element interface must be either connected to another software element interface or allocated to a port of an execution element.

- It is not allowed that a software element interface is neither connected nor allocated, meaning that it either needs to have a connection to another software element interface or an allocation to an execution element port.
- *Reason*: It must be clear to what the output of a software element should be forwarded.

WFR-T4: All execution and communication elements must be connected to at least one other technical component.

- A communication element must be connected to at least one execution element. An execution element must be connected to at least one other technical component like another execution or communication element or to an external technical component like a sensor or actuator.
- *Reason*: A stand-alone communication element is useless because it cannot provide any actual communication. A stand-alone execution element that has no connection to other hardware does not make sense for a cyber-physical system since a characteristic of a cyber-physical system is its interaction between software execution and physical world.

WFR-T5: A logical subcomponent of a logical component that was tagged to one engineering discipline must not be traced to any technical element of a different engineering discipline.

- If a logical component is already indicated as related to a certain engineering discipline and it contains further logical subcomponents, these subcomponents can have only traces to technical elements of the same engineering discipline in the technical view.
- *Reason*: For example, pure software can only be handled via software elements in the technical view, since only these can and will be deployed and executed.

A special kind of well-formedness rules apply for allocations between software and execution elements. They are differentiated from the general well-formedness rules above, since they heavily depend on the properties which are annotated to software and execution elements. These rules are denoted as deployment allocation rules (DAR). Based on the introduced examples of properties for software and execution elements, the following three rules are presented as examples of possible DARs. These rules can easily be expanded and adjusted to fit the needs of any other project.

DAR-1: Every execution element must have enough memory capacity for all deployed software elements on it.

- If an execution element has software elements that were allocated to it for deployment, the execution element must have a flash memory capacity that is at least as large as the sum of all the flash memory that is needed by these software elements. If this is not the case, the execution element must either get a larger memory capacity or less software elements must be deployed on it.
- *Reason*: If there is too little memory space on the execution element, not all software elements can be deployed, or some will be overridden. This will usually result in wrong system behavior if the software can be executed at all.

DAR-2: Every execution element must have enough RAM capacity for all deployed software elements on it.

- If an execution element has software elements that were allocated to it for deployment, the execution element must have a RAM capacity that is at least as large as the sum of all the RAM that is needed by these software elements. If this is not the case, the execution element must either get a larger RAM capacity or less software elements must be deployed on it.
- *Reason*: If there is too little RAM space on the execution element, not all software elements might get executed correctly and in time. This will likely result in wrong system behavior if the software can be executed at all.

DAR-3: Every execution element must have an ASIL safety level that is high enough for all deployed software elements on it.

- If an execution element has software elements that were allocated to it for deployment, the execution element must guarantee an ASIL safety level that is at least as high as the highest ASIL safety level of all these software elements. If this is not the case, the execution element must either get certified for a higher ASIL safety level or the software elements that have the too high ASIL safety levels must be deployed to a more suitable execution element.
- *Reason*: If an ASIL safety level cannot be satisfied by the hardware, the safety regulations according to ISO 262626 cannot be met.

6.6 Modeling of Subsystems

SPES has defined the concept of granularity layers which allows the definition of subsystems with a decoupled development process due to the change of the current system under development (SUD). The decoupling of engineering processes and dividing them into individual engineering processes enables, e.g., reuse and integration of a supplier relations. Especially the structuring of models for the individual views, as described in previous chapters, and the concept of the Universal Interface Model (UIM) allow components from the technical view to be viewed as independent subsystems that can be developed with independent processes, methods, and tools. The methodology can then be applied again recursively to these subsystems. The integration of the subsystems back into the supersystem only works if the subsystem interfaces follow the UIM (see Section 6.6.3).
The following sections will explain what layers of granularity mean in terms of subsystems, how subsystems can be developed, and how they can be integrated into an overall system, called supersystem. This is closed by remarks about context and tracing in subsystems, and an explicit example of a subsystem.

6.6.1 Layers of Granularity

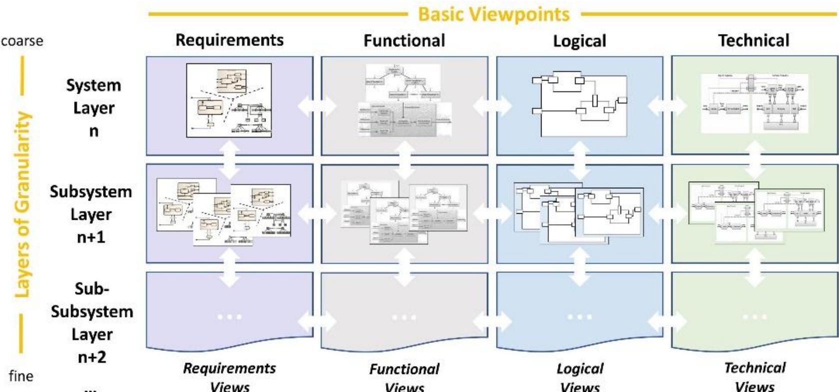


Figure 6-27: Layers of granularity in the SPES matrix [Pohl et al. 2012, modified].

The layers of granularity represent the vertical refinement axis of the SPES matrix and are orthogonal to the viewpoints (see Figure 6-27). We define a *layer of granularity* as the set of all subsystems identified in the technical view of the respective supersystem. Whether a technical component is refined in the current models of the supersystem's technical view or as a separate subsystem depends on the context of the development project. For instance, a subsystem might be defined if the development responsibility for the component changes due to outsourcing to suppliers or a suitable component already exists for reuse.

It is possible to declare all components in the technical view as subsystems and to further refine them at the next layer of granularity, resulting in a clear tree structure composition of the entire SUD into subsystems. However, it must be considered that with each transition to a new granularity layer, the model complexity increases. The reason for this is the context models of the subsystems, which cannot be taken over *1:1* from the models of the supersystem (see Section 6.6.4). It is important to note that subsystems always have an independent architecture and an associated technical implementation.

6.6.2 Development Styles with Subsystems

We distinguish between two basic approaches for (super)system and subsystem development, the *top-down* and the *bottom-up* approach, which can also be combined depending on the use case. Both will be explained in the following.

Top-Down. The *top-down development* starts with modeling the supersystem, usually at the first layer of granularity, and recursively defines subsystems for lower layers of granularity. This means that for this approach, subsystems are specified by the models of the supersystem and implement exactly the requirements of the supersystem. Based on this specification, model refinement is then performed in each subsystem on the next lower layer of granularity. Later, the UIM enables seamless reintegration of the subsystem models into the supersystem models. This approach is often found in the development of completely new systems.

Figure 6-28 shows an example of a top-down approach. The supersystem S is decomposed into three components A, B, and C. All three components should possibly be further developed at a lower layer of granularity so that the models of three subsystems are located at the second layer of granularity (one subsystem for each component). This approach can be repeated recursively to create a deeper nesting of granularity layers.

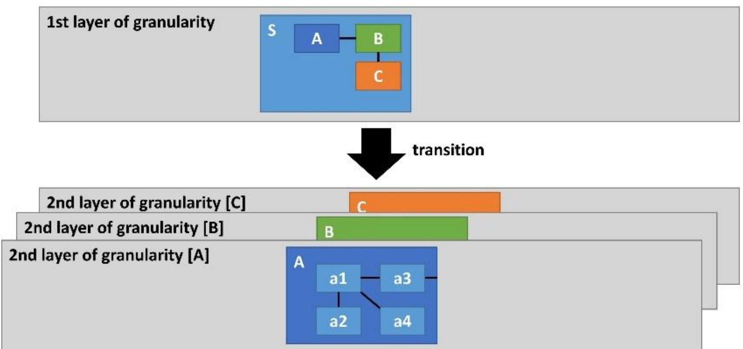


Figure 6-28: Example of a top-down development with subsystems.

Bottom-Up. The *bottom-up development* starts at a lower layer of granularity. Systems of lower layers are combined and integrated to form (super)systems of higher layers, i.e., we compose the supersystem from a set of already existing subsystems. This approach is often found in building block approaches, where already available systems are combined to develop the supersystem, or when existing system elements are reused. Hereby, it does not matter whether these subsystems already exist and can directly be reused or are first independently developed. In both cases, the integration of the subsystems into the supersystem will probably require some form of wrappers, because the initial interfaces of the independently developed subsystems might not perfectly match the interfaces of the supersystem to which the subsystems should connect. This is a major contrast to the top-down approach where the interfaces should match since they were defined in advance based on the supersystem.
It can be useful to combine the bottom-up and top-down approaches. Existing components can then be reused from previous projects or suppliers while at the same time it is still possible to specify some parts of the supersystem in advance and progressively develop them later based on these specifications or outsource them. Common to both approaches and their combination is that a transition between the different layers of granularity is required.
Figure 6-29 shows an example of a bottom-up approach. At the second layer of granularity, there are three existing subsystems, each describing a component of the supersystem. These components A, B, and C are then integrated at the first layer of granularity with their black box view to form the supersystem S. Internally, these subsystems can be further structured, which will be transparent to the combining system.

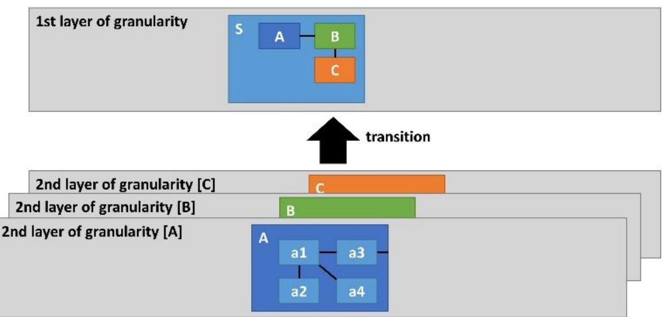


Figure 6-29: Example of a bottom-up development with subsystems.

In bottom-up development in particular, subsystem developers face the problem that the requirements for the subsystem have not yet been defined as it would have been the case during top-down development. However, to a certain extent this problem also arises in top-down development when subsystem development takes place alongside (super)system development and begins even before the requirements have been fully defined. In such situations, like in the highly work-sharing automobile industry, subsystems are developed first against assumed requirements, which are brought into the supersystem context later. To make this possible each subsystem must meet:

- a subset or refinement of the textual requirements of the supersystem, and
- all supersystem constraints with reference to the specific subsystem, and
- all additional constraints arising, e.g., from context relationships, and
- all requirements resulting from interface models in all views of the system.

Subsystems may still have additional requirements as well as their own context that must be considered for its development. SpesML offers two additional tracing relationships to express this situation, called *Require* and *Matches*, which are further explained in Section 6.7.11.

6.6.3 Integration of Subsystems

We use the paradigm of the UIM to integrate a subsystem into the supersystem. In particular, the integration of the subsystem is always done as a black box based on its syntactic and semantic interface.

In the top-down scenario, and by following the design recommendations presented in this book, we can identify in the technical view of the supersystem a component that represents the subsystem and is defined by the corresponding models in the logical and technical views of the supersystem. The UIM concept allows the subsystem to be integrated into both the logical view models (syntactically and semantically) and the technical view models (syntactically), each as a black box. If white box functions of the functional white box models can be uniquely assigned to a logical component, there is a canonical relationship between those white box functions in the functional view and the corresponding subsystem in the technical view.

In the bottom-up scenario, we usually refer to already existing subsystems, which were developed independently of the supersystem. To integrate existing subsystems into a system (on the next higher granularity layer), the subsystem must be prepared for:

- *Fulfillment of requirements of the supersystem*: The subsystem must fulfill the requirements specified by the supersystem. It is not necessary to adopt requirements in a *1:1* manner into the subsystem. The subsystem can also “overfulfill” requirements, i.e., offer more than is required by the supersystem. In SpesML, the *Matches* tracing relationship is introduced for this purpose.
- *Interfaces of the subsystems*: The subsystems are integrated into the supersystem according to the UIM paradigm, which means the subsystems’ interfaces must be conform to the UIM. In case of the bottom-up approach, a wrapper might be required to align the interfaces of the subsystem components to the related ones in the supersystem. This can either be done in the subsystems themselves or directly within the supersystem. It requires refinement or abstraction steps of the syntactic interfaces.

If we assume the proposed relation between the models of the logical and technical views of the supersystem (see Section 6.5), it is possible to integrate the subsystem models in both views. Since the models of the two views model different properties of the subsystem, namely the abstract interface behavior in the logical view and the technical implementation of the syntactic interface, both models contribute substantially to the definition of the subsystem and must be integrated. In addition, if white box functions that are realized together in the same subsystem have already been identified in the white box models of the functional view of the supersystem, those white box functions become system functions of the respective subsystem at the next lower layer of granularity. This means that meaningful tracing relationships between system functions of the subsystem and those white box functions exist.

6.6.4 Context of Subsystems

Since we change the scope of the SUD when moving to another granularity layer, the context plays an important role in modeling of subsystems.
The operational context on a granularity lower layer $n+1$ contains all context elements from the SUD on the higher layer n that have a connection to the selected component in the architecture on layer n. It should be noted that these components are not exact copies of the components of the architecture of the SUD on the higher layer n. Only the interface of a context component to the SUD of the lower layer $n+1$ is syntactically and semantically identical.
For a more detailed discussion of contexts with respect to different granularity layers (see Section 6.1).

6.6.5 Tracing Between Layers of Granularity

There is also tracing between layers of granularity. The whole concept of tracing and details about tracing between systems and subsystems on different layers are explained in Section 6.7.

6.6.6 Software Execution Subsystem as Explicit Example

The Software Execution Subsystem (SES) is an example of a specific subsystem that plays an important role when developing cyber-physical systems. An SES is formed if a

technical component on the system level is identified that represents a software application together with its execution platform. Such a technical component is further developed on the next lower granularity layer as a separate subsystem. A system can contain several SESs, each modeling a part of the whole application software of the system. For the SES, SpesML offers unique models that are only used within this specific subsystem, e.g., the software element architecture or the execution platform. For details see Section 6.5.2.

6.7 Tracing between Views and Layers of Granularity

The models of the SpesML views introduced above are not independent from each other. Therefore, it is important to understand how elements from these views can be properly traced and related to each other. Establishing tracing relationships is often a tedious task for the user, so this process has to be as easy as possible and should be supported by the tool. The amount of tracing relationships the user has to define manually must be kept at a minimum and should have a precise semantic meaning. SpesML introduces a set of stereotypes to allow easy setup of tracing matrixes (see Section 7) and other customizations.

6.7.1 Models in SPES

In SpesML, an architecture description of a *system under development* (SUD) is structured with the help of predefined viewpoints. Each viewpoint predefines a set of models and model elements, for modeling the respective view of the SUD.
The model elements are related to each other both within a single view (e.g. when models of a view are further refined) and across view boundaries (because the viewpoints model the SUD at different levels of abstractions). Note however, that relationships between model elements from different views are *n:m* in general.

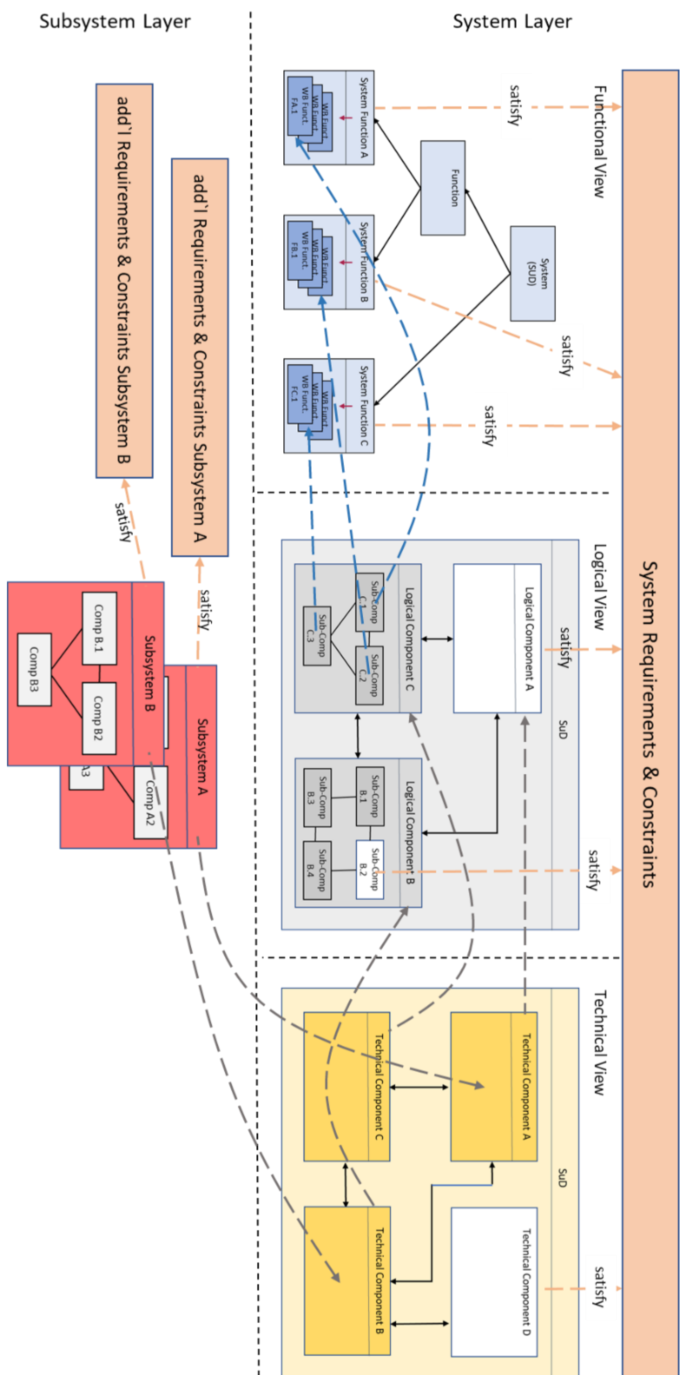


Figure 6-30: SpesML Model Relations Overview

This means, that looking from left to right in Figure 6-30 a model element in the left view is "implemented[2]" by *m* model elements in the right view, and vice versa, a model element of this right view may "implement" parts of *n* different model elements of the left view. Obviously, such a situation strongly limits the options for semantically expressive relations and analyses on the level of model elements. Therefore, in our approach a practical compromise (design recommendations) between generality and the possibility of enabling semantically expressive tracing relationships between model elements is introduced. There may be usage scenarios where a different approach has to be taken. This then may come at the expense of the semantically meaningful trace relationships.
Figure 6-30 gives an overview of the trace relations of an SUD across the different views and across layers of granularity, which are described below in more detail. In SpesML the *satisfy* relation for tracing between requirements and the model elements from the different views and the *realizes* and *realizes redundant* relations for tracing between the views (different architectures) are used.

6.7.2 Requirements Tracing

In SpesML, requirements are described as natural language statements. Requirements are not isolated model elements, but are typically related to other requirements and other model elements through tracing relationships (see Section 6.2)
Requirements may be refined into more detailed requirements (modeled through a *derived* relation from the more detailed requirement) or allocated to architectural elements in the views (modeled through a *satisfy* relation from the architectural element to the requirement). The combination of satisfy and derive links allows the representation of justification chains linking architectural elements to high-level stakeholder needs or to obligations arising from the development context, such as compliance.
Other tracing relations can be used in the context of modular subsystem development or to link simulation setups as verification evidence or explanatory information to requirements.
Note: In order to keep the number of trace links small, it is a good idea to already split requirements and differentiate at the requirements level.

[2] From a formal point of view this "implementation" can be modeled as a refinement relation between the models under consideration.

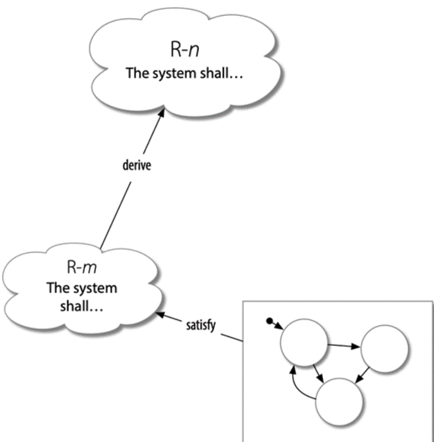


Figure 6-31: Requirements tracing.

6.7.3 Tracing between Elements in the Functional and Logical Views

For system developers it is important to understand how system functions (model elements of the functional view - i.e. system black-box and white-box functions) are implemented by model elements of the logical architecture (logical components of the logical view). As discussed above, designing a logical architecture that does not take into account design decisions from the functional architecture view may lead to an *n:m* relation, i.e., a white-box function is realized in the logical architecture by *n* logical components and a logical component realizes portions of *m* white-box functions, thus strongly limiting the semantics of the tracing relations.
In order to define semantically rich tracing[3] relations between the model elements of the views, SpesML suggests a *design pattern* that provides a closer connection between the system function models in the functional view and the structure of the logical architecture by using the concept of system function white-box models (see Section 6.3).
In order to build a bridge between the functional and the logical architectures SpesML recommends to design the white-box models (*causal chains)* of the system functions in the functional view with the structure of the logical architecture in mind: In a design decision, a structural architecture is developed from the set of white-box functions by uniquely mapping, i.e., tracing, white-box functions to logical components (see Figure 6-32). To that end the logical components are decomposed into subcomponents that implement exactly one white-box function. Subcomponents can be combined into logical architecture components by applying the Universal Interface Model (UIM – see Section 4.3). From a formal point of view, these subcomponents then represent a refinement of the white-box functions. In the tool this is

[3] Trace in this context means, that a logical component is linked to the functions modeled in the white-box function, expressing their relation.

modeled by a realize relation from the logical subcomponent to the white-box function. Composition of all these subcomponents yields the syntactic and semantic interface of the respective component (again by application of the UIM). The interfaces of such subcomponents (syntactically and semantically) are derived from the interfaces of the white-box functions via a refinement relation which is a well-formedness rule that can be verified. This results in a network of communicating logical subcomponents to which each white-box function of the functional view of the SUD is uniquely assigned.

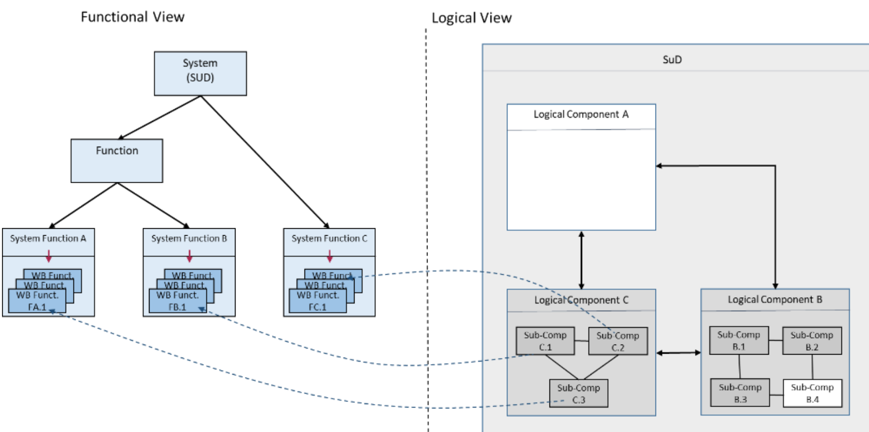


Figure 6-32: Mapping Functional View to Logical Architecture.

The functional model of a system can be viewed as a network of communicating instances of white-box function types, where in most cases the network would contain only one instance of each type. However, there may be some white-box functions where more than one instance of that white-box function exists, i.e., there may be white-box functions that occur in multiple white-box models. Mapping to logical components in such a situation needs a design decision. Our approach supports such a scenario as long as the whole (type of the) white-box function is implemented through the corresponding subcomponents. Two options are available:

- Each instance of the white-box function is implemented by a separate subcomponent in the logical view. This case will be modeled by the realizes (redundant) tracing relation.
- The white-box function is implemented through a shared subcomponent in the logical view. As the white-box function types are traced only, this case is modeled with the standard realizes relation.

Note: The logical architecture can also contain further (sub-) components that may result from additional requirements (i.e. requirements that are not addressed by system functions in the FVP). Those (sub-)components will not have a direct tracing relation to a white-box function. An example that may lead to such (sub)components are requirements that arise through design decisions (white boxes in Figure 6-33).

6.7.4 Definition of Software-Components in the Logical Architecture

In the SpesML approach predominantly *cyber-physical systems* are considered, i.e., systems that combine cyber capabilities (computation and/or communication activities) with physical capabilities (motions or other physical processes). The cyber parts of the system hereby control the physical parts[4] (physical components).

Often the behavior of the SUD as observable at the interface between the software- and the physical part (see Section 4.3) shall be modeled. In order to achieve this, the control parts must be designed as one or more independent components, already in the logical architecture structure model.

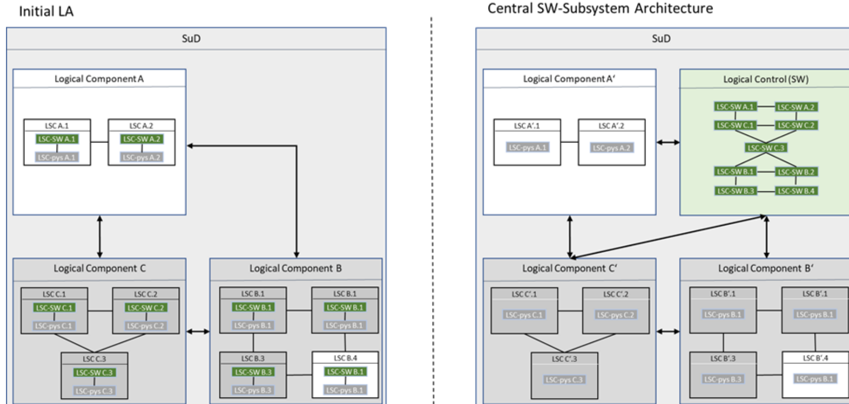


Figure 6-33: From Initial Logical Architecture to Central SW-Subsystem Architecture.

The following situations can occur:

- The split into cyber functions and physical functions has already been done during the design of the functional white-box models, i.e., the white-box functions which correspond to the cyber part of the SUD have been identified. In this case, there is already a structure of subcomponents in the logical architecture consisting of software components and physical components with appropriate interfaces between these subcomponents, which can be rearranged to form logical components that implement the cyber and the physical components of the SUD forming the *central SW-Subsystem architecture*. This approach has the advantage that a continuous semantical tracing between all modeling views of the SUD is achieved. The drawback is, that already in the functional models an implementation decision (this white-box function will be implemented in SW) has to be made which may not be appropriate in all cases.

---

[4] Here, physical parts stand for all parts of the SUD that are not part of the control component. Therefore, physical parts may also contain software shares.

- If the white-box functions are not already designed according to this split, the subcomponents in the logical architecture can be further decomposed into a cyber and a physical sub-subcomponent which now can be rearranged and composed to achieve the targeted structure (see Figure 6-33). This approach does not require implementation decisions already in the functional view, but has the disadvantage that the behavior models of the white-box functions can no longer be used directly to model the behavior of these sub-subcomponents. In addition, there is a disruption in the tracing relations between the modes of the functional view and the models of the central SW-Subsystem architecture since white-box functions are now split over several logical components.

Formally, the central SW-Subsystem architecture constitutes a refinement of the initial logical architecture. The initial logical architecture is therefore only an intermediate step in the development process, the models of which are no longer needed in the further course of development. SpesML recommends to adopt this refinement step in the white-box models of the functional view and to decompose the corresponding white-box functions into a cyber and a physical part as well. The subcomponents of the central SW-Subsystem architecture then *realize* the refined white-box functions.

6.7.5 Tracing Relationships between Elements in the Logical View and Technical View

As with the transition from the functional to the logical view, the relationship between logical and technical architectural components is *n:m* in general, if the technical architecture is designed without the logical architecture in mind.
Again, the goal is to create meaningful tracing relationships between the model elements of the views. Therefore, it may be appropriate to develop an initial architecture in the technical view that has the same component structure as the logical architecture, which yields a *1:1* tracing relation between logical and technical components on this layer of abstraction (Figure 6-34). The tracing is modeled through a *realizes* relation from the technical components to the logical components. To cover more technical detail, the components of this initial technical architecture are decomposed in further development steps.

Note:

- As with the logical architecture, the technical architecture can also contain further components that are defined by additional implementation requirements which will not have a direct tracing relation to a logical component.
- It is also possible that logical components are implemented multiple times by technical components in the technical architecture, e.g. to model redundancy in the technical view for the first time. The relations *realize (redundant)* are used to model this.

The syntactical interface of these technical components is then a refinement of the related logical components. The way, how this interface refinement is designed, has a major impact on the details of the technical architecture.

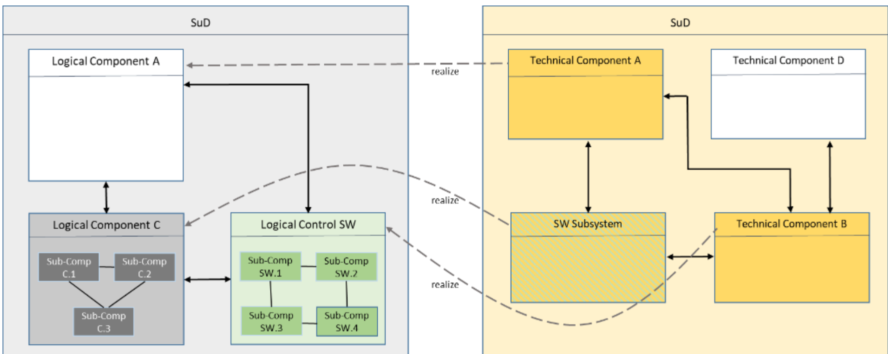


Figure 6-34: Mapping Logical Architecture to Technical Architecture.

6.7.6 Tracing Relations for the Software Execution Subsystem

The runtime part of the *software execution subsystem* (the *central SW-Subsystem* in case of a CPS) is described by a set of (executable) tasks that process inputs to outputs, or by a bus message catalog that contains all messages representing the technical data flow between these tasks (task architecture). (Section 6.5.2)
In general, the software tasks can be designed independent from the subcomponent structure of the logical control component. However, in practice these subcomponents will often be used to derive a set of software tasks necessary to implement the software execution subsystem, defining a *1:n* relationship between the subcomponents and the software tasks. This in turn defines a refinement relationship between the logical subcomponents of the logical control component and the tasks of the software execution subsystem. The tracing is again modeled through a realizes relation from the software tasks to the logical subcomponents.

6.7.7 Tracing of Context Elements

To model the interaction of the SUD with its environment (e.g. for simulation purposes) SpesML introduces context models for each view in SpesML (Section 6.1). Obviously, these context models are not independent of each other. Therefore, it is interesting to understand the relations between the elements of these context models.

Context Relations Between Functional and Logical View:

The tracing relations between context functions and context systems are analog to the relations between system functions and logical components: Context systems realize context functions. Context systems that provide redundant functions are modeled with a realize (redundant) relation.
Design recommendation: It is strongly recommended to model the structural context in the logical view in such a way that there is again an n:1 relation between context functions and

context systems (i.e., a context system realizes n context functions, while each context function is realized as a whole by exactly one context system or by multiple context systems in the redundant case).

Context Relations Between Logical and Technical View

At the highest level of abstraction, there is a 1:1 relationship between logical context systems (in the logical view) and technical context systems (in the technical view). Tracing relations between these context systems are again realizes and realizes (redundant) respectively.

6.7.8 Tracing between Layers of Granularity (Tracing between Subsystems)

The structuring of the models in the individual views described in the previous sections allows components from the technical view to be viewed as independent systems that can be further developed independently. Each *subsystem* is treated as an independent system under development[5]. It is important to note that subsystems always have an independent architecture and an associated technical implementation.
Hereby, it is irrelevant which methodology (processes, methods, tools) is used to develop a subsystem. In the SPES methodology all such subsystems of a supersystem are clustered in one *layer of granularity*. Typically, only a careful selection of the technical components will be processed as subsystems.
This definition has two important consequences: Firs, layers of granularity must not be confused with component decomposition. For layers of granularity the focus is on independence of the development processes and second, the subsystems of a granularity layer in general do not constitute a complete decomposition of the supersystem.
As subsystems are derived from components of the technical architecture, there always exists a (trivial) model relation between the subsystem and the respective technical component (due to the construction of the layer of granularity).
How architecture models of the subsystem can be derived from the models of the supersystems depends on the nature of the tracing relationships of the models in the supersystem.
As mentioned above, the model relationships between the models of the functional view and the logical view or the logical view and the technical view of an SUD are *n:m*, respectively, in the general case. In such cases, meaningful model tracing relations between the supersystem and the subsystem for the models of the functional or the logical view and model integration of the models of the subsystem into those of the supersystem cannot be identified.
Following the design recommendations from this document for each subsystem, as well as for the supersystem, achieves traces to a unique component in the technical view of the supersystem that represents the subsystem and can be traced back to the corresponding models in the logical architecture of the supersystem. The universal interface concept allows the subsystem model to be integrated into both the logical view models (syntactic and semantic) and the technical view models (syntactic), each as a black-box.

[5] A prominent example of a subsystem is the SW execution subsystem discussed above.

6.7.9 Tracing for Software Execution Subsystems

Following the design recommendations described above (i.e. identifying the sub-functions that will be implemented in SW already in the white-box models and building a 1:1 relation between the logical and technical components) and assuming that the models of the software execution subsystem follow the SPES approach, further semantically rich tracing relations across the layers of granularity can be defined.

If white-box functions that are realized in SW have already been identified in the white-box models of the functional view of the supersystem those white-box functions become system functions of the software execution subsystem (at the next layer of granularity), which means that meaningful tracing relationships (black-box tracing) between system functions of the software execution subsystem and those white-box functions exist:

- The software execution subsystem defined in the technical architecture of the supersystem becomes the new SUD in the next layer of granularity, and
- a logical component in the logical architecture of the supersystem that represents the software execution subsystem can be identified, and
- white-box functions in the functional view of the supersystem may become system functions on subsystem level. Whether a white-box function becomes a system function on subsystem level depends on how fine granular the white-box functions have been defined at the supersystem level: In case, there exist white-box functions that do not have a communication channel (interface) at the boundary of associated system function (i.e., there are *causal chains* of white-box functions within a system function that can be combined to form a larger white-box function without affecting the structure of the logical components) then the whole causal chain is traced to a system function on subsystem level (Figure 6-35).

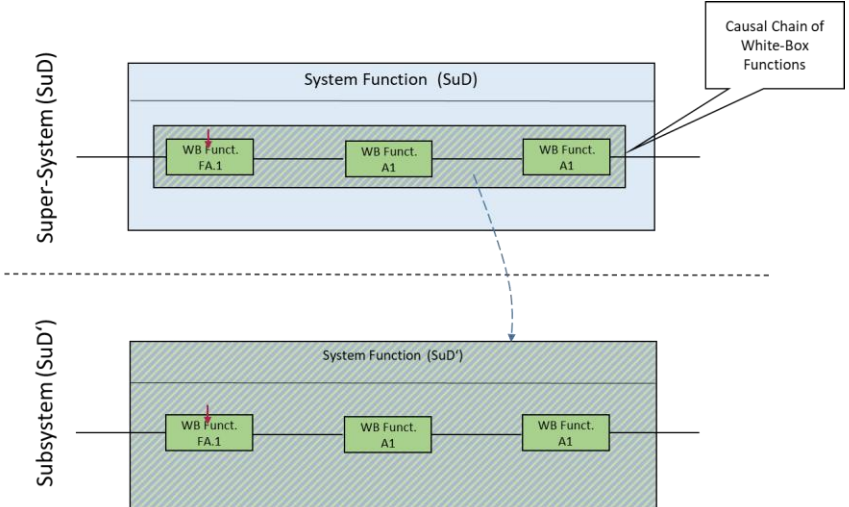


Figure 6-35: Causal chains of white-box functions.

6.7.10 Tracing of the Operational Context

The models of the SpesML views contain also models of the operational context, which also have a relation to the context of a subsystem. In general, the operational context of the subsystem is a projection of the context of the supersystem defined as follows:

- The operational context is always defined with reference to the system being developed. From the definition of a subsystem and given the proposed relations between technical and logical components, a subsystem is represented by a component in the logical architecture of the supersystem.
- The operational context of the subsystem contains all context elements from the supersystem that have a direct channel to the component representing the subsystem in the logical architecture of the supersystem.
- Furthermore, the operational context of the subsystem contains all components from the logical architecture of the supersystem that have a communication channel to in the component representing the subsystem. It should be noted that these context elements are not 1:1 copies of the components of the logical architecture of the supersystem, as their behavior might be additionally influenced other logical components or by the context of the supersystem. Only the syntactic and semantic interfaces of those components are identical.

6.7.11 Development Against Assumed Requirements

Subsystem developers face the problem that the requirements for the subsystem are not yet finally defined at design time of the subsystem. In such situations - particularly in the highly work-sharing automobile industry - subsystems are developed first against assumed requirements, which are brought into the supersystem context at a later time. This also means,

that the context, the subsystem operates in, is not known at design time of the subsystem. To overcome this problem, certain assumptions from a subsystem point of view must be made that need to be valid for the subsystem to be integrated into a supersystem.
In the case of integration on model level, requirements across layers of granularity are related with *match links*. SpesML offers two additional tracing relationships to express this situation (see Figure 6-36 for an example):

- *Require* (from model element to requirement): This relationship states that a model element - typically a function of the FVP or a component of the LVP, or the TVP - has certain expectations that are to be fulfilled by the context of the model element.
- *Matches* (from requirement to requirement): This relationship expresses that expectations are provided by properties guaranteed by the subsystem: The guarantees of the subsystem must include:
    - A subset or refinement of the textual requirements of the supersystem.
    - Supersystem constraints with reference to the subsystem.
    - Additional constraints on the subsystem arising, for example, from context relationships.
    - Requirements that result from the interface models of the supersystem in all three views.
    - Subsystems may still have additional requirements as well as their own context that must be considered for their development.

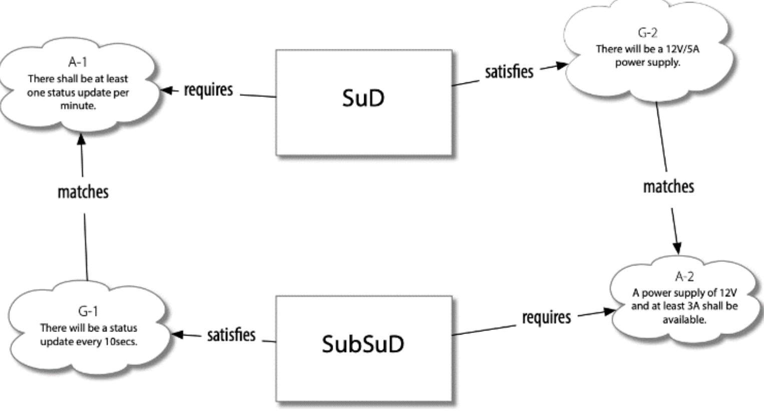


Figure 6-36: Matching guarantees to assumptions.

References

[32] Pohl, K., Hönninger, H., Achatz, R., Broy, M. (Eds.) (2012): Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology. Springer, 2012.

[33] Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, 2001

[34] Eder, J., Voss, S., Bayha,A., Ipatiov, A., Khalil, M.: Hardware architecture exploration: automatic exploration of distributed automotive hardware architectures. Software and Systems Modeling, Vo. 19, pp. 911-934, Springer, 2020.

[35] INCOSE. Guide for Writing Requirements v3, INCOSE, 2019.

[36] Mavin, A., Wilkinson, P.: A. Harwood and M. Novak*. EARS (Easy Approach to Requirements Syntax)*. In: 17th IEEE International Requirements Engineering Conference, pp. 317–322, 2009.

Rohit Gupta
Mathias Pfeiffer
Nikolaus Regnat
Bernhard Rumpe
David Schmalzing

7 A MagicDraw Plugin for the SpesML

This chapter describes the MagicDraw Plugin for the SpesML. It starts by giving an overview of the most important parts of the plugin. It then specifies how the SysML metamodel has been extended to cover the SpesML needs and how MagicDraw has been extended to allow the definition of expressions in state machines. Afterwards, it defines how the SpesML well-formedness rules have been implemented using MagicDraw validation rules. The last part of this chapter shows how simulation is realized in the context of SpesML.

7.1 Overview

Introducing MBSE into an organization is not an easy task. One needs to choose a modeling language, define a modeling method, and choose an appropriate tool. Moreover, language, method, and tool should be integrated, especially considering that the typical system architect is not also a modeling expert.

In the context of SpesML, the Systems Modeling Language (SysML) [OMG 2019], the SPES methodology, and the modeling tool MagicDraw from Dassault Systèmes [37] are combined. For this purpose, a dedicated MagicDraw SpesML Plugin has been developed that combines language, method, and tool in an easy-to-use way [38]. The plugin consists of the following parts:

- **MagicDraw SpesML Profile** – this profile contains all SpesML-specific stereotypes, tags, and further customizations according to the SPES methodology.
- **MagicDraw SpesML Model Template** – this template defines the initial model structure for easy creation of new SpesML modeling projects.
- **MagicDraw SpesML Perspectives** – these MagicDraw perspectives allow customization of the user interface, e.g., to better support novice and expert users.
- **MagicDraw SpesML Java Plugins** – these Java-based plug-ins extend the capabilities of MagicDraw, e.g., by adding well-formedness rules or simulation aspects.

7.1.1 MagicDraw SpesML Profile

One of the main features of MagicDraw is the domain-specific language (DSL) customization engine, which allows for adapting MagicDraw to a specific model domain and language profile. To this end, a specific SpesML profile has been defined as part of the MagicDraw SpesML profile.

A profile in MagicDraw does not only consist of stereotypes and tag definitions but also enables the definition of so-called customization elements. These MagicDraw-specific elements allow the definition of additional rules or context conditions for each stereotype. This facilitates not only the integration of parts of the SPES method but also directly influences the user experience.

All SpesML viewpoints (Requirements, Functional, Logical, and Technical) are covered, and the profile contains dedicated stereotypes that represent the specific SpesML model structure. Instead of using standard UML package elements, the profile defines individual stereotypes for each structural level. By defining customizations for these stereotypes only a certain set of additional elements can be created. For example, under the Logical Viewpoint (package) only the specific elements or diagrams that are part of the Logical Viewpoint to be created. This step already provides intuitive guidance to the user, as it combines the modelling language, method, and tool in an easy-to-use way. This is less error prone compared to the typical approach of simply giving users access to all UML/SysML elements without a defined scope and later providing guidelines on what to do [39]. It also ensures that all SpesML models are much more uniform, improving not only readability but overall model quality. Subsequently, it allows for easier integration of automation techniques, like document generation or any other form of accessing the model data using the MagicDraw API.

7.1.2 MagicDraw SpesML Model Template

A template is used whenever a user creates a new SpesML project. It consists of a dedicated icon, a description, and a predefined package structure within the containment tree based on the dedicated SpesML stereotypes (defined in the profile). While simple to setup, the model template not only makes things much easier for the user but also directly guides the user to follow the SpesML approach in a specific order. It does not only contain a predefined numbered package structure but can also be configured to contain certain other model elements (for example a Logical Context element), custom diagrams, matrices, tables, and relation maps. Figure 7-1 shows a new MagicDraw project based on the SpesML Model Template. These elements are discussed later in Chapter 7.2.

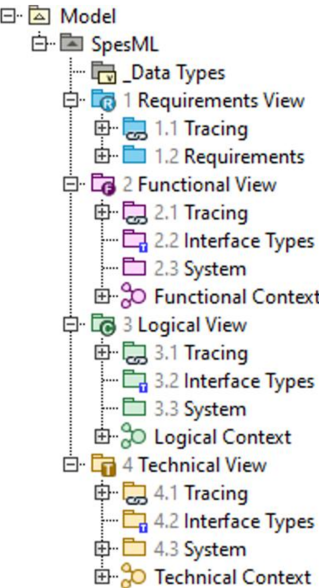


Figure 7-1: A new MagicDraw project based on the SpesML Model Template.

7.1.3 MagicDraw SpesML Perspectives

Perspectives customize the user interface of MagicDraw by removing toolbar menu entries, context menu actions and a wide variety of MagicDraw functionalities. MagicDraw offers a lot of features that are often overwhelming to new and inexperienced users. Reducing the user interface to a minimum necessary of a given context will help these users to focus on the relevant functionalities of the tool, while still allowing more experienced users to make use of the full potential by choosing a perspective with more functionalities. In the context of SpesML, the SpesML (Novice) and SpesML (Expert) perspectives to support this approach have been defined.

7.1.4 MagicDraw SpesML Java Plugins

MagicDraw offers a Java-based plug-in mechanism to extend the capabilities of the tool. For SpesML the following set of plugins have been developed to help enhance the user experience and add validation and simulation capabilities.

- **SpesML DragAndDrop Plugin** - this plugin allows to enhance the user experience by allowing drag&drop configurations e.g., to guide users what elements should / should not be dragged & dropped to certain SpesML diagrams.
- **SpesML FunctionsLanguage Plugin** – this plugin extends the capabilities of MagicDraw with a SpesML conform functions language that enables simulation.
- **SpesML Stereotype Plugin** – this plugin allows to automatically adds stereotypes to the elements created by the user. This improves the user experience as element are visualized consistently and only relevant properties are shown based on the corresponding stereotype customization.
- **SpesML SymbolTable Plugin** – this plugin creates a symbol table for the SpesML project and is used by the FunctionsLanguage and Validation plugins.
- **SpesML Validation Plugin** - this plugin bundles all SpesML specific validation rules into a SpesML specific validation suite that can be executed. It checks whether the modeled elements adhere to the SpesML specification.
- **SpesML Visualization Plugin** - this plugin allows to provide custom visualizations for dedicated SpesML elements under certain conditions. For example, elements that are marked “external” via a dedicated tag are shown in a specific color to easily distinguish them from other elements.
- **SpesML WebsiteLink Plugin** - this simple plugin adds a link to the SpesML website into the Help menu of MagicDraw. This allows users to easily access to the SpesML documentation directly from within MagicDraw.

7.2 Extension of the SysML Metamodel

One major part of the MagicDraw Plugin for SpesML is a dedicated *MagicDraw Profile* (see Section 7.1.1) that defines all needed elements of the Spes modelling language. A profile in MagicDraw consist of stereotypes, tag definitions, and customization elements. Stereotypes and tag definitions are extension mechanisms of UML/SysML and allow to extend the modelling language by defining new elements or add specific properties to elements. Customization elements are MagicDraw-specific and allow to define rules or context conditions for each stereotype. This also allows to not only embed parts of the SpesML method but also to directly influence the user experience design.

7.2.1 SpesML Elements

To extend the SysML Metamodel, the dedicated stereotypes are first created that represent the chosen SpesML package structure. Instead of using standard UML/SysML package elements, individual stereotypes for each individual package are created that is required to structure a SpesML MagicDraw project. When configuring the customizations for these stereotypes, only creation of specific other elements is allowed. For example, under the *SpesML Logical Viewpoint* package only specific elements or diagrams that are part of the Logical Viewpoint can be created. This step already provides instructive guidance to the user, as it combines the basic modelling language SysML, the SpesML method and the tool in an easy-to-use way. This is less error-prone compared to the typical approach of simply giving users access to all SysML elements without a defined context and later provide guidelines on what to do. It also ensures that all SpesML models are much more uniform, improving not only readability but also overall model quality. In addition, it allows for easier integration of automation techniques, be it document generation or any other form of accessing the model data, e.g., using the API.

Figure 7-2 shows an exemplary stereotype and its corresponding configuration from the SpesML profile. A profile is a special kind of package that extends a reference *meta model* by defining stereotypes. Each stereotype is based on an already existing element from the meta model, a specific *meta type*. For example, the stereotype «SpesML Logical Viewpoint» is defined with the meta type *Package* and comes with a distinct icon, setting it apart from normal packages. The corresponding customization element of the same name defines additional aspects of this stereotype. The attribute *abbreviation* defines the default name of the element when a user creates a new element of that type. The attribute *category* influences the user interface: when a user right-clicks on the viewpoint this element will show up in a category called SpesML Packages. The attribute *disallowedRelationships* is setup so that all possible relationships on this element are forbidden. The attributes *hiddenOwnedDiagrams* and *hiddenOwnedTypes* configurations hide all normal UML/SysML diagrams and elements from the user; these will not show up in the context menu of this element. The attribute *suggestedOwnedTypes* setting references only those SpesML stereotypes that a user can create under this element. Due to this configuration, users do not have to create generic SysML elements and manually apply stereotypes but can directly create the SpesML elements.

The «SpesML Logical Component» stereotype is defined with the meta type *Class* as SysML stereotypes in MagicDraw cannot be used as meta types. However, this can be achieved by inheriting the «SysML Block» stereotype. This way the new stereotypes works like a SysML block from the users' perspective. The «SpesML Logical Component» stereotype also inherits from an abstract stereotype «Traceable Logical Element». This stereotype is also used to setup dedicated Tracing Matrixes. This allows the decoupling of the Matrix definitions from the actual stereotypes, making them less error-prone to updates or extensions of the SpesML Profile.

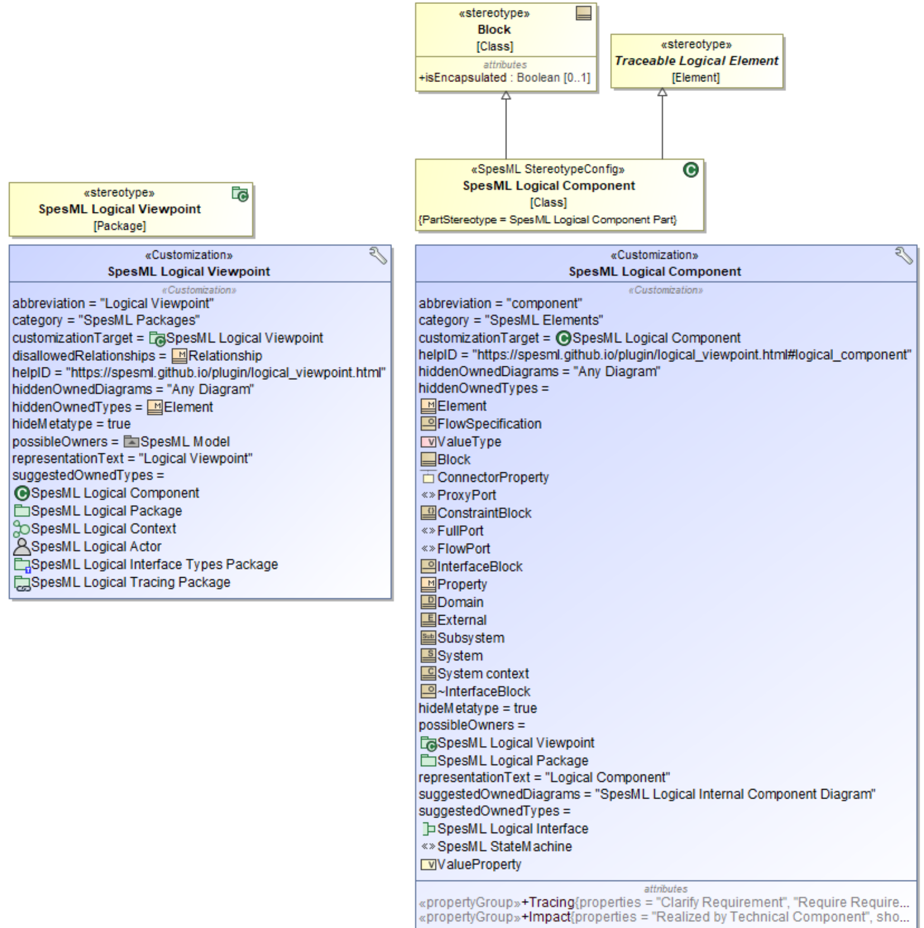


Figure 7-2: An excerpt of the SpesML profile stereotypes and customizations through which custom language elements in MagicDraw can be configured.

7.2.2 SpesML Diagrams

For SpesML, several custom diagrams have been implemented instead of using plain SysML diagrams. These custom diagrams have several benefits. The types and position of diagrams a user can create can be defined. In the SpesML *MagicDraw Profile* and corresponding SpesML *MagicDraw Model Template* a specific package is also defined. Using the MagicDraw customization capabilities, only specific diagrams are configured to be created. For example, the *SpesML Logical Internal Component Diagram* can only be created below the *SpesML Logical Components* and *SpesML Logical Context* elements. This provides enhanced user guidance directly through the MagicDraw user interface, thus relieving users from the burden of consulting additional documentation. Additionally, custom diagrams provide the possibility to have custom toolbars (only showing those elements required on the diagram) and allow for dedicated automation capabilities specific to certain diagram types.

Note: Due to restrictions in MagicDraw, it is currently impossible to create custom diagrams directly based on SysML diagrams. Instead, all SpesML diagrams are based on UML diagrams. Here custom diagrams based on the *UML Composite Structure Diagram* are mainly used.

Besides creating custom diagrams based on UML diagrams, MagicDraw also supports the creation of custom tables, matrixes, and relation maps. In SpesML, custom matrixes for tracing purposes are used, i.e., to allow the user to trace elements between the different views. Custom relation maps are used to provide additional impact analysis capabilities, e.g., to allow users to identify the impact of a change by displaying the tracing chain of an element. SpesML makes use of one custom table for handling *SpesML Requirement* elements. Figure 7-3 shows the custom diagrams that have been defined for the Logical Viewpoint. The *SpesML Logical Impact Map* and the *SpesML Logical Tracing Map* are both based on relation maps. The *SpesML Logical Internal Component Diagram* as well as the *SpesML Logical Test Case Diagram* are based on the *UML Composite Structure Diagram*. The remaining diagrams are custom matrixes.

SpesML Logical Impact Map
SpesML Logical Internal Component Diagram
SpesML Logical Test Case Diagram
SpesML Logical Tracing Map
SpesML LogicalTestToRequirement Matrix
SpesML LogicalToFunctional Matrix
SpesML LogicalToRequirement Matrix

Figure 7-3: The SpesML custom Diagrams, Matrixes, and Relation Maps for the Logical Viewpoint.

An example of the *SpesML Logical Tracing Map* can be seen in *Figure 7-4*. This special diagram is based on the MagicDraw relation map and allows to show relationships between elements. This enables users to easily explore tracing chains of elements.

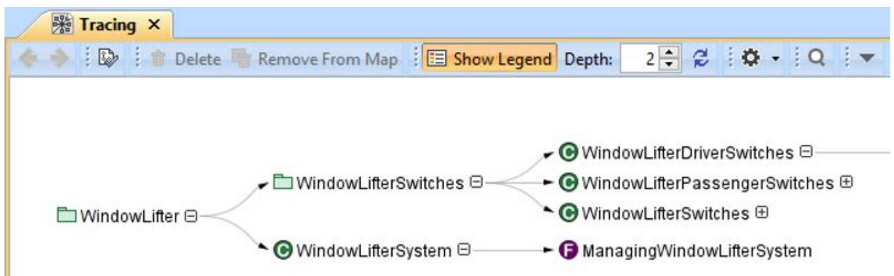


Figure 7-4: An example showing the SpesML Logical Internal Component Diagram.

There is also a dedicated *SpesML Logical Impact Map*. It works like the tracing map but shows elements that trace to a *SpesML Logical Component*, allowing for an easy impact analysis.

An example of the *SpesML Logical Internal Component Diagram* can be seen in Figure 7-5. The diagram is based on a *UML Composite Structure Diagram* and allows users to define the decomposition of *SpesML Logical Component* elements and connect *SpesML Interfaces* (based

on *SysML Proxy Port* elements) using *Connectors*. In addition, the diagram comes with a customized toolbar that only contains a minimum number of entries. In particular, the diagram does not allow users to create elements like SpesML Logical Components directly. Instead, a user must create all elements in the containment tree and drag&drop them into the diagram. This was a deliberate decision, guiding the user to create model elements in their appropriate locations before using them in a diagram.

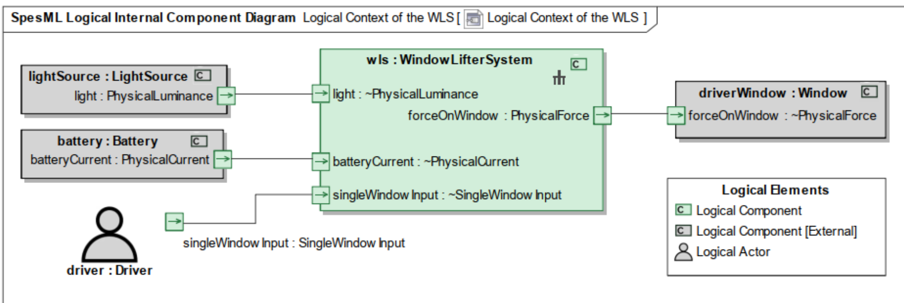


Figure 7-5: An example showing the SpesML Logical Internal Component Diagram.

An example of the *SpesML LogicalToFunctional* matrix can be seen in Figure 7-6. This matrix is based on a MagicDraw dependency matrix and allows users to define which *SpesML Logical Components* realize which *SpesML Functions* (from the Functional View). The matrix also allows users to quickly get an overview which functions have not been realized by any logical component.

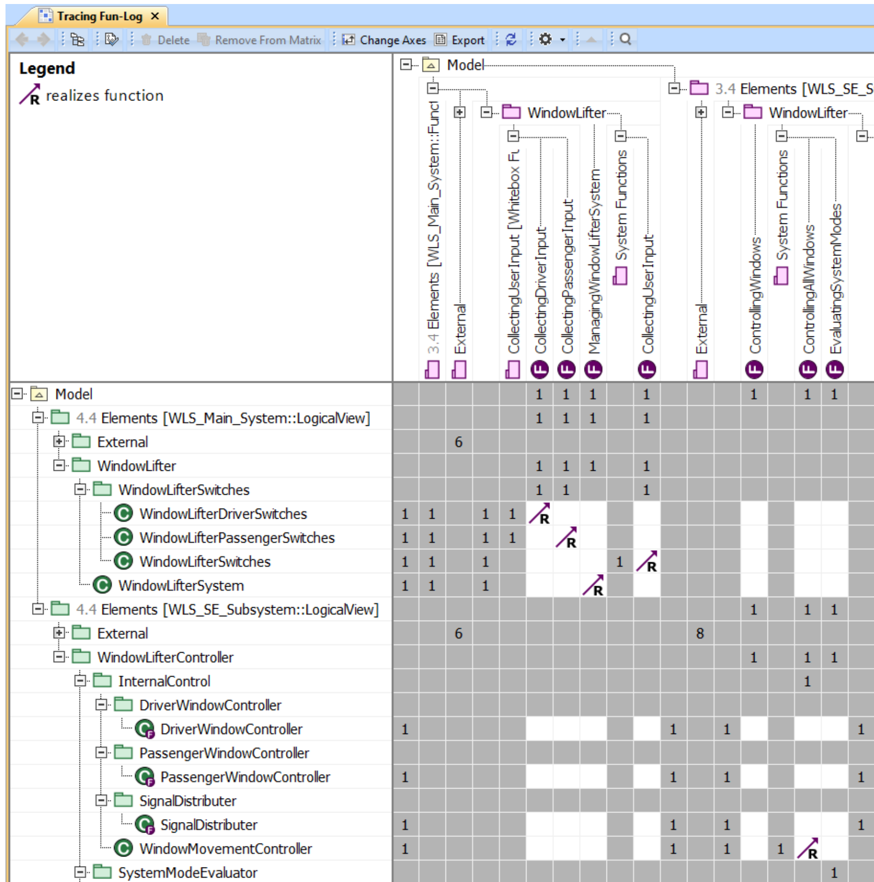


Figure 7-6: An example showing the SpesML LogicalToFunctional Matrix.

The *SpesML LogicalToRequirement* matrix is also based on a MagicDraw dependency matrix and allows users to get an overview of which *SpesML Logical Component* elements *satisfy* certain *SpesML Requirements* (from the Requirements View). The *SpesML LogicalTestToRequirement* matrix achieves a similar effect by allowing users to define verify relationships between *SpesML Logical TestCase Component* and *SpesML Requirements*.

In SpesML, all viewpoints have their own dedicated diagrams, relation maps, and matrixes that are pre-configured and ready to use.

7.3 Extension of MagicDraw with Expressions

This section introduces an expressive textual language to describe even complex behavior of systems and subsystems. The textual language is then embedded into the meta model extensions presented in the previous section, which focused on structure specifications. More specifically, the expression language is embedded into state machines, as they provide means to formally express behavior in SpesML. State machines describe (sub-)system states and transitions between them. The textual expressions are used to aptly describe source and target state, i.e.,

the guard condition that constrains the allowed configurations before the transition and the effect that describes the calculations and results of said transition. An introduction of the textual language itself can be found in Chapter 5. This section will now detail both the textual languages implementation as well as it's integration with MagicDraw elements, particularly those of the SpesML meta-model.

7.3.1 Implementation of a Textual Language

To design and implement a textual language component, it is necessary to repeat several standard tasks such as to define a grammar, implement a parser, and implement data classes. Furthermore, a modern and feature-rich language requires the use of advanced technologies such as symbol tables that allow for validated references between model elements. This is particularly important with regards to the need of a well-defined language (see Chapter 5) that is described using MontiCore [40]. MontiCore is a workbench and a framework for engineering modular textual modeling languages from *context-free grammars* (CFGs) [41]. These CFGs define a language's concrete and abstract syntax. From a CFG, MontiCore generates the basic infrastructure for the engineering of modeling languages, which includes a parser, a symbol table, model consistency and model transformation infrastructure, and a common workflow. The following describes some key steps of this workflow that lead to the implementation of a well-defined textual language component.

As shown in Chapter 5, SpesML's textual components need to be able to express literals, operators, as well as reference both executable functions and existing graphical model elements. Instead of redefining these language components, MontiCore's rich language component library[6] is employed. Listing 7-1 shows the basic structure of the CFG for SpesML's textual language component. The grammar *ExpressionsBasis* defines core interfaces and imports the kinds of symbols necessary to serve as hook points for other library language components. The *ExpressionsBasis* library defines symbols for *Types* (of all kinds), *Functions*, *Variables* and *TypeVariables*. These symbols are of general form and can be used in any context, as they do not define nor require a specific syntax. To be able to express common literals such as characters: 'c', Strings: "text", Booleans: "true", "null", or numbers 10, -23, 48l, 23.1f, the grammar is based upon the aptly named *MCCommonLiterals*. Besides literals, the textual language needs to be able to reference executable functions and existing graphical elements such as flow or value properties by their name. For this purpose, the grammar extends the library *CommonExpressions*. The library provides the means to model arithmetic, comparisons, variable use (v), attribute use (o.att), method calls (foo(arg, arg2)) and brackets (exp). As expressions should be able to describe timed behavior of system- and white-box functions, they need ways to describe input and output histories. For this purpose, the CFG further extends the library *StreamExpressions* which provides special syntax constructs for the most common operations on streams such as appending and concatenating.

---

[6] monticore.de

```
01 grammar SpesMLExpressions                                   MC-Grammar
02   extends de.monticore.expressions.ExpressionsBasis,
03           de.monticore.literals.MCCommonLiterals,
04           de.monticore.expressions.CommonExpressions,
05           de.monticore.expressions.StreamExpressions,
06           de.monticore.expressions.AssignmentExpressions
07 {
08   Start Expression ;
09   ...
10 }
```

Listing 7-1 The MontiCore grammar of SpesML's textual elements. The grammar extends from MontiCore's rich library of textual language components such as common literals, common expressions, and more specific expressions such as for streams and assignments.

Additionally, both state machines and executable functions need a way of assigning values to both temporary variables as well as flow and value properties of the respective system components, i.e., SysML blocks. For this purpose, the CFG extends from the library *AssignmentExpressions* which contains assignment expressions like =, +=, etc. and suffix and prefix expressions like ++ or --. Finally, assembling the sub-languages into a coherent language is done automatically by MontiCore through the libraries interface *Expression* as well as by defining *Expression* as the start rule.

Besides this re-use of existing library language components, the CFG *SpesMLExpressions* defines new elements to integrate the textual expressions with SysML's graphical elements. Listing 7-2 shows these additions. Names in SysML and thus SpesML may include spaces and are typically delimited by single quotes. A token *SysMLName* is thus introduced which allows for single characters and escape sequences. As the token is delimited with single quotes, the lexer is instructed to cut off the first and last letter when storing the parsed text of the token. Introducing such a *SysMLName* token is not sufficient however, as it still needs to be linked to the *Expression* interface. This is done be extending all non-terminals of the re-used language components where MontiCore's *Name* token was previously used. Specifically, the extension points are the extension of *NameExpression* to *SysMLNameExpression* and *FieldAccessExpression* to *SysMLFieldAccessExpression*.

```
01 token SysMLName                                             MC-Grammar
02   = '\'' (SingleCharacter|EscapeSequence)+ '\''
03   : setText(getText().substring(1, getText().length()-1)) ;
04
05 SysMLNameExpression extends NameExpression
06   = SysMLName ;
07
08 SysMLFieldAccessExpression extends FieldAccessExpression
09   = Expression "." SysMLName ;
```

Listing 7-2 The body of the CFG defines an extension of Names to allow for SysML-specific names. All relevant expression non-terminals are then extended to use the new token.

```
01 grammar SpesMLEffects extends SpesMLExpressions        MC-Grammar
02 {
03   interface Effects ;
04   start Effects ;
05
06   SingleAssignment implements Effects
07     = assignment:Expression ;
08
09   MultipleAssignments implements Effects
10     = ( SingleAssignment ";" )* ;
11
12   Block implements Effects
13     = "{" MultipleAssignments "}" ;
14 }
```

Listing 7-3 SpesMLEffects defines an extension to SpesMLExpressions for parsing the effects of state machines. The common interface Effects denotes the starting rule. Then three versions of effects are introduced: single assignments without any delimiters, assignment lists with delimiting semicolons and finally blocks which wrap assignment lists.

For the purpose of parsing and subsequently checking the effects of state machines, an extension to the general *SpesMLExpression*s grammar is defined in Listing 7-3. Effects can take three different shapes: single assignments without any delimiters (x=5), assignment lists with delimiting semicolons (x=5; y=6;), and finally blocks ("{x=5; …}") which wrap aforementioned assignment lists in brackets. For this purpose, *SpesMLEffects* extends *SpesMLExpressions* and introduces a common interface *Effects* which also serves as the starting rule. Then the three shapes of *Effects* are constructed iteratively. A *SingleAssignment* is simply an *Expression* and is only checked later as an actual assignment. This is necessary due to the inherent left-recursiveness of the desired *AssignmentExpression*-non-terminal combined with MontiCore's underlying parser technology. Building on *SingleAssignment, MultipleAssignements* allows repeated use of single assignments by separating them via a semicolon. The *Block* finally wraps the *MultipleAssignements* in curly braces.

MontiCore uses Context Conditions (CoCos)[7] to provide language engineers the means to check the correctness of parsed models. CoCos are a useful tool for restricting an otherwise lax grammar. They can effectively be used to provide meaningful and context-sensitive feedback to the modeler, instead of typically cryptic parser errors. This feature is particularly useful here, as SpesML attempts not to evaluate, but to guide modelers to a semantically well-founded and methodically sound development process. Such a CoCo is used to check each *SingleAssignment* to be an *AssignmentExpression* by checking it's AST-object to be an instance of the generated class *ASTAssignmentExpression.* Calling the CoCo is handled by MontiCore's infrastructure and makes use of the double-dispatch visitor pattern. This greatly streamlines the language engineering process and reduces the potential for errors as language engineers are only responsible for implementing the check itself, instead of being required to provide any framework- or boilerplate-code.

[7] MontiCore Reference Manual, Hölldobler et al, 2021. Page 221ff.

7.3.2 Adapting Graphical Elements to MontiCore Symbols

An integrated language requires a seamless connection between graphical and textual modeling. Thus, to embed an external, textual language into an otherwise graphical DSL in MagicDraw, it must be integrated in such a way that it: (1) accepts only the allowed set of syntactic sentences, and (2) these are validated for well-formedness with respect to their context. It is thus necessary for graphical elements to become referenceable from the textual elements. It is important to note that it is not sufficient to merely reference elements. As the first section in Chapter 5 motivates, it is necessary to provide checks and analysis for models. To realize this, MagicDraw's elements are adapted to MontiCore's symbol infrastructure. Defining symbols in MontiCore is done via CFGs. The only difference to typical language definitions, is that no concrete syntax is provided. Listing 7-5 shows the CFG for SpesML symbols.

```
MC-Grammar
01 scope symbol SpesMLPackage = Name ;
02
03 scope symbol SpesMLComponent = Name ;
```

Listing 7-4 Spanning the scope tree are the packages and components. Defining a scope for each of these assures that graphical elements can uniquely be classified and thus identified.

To be able to integrate the previously introduced Expressions, Effects, and executable functions, the grammar extends from those three. Through this extension mechanism, MontiCore generates the relevant adaptation between the three language components and allows for cross-references. In addition, the library of object-oriented symbols, *OOSymbols*, is extended. This grammar defines symbols for object-oriented types, methods, and fields. Most of the symbol infrastructure can be re-used from library components.

```
MC-Grammar
01 grammar SpesMLSymTab
02   extends SpesMLExpressions,
03           SpesMLEffects,
04           e.monticore.mf.MontiFun,
             de.a.symbols.OOSymbols
05 ;
```

Listing 7-5 MontiCore grammar for MagicDraw's SpesML elements. Extends from the previously introduced Expressions, Effects, and executable functions to be able to integrate all three. Further extends from OOSymbols, a library of object-oriented

As Listing 7-4 shows, the SpesML symbol table additionally defines dedicated elements for packages and components. The keyword *symbol* denotes the introduction of a new symbol class. Symbols live in *scopes*, which form a hierarchy and provide means of nesting, separation and regulate visibility of symbols at certain locations of the tree -like structure. This allows them to simultaneously span a scope containing child symbols, and also be a child symbol of some other scope, effectively allowing the construction of a tree. The root scope of this tree is called the global scope. Within the root scope reside artifact scopes. For SpesML, an artifact scope spans the scope hierarchy of a single MagicDraw project. Generally speaking, each *symbol* has to be identifiable via a *Name*. Finding a symbol within a scope or scope structure is called resolving.

```Java
01 public class SpesMLGenitor
02   extends ModelHierarchyVisitor {
03
04   private BiMap<Element, ISymbol> symMap;
05
06   private Stack<ISpesMLSymTabScope> scopeStack;
07
08   @Override
09   public void visitPackage(Package element) {
10     var builder = SpesMLSymTabMill.packageSymbolBuilder();
11     var symbol = builder.setName(element.getName()).build();
12     symMap.put(element, symbol);
13     var enclosingScope = scopeStack.peek();
14     enclosingScope.add(symbol);
15     var spannedScope = SpesMLSymTabMill.scope();
16     spannedScope.addSubScope(spannedScope);
17     scopeStack.push(spannedScope);
18     traverse(element);
19     scopeStack.pop();
20   }
21
22   ...
23 }
```

Listing 7-6 The scope genitor simultaneously builds the symbols and scope skeletons, as well as creates a bi-directional map between MagicDraw elements and MontiCore symbols.

MontiCore's resolving strategy handles nested scopes and qualified names in an intuitive Java-like manner. Most of the remaining symbols will be entirely re-used from the *OOSymbol*s library. The interface types form an exception and are introduced as an extension to *OOType*s as Listing 7-7 shows. This change was made necessary by the requirement to distinguish between value properties and flow properties at symbol level. The former will be constructed as regular OOTypes. Differentiating between values and flows is relevant when employing stream functions, which should only be allowed on flows, not values.

```MC-Grammar
01 scope symbol SpesMLInterfaceType implements OOType
02   = Name;
```

Listing 7-7 Interface types are introduced as an extension to the library symbol OOType. This change was necessitated by the desire to differentiate at a symbol level between value properties and flow properties. This is relevant as stream functions should only be allowed on flows, not values.

After having defined the necessary symbols, the symbol table needs to be constructed from MagicDraw's elements. This construction is handled in two phases: 1) build symbols and scope skeletons for each relevant element and then, once all skeletons exist, 2) link the appropriate and now existing elements. The first phase is handled by a so-called *genitor*. The second phase, i.e., linking, is necessary to establish type traces. These traces provide the typing information of flow and attribute properties. Such a two-phase construction is adapted from MontiCore's own symbol table instantiation[8]. However, instead of traversing the AST of a textual language produced by a MontiCore parser, the containment tree of MagicDraw is traversed instead.

[8] MontiCore Reference Manual, Hölldobler et al, 2021. Page 179ff.

The genitor begins by preparing a fresh global scope and pre-loading primitive symbol types such as *int*, *bool*, and *String*. An empty map is initialized which stores the bi-directional links between MagicDraw's containment tree elements and corresponding symbols. This enables referencing symbols from graphical elements and vice-versa. An additional stack holds the created scopes which in turn store the symbols. The stack is initialized with the global scope as single top-most element. After setup, the genitor begins visiting the containment tree. Listing 7-6 gives an overview of the class, its most important fields, and exemplary shows the creation of a package symbol and scope. MagicDraw's visitor infrastructure is used by extending from *ModelHierarchyVisitor*. The genitor is passed to the containment tree's root package, after which the *visitPackage* method is executed my MagicDraw's runtime. The genitor builds a symbol using the MontiCore-generated builder class and sets the name appropriately. All symbols share the common *ISymbol* class provided by MontiCore's language libraries. The symbol is linked to its corresponding element in the containment tree via the bi-directional *symMap*. The symbol is then added to its enclosing scope, which is determined as the top-most element on the *scopeStack*. As packages also span a scope, such a scope is created and inserted as a sub scope. Then the newly created scope is pushed to the stack, marking it as the enclosing scope for child elements visited through the *traverse* method. The scopes used and produced are *ISpesMLSymTabScope*s which are compatible with all language components (expressions, functions, type symbols) of the SpesML symbol table infrastructure. The depth-first traversal, which is hand-coded in the *traverse*-method, iterates all children of the current element and triggers each child to accept the genitor. This in turn triggers MagicDraw's visitation procedure. The traversal algorithm is greatly inspired by MontiCore's own traversal infrastructure. After traversal is completed and potential children's symbols and scopes are built and linked, the previously created and pushed scope is removed (popped) from the stack. Analogous to packages, the symbols and scopes of components, as well as interface type definitions and value type definitions are also constructed and linked. The only perceivable difference is the use of a different builder: *SpesMLComponentSymbolBuilder* for component definitions, *SpesMLInterfaceTypeBuilder* for interface type definitions, and *OOTypeSymbolBuilder* for value type definitions. These builders are all generated by MontiCore from their definitions in their respective CFG.

```java
01 public class SpesMLGenitor
02   extends ModelHierarchyVisitor {
03   ...
04
05   @Override
06   public void visitProperty(Property element) {
07     var builder = SpesMLSymTabMill.fieldSymbolBuilder();
08     var symbol = builder
09         .setName(element.getName())
10         .setIsStatic(false).build();
11     symMap.put(element, symbol);
12     var enclosingScope = scopeStack.peek();
13     enclosingScope.add(symbol);
14   }
15 }
```

Listing 7-8 Properties represent fields of either components, value type definitions, or flow type definitions. Compared to packages, components, or type definitions, they may be static and do not span a scope, nor do they invoke further traversal.

When looking inside components, interface type definitions, and value type definitions, they all contain fields which in turn are typed by interface or value types. Through the hand-coded traversal, these fields are visited as the leaves of the traversal tree. Listing 7-8 shows the visitation of these fields as their MagicDraw implementation *Property*. The *FieldSymbolBuilder* is invoked to create the symbol. Additionally, to having a unique name, a field may be static class- or instance bound, i.e., static or non-static. Fields of components, value types, or flow types are all non-static. The builder offers a *setIsStatic* method which is invoked with the Boolean *false*. Then, the newly built symbol is linked to its corresponding element in the containment tree through the *symMap* before being added to the enclosing scope currently on top of the *scopeStack*. As fields do not have children, no spanned scopes are created, nor is the traversal algorithm invoked. Similar to packages, components, or type definitions, enum definitions are also visited and produce *OOTypeSymbols*. Compared to *Property*s however, their fields, i.e., the enum literals, are set to be static through the builder. This enables their reference through their defining type, i.e., the enum definition.

```java
01 public class SpesMLCompleter
02   extends OOSymbolsVisitor {
03
04   @Override
05   public void visit(FieldSymbol symbol) {
06     var element = symMap.get(symbol);
07     var typeName = ((TypedElement) element).getName();
08     var typeSym = symbol.enclosingScope.resolveType(typeName);
09     var type = SymTypeExpressionFactory.createType(typeSym);
10     symbol.setType(type);
11   }
12 }
```

Listing 7-9 The second phase links the types to the field symbols. To do so, MontiCore's traversal infrastructure is used. Each FieldSymbol is visited, the type name retrieved from the matching containment tree element, and the type resolved through the symbol table.

After the first phase, the *FieldSymbol*s require further linking. A MontiCore traverser traverses the scope tree and employs a visitor to visit all *FieldSymbol*s. Listing 7-9 overviews

this visitor. The visiting method retrieves the containment tree element through the *symMap*. From this element, the name of the referenced type is retrieved. This name is then resolved in the symbol table, starting from the current *FieldSymbol*. Technical reasons require the construction of a symbol-type-expression from this symbol. This is then set as the field type. This works the same way as for value properties or proxy ports of components, value properties of value or flow type definitions, as well as Enum literals. The later will simply reference their owning Enum definition as their type.

7.3.3 Integration of Textual and Graphical Model Elements

The adaptation of graphical elements to MontiCore's symbol table infrastructure permits the integration of textual with graphical model elements. To do so, the expression's enclosing scope is embedded into the scope hierarchy spanned from graphical elements. This then enables checking the type-correct usage of graphical model elements within textual model parts. This integration is implemented as part of a set of validation rules (see Section 7.4).

First, the symbol table for the graphical model elements is built as described above. Simultaneously, the textual model elements are processed with the textual MontiCore language previously developed. If the textual expression is syntactically valid, i.e., can be parsed into an AST structure, MontiCore continues with scope creation. Otherwise, a corresponding error message is reported back to the user. The expression's AST that results from parsing has, at this point, a placeholder enclosing scope. This enclosing scope is then set to the component's scope from the graphical model, in which the textual expression resides. Listing 7-10 shows this process. The link between embedding component and its symbol is established using the bi-directional *symMap* reference linking containment tree elements to symbols and vice versa.

```
Java
01 void embed(ASTExpression ast, Class component,
02     BiMap symMap) {
03         ISymbol componentSymbol = symMap.get(component);
04   IScope enclosingScope = componentSymbol.getSpannedScope();
05   ast.setEnclosingScope(enclosingScope);
06 }
```

Listing 7-10 Embedding the textual expression's scope into the scope hierarchy spanned from graphical model elements. The expression has previously been parsed to an AST. The graphical model element representing the embedding component was retrieved using MagicDraw's API. The bi-directional symMap was constructed from the containment tree.

After embedding the respective scope hierarchies into each other, MontiCore's type check facade[9] can now be employed to check the reference- and type-correct usage of named graphical elements, as well as executable functions and their combination with the library of literals and operators. For this purpose, the libraries provide composable implementations of Type Derivers. These derivers can iteratively derive a type from an expression by visiting the AST. They are implemented modularly using MontiCore's visitor infrastructure. This permits them to each only provide the derivation functionality for AST classes introduced by the respective language

[9] MontiCore Reference Manual, Hölldobler et al, 2021. Page 361ff.

component. The integration of results as well as synchronization during run-time is handled entirely by the type check facade. Error reporting and interfacing with the modeler is handled by MagicDraw's validation rule mechanism.

7.4 Validation Rules

SpesML defines a set of well-formedness rules to ensure model completeness and correctness (in the sense of the SPES methodology). Within the MagicDraw SpesML Plugin, a subset of these well-formedness rules has been implemented using the MagicDraw validation engine. This means that not all the well-formedness rules have been fully implemented with the validation rules specified by the validation engine. The validation engine allows to create custom validation rules that can be implemented as Object Constraint Language (OCL) expressions or as binary validation rules (in Java). The OCL is a formal language used to define constraints and are used to specify the invariant conditions that must hold for the system under development. The binary validation rules are written using custom Java classes when the specification of constraints using OCL becomes hard, e.g., during string manipulation. These rules are grouped into validation suites such as the *SpesML Validation Suite*. This validation suite can be started manually from within MagicDraw, and any rule violations will then be reported as warnings or errors by MagicDraw. When customizing the SpesML Plugin it is possible to adapt the existing or add new validation rules, depending on the specific needs. The well-formedness rules associated with the specific rules have been discussed previously. Further, a special kind of well-formedness rules, termed deployment allocation rules (DARs), are also discussed in Chapter 6. DARs depend on the properties annotated to the software and execution elements. The following validation rules have been defined for SpesML and are associated with those well-formedness rules.

7.4.1 ChannelOutDirection

This validation rule is implemented as a binary validation rule (Java) and checks that all *SpesML Channel* elements are defined with the direction *out*. If an *in* direction is needed, the *is conjugated* property must be used. The rule implements the following well-formedness rule:

- WFR-L8: The SpesML Channels must have the direction 'out' as it determines the actual direction of the channel represented by the flow property.

7.4.2 ConnectedProxyPortsOfBlock

This validation rule is implemented as a binary validation rule (Java) and checks that all SpesML structural elements (e.g., *SpesML Function*, *SpesML Logical Component*, …) that are decomposed into *SpesML Part Properties* (e.g., *SpesML Function Part*, *SpesML Logical Component Part*, …) have their ports properly connected on a *SysML Internal Block Diagram* based diagram. The rule implements the following well-formedness rule:

- WFR-L6: SpesML structural elements that are decomposed into other logical components such as the SpesML Part Properties must be connected properly.

7.4.3 DistinctTaskAllocation

This validation rule is implemented as a binary validation rule (Java) and checks that each Software Element is allocated to exactly one Execution Element. The rule implements the following well-formedness rule:

- WFR-T2: Each software element must be allocated to exactly one execution element.

7.4.4 EmptyName

This validation rule is implemented as a binary validation rule (Java) and checks if specific SpesML elements have a name. The rule implements the following well-formedness rules:

- WFR-L1: All SpesML Elements must have a name since elements are referenced via their names in a textual language specification.

7.4.5 EmptyType

This validation rule is implemented as a binary validation rule (Java) and checks that all SpesML elements that can be typed are properly typed. For example, every logical component should have at least one logical interface. This means that attributes of value types must have a type, flow properties must have a type, proxy ports must have a type, and parts and value properties of a block must have a type. The rule implements the following well-formedness rules:

- WFR-L2: Every attribute of value types, flow properties, proxy ports, and value properties and parts of every block must have a type to have a relevant and observable behavior in the system.

7.4.6 EnoughMemorySpaceForTask

This validation rule is implemented as a binary validation rule (Java) and checks that *SpesML Execution Elements* have enough memory capacity for all deployed/allocated *SpesML Software Elements* on it. The rule implements the following well-formedness rule:

- DAR-1: Every execution element must have enough memory capacity for all deployed software elements on it.

7.4.7 EnoughRAMCapForTask

This validation rule is implemented as a binary validation rule (Java) and checks that *SpesML Execution Elements* have enough RAM capacity for all deployed/allocated *SpesML Software Elements* on it. The rule implements the following well-formedness rule:

- DAR-2: Every execution element must have enough RAM capacity for all deployed software elements on it.

7.4.8 MatchingTaskPortAllocation

This validation rule is implemented as a binary validation rule (Java) and checks that the SpesML port deployment is aligned with *SpesML Software Elements* to *SpesML Execution Element* deployment. Software element ports can only be allocated to ports of execution elements to which the software element as owner of the port is deployed itself. The rule implements the following well-formedness rule:

- WFR-T1: Interface deployment must be aligned with software element to execution element deployment.

7.4.9 NoPorts

This validation rule is implemented as a OCL 2.0 validation rule and checks that all SpesML structural elements (e.g., *SpesML Function*, *SpesML Logical Component*, …) have at least one *SpesML Interface*. The rule implements the following well-formedness rule:

- WFR-L2: Every logical component must have at least one logical interface, meaning every block must own at least one proxy port.

7.4.10 OnlyOneBehavior

This validation rule is implemented as a OCL 2.0 validation rule and checks that certain SpesML elements, such as the functional context, logical context, software element, mechatronics component, electrical components, and so on, have a maximum of one behavior definition (e.g., state machine). The rule implements the following well-formedness rules:

- WFR-L3: All logical components need to have an interface behavior model in the form of a decomposition or a state machine.

7.4.11 PlatformElementConnections

This validation rule is implemented as a binary validation rule (Java) and checks that each *SpesML Execution* and *SpesML Communication* element is connected to at least one other (technical) component. The rule implements the following well-formedness rule:

- WFR-T4: All execution and communication element must be connected to at least one other technical component.

7.4.12 PortTiming

This validation rule is implemented as a binary validation rule (Java) and checks that the SpesML *Timing* settings on *SpesML Logical Interfaces* are consistent. An example is that a state machine should not assign multiple values to logical output interfaces such that the behavior of the logical component cannot be analyzed. The rule implements the following well-formedness rule:

- WFR-L5: State machines must not assign multiple values or sequences to logical output interfaces at the same time.

7.4.13 PureSoftwareTracing

This validation rule is implemented as a binary validation rule (Java) and checks that a logical subcomponent of a *SpesML Logical Component* that was tagged as software must not be traced to any other element than a *SpesML Software Element*. The rule implements the following well-formedness rule:

- WFR-T5: A logical subcomponent of a logical component that was tagged to one engineering discipline, e.g., software, must not be traced to any technical element other than a software element.

7.4.14 STMEffects

This validation rule is implemented as a binary validation rule (Java) and checks the validity of effects in transitions and initial sequences in state machines. It checks a) the assignee, i.e., channels or attributes, b) the assigned value, i.e., a literal or potentially complex expression, and c) their combination. This rule makes use of the textual language parts introduced in

Chapter 5 as well as their integration with MagicDraw described in Section 7.3. It comprises a multitude of checks in a single validation rule. The assignee, on the left-hand side, is checked if it exists, meaning if it is an attribute of the owing block. Attributes may be flow properties of proxy ports or value properties of the owing block. The value, on the right-hand side, is checked if it is a valid expression according to the textual expression language specification in Chapter 5. This means both the assignee and the assigned value should consist of sound syntax and have type-correct references to existing elements only. Such existing elements might be the flow properties and the value properties of the owning block, executable functions (see Chapter 5), or static elements such as enumerations. Finally, the assignee and the value are checked for compatibility with regards to their type. It thus implements the following well-formedness rules:

- WFR-L3: All logical components and sub-components need to have an interface behavior model in form of decomposition or a state machine.
- WFR-L4: State machines must only refer to interfaces defined by the logical component in which they are defined.
- WFR-L5: State machines must not assign multiple values or sequences to logical output interfaces at the same time.

7.4.15 STMGuards

This validation rule is implemented as a binary validation rule (Java) and checks the validity of transition's guards in state machines. This rule makes use of the textual language parts introduced in Chapter 5 as well as their integration with MagicDraw described in Section 7.3. The guard is checked to be an opaque element containing a string. This string is checked to be a valid expression according to the textual expression language specification in Chapter 5. This entails both being of sound syntax as well as type-correct references to existing elements only. Such existing elements might be the aforementioned flow properties and value properties of the owning block, executable functions (see Chapter 5) or static elements such as enumerations. Finally, the type of the guard expression is assured to be of Boolean type. It thus implements the following well-formedness rules:

- WFR-F10: Each value that can be sent over a mode channel must have a corresponding state in the mode model.
- WFR-F11: Transitions in the mode model do not have any guards or actions.

7.4.16 SufficientAsilLevelForTask

This validation rule is implemented as a binary validation rule (Java) and checks that the *SpesML Execution Elements* have an ASIL safety level that is high enough for all deployed/allocated *SpesML Software Elements* on it. If the ASIL safety level is not high enough, then the execution element must be certified for a higher safety level, and if the safety level is too high, then it should be deployed to a more suitable execution element. The rule implements the following well-formedness rule:

- DAR-3: Every execution element must have an ASIL safety level that is high enough for all deployed software elements on it.

7.4.17 TaskPortConnection

This validation rule is implemented as a binary validation rule (Java) and checks that each *SpesML Software Element* port is either connected to another *SpesML Software Element* port

or allocated to a port of an *SpesML Execution Element*. The rule implements the following well-formedness rule:

- WFR-T3: Each software element interface must be either connected to another software element interface or allocated to a port of an execution element.

Tool-Specific Validation Rules

Other validation rules have also been defined as part of the MagicDraw SpesML plugin that are tool-specific and are not associated with any well-formedness rules described previously in Chapter 6. The following list of tool-specific validation rules along with a rationale have been defined for the SpesML plugin in MagicDraw.

7.4.18 EmptyEnum

This validation rule is implemented as a OCL 2.0 validation rule and checks that all *SpesML Enumeration* elements define at least one enumeration literal. This rule ensures that enumeration elements are not meaningless when there is no enumeration literal defined. This rule is part of a validation suite that is configured to run automatically.

7.4.19 PartsAndValueProperties

This validation rule is implemented as a binary validation rule (Java) and checks that all SpesML structural elements (e.g., *SpesML Function*, *SpesML Logical Component*, …) do not contain value properties if they also contain part properties. This rule ensures that only those blocks that do not contain any parts are allowed to have value properties configured. This rule is part of the validation suite that is configured to run automatically.

7.4.20 PortCycles

This validation rule is implemented as a binary validation rule (Java) and checks that in each SpesML diagram that is based on a *UML Composite Structure Diagram*, in each cycle of ports at least one port of the cycle is delayed. A port is called *delayed* if its specification defines initial sequences for its flow properties. This rule ensures that the compositions are well-defined for the ports. For atomic components, component modelers may explicitly define outgoing ports as delayed. For composed components, the delayed property of a port is derived from the component composition. This means that an outgoing port of a composed component is delayed if all paths in the component's topology to that port are delayed, i.e., there exists at least one port on each path that is delayed. On the other hand, incoming ports are never delayed.

7.5 Simulation

The simulation of SpesML system models is the execution of the behavioral descriptions of elementary systems and their communication in accordance with their composition. Simulation enables agile system model prototyping and development through behavior exploration, debugging, validation, parameter optimization, and various kinds of automated analyses. These can in turn facilitate a pervasive development approach through continuous integration providing early and automated feedback in response to changes.

The simulation of SpesML system models features both synchronous and asynchronous event-based communication of interactive, distributed systems. The simulation realizes the semantic mapping of SpesML system models to Focus [42] supporting three timing domains, namely synchronous (time-aware, synchronous communication), timed (time-aware, asynchronous communication), and untimed (time-unaware, asynchronous communication). The execution of behavioral descriptions of different timing domains can be mixed in a single simulation, as components can be composed of subcomponents of different timing domains. Simulation and real-time, however, are decoupled, enabling to run the simulation faster or even slower than real-time.

7.5.1 Concept

Simulation of SpesML system models is realized as an extension of the SpesML MagicDraw plugin. The simulation plugin adds a user dialog in MagicDraw that allows users to run tests against SpesML system models. When executing a test, the plugin compiles the test and the system models in a multi-step process via MontiArc [43] to Java code, runs the simulation, protocols system states and communication, and feedbacks the test results back to MagicDraw.

During the compilation, the simulation plugin first translates SpesML system models to MontiArc component models, subsequently generates Java source files, and finally compiles these to Java class files. While this multi-step compilation process produces some overhead, it also enables the reuse of MontiArc, which already realizes the Focus semantics. Using MontiArc avoids a MagicDraw-specific implementation and thus enables tool interoperability via MontiArc component models. SpesML, MontiArc, and other frameworks that compile to MontiArc can integrate. Furthermore, MontiArc being a textual language, system engineers can switch between graphical and textual modeling.

After the compilation, the simulation plugin executes the simulation, that is, the generated and compiled Java code. The plugin then runs the simulation until a predefined termination criterion is met. During execution, the simulation logs communication messages and system states, and after test execution, the plugin displays the test results in MagicDraw.

7.5.2 MontiArc Background

MontiArc is a textual architecture description language (ADL) for modeling component and connector (C&C) architectures. Central to MontiArc are component descriptions that define stream processing functions of communicating systems. Each component description is a declaration of a component's interface, structure, and input-to-output behavior. A component's interface consists of typed and directed input and output ports. Connectors between the components' interfaces compose these to form topologies of communicating systems. The behavior of a component results from the composition of the behavior of its subcomponents or is defined by an atomic behavior description.

MontiArc component descriptions are strongly typed. Only ports with compatible interfaces can connect, and expressions in behavior descriptions must be well-typed. The type of a port defines the possible messages the port can receive or send. The available types are defined by class descriptions in a textual UML/P [45] class diagram.

Like SpesML, MontiArc's semantic domain is FOCUS. Models are made executable by means of a Java code generator that generates a behavior simulation from component descriptions. MontiArc supports three of FOCUS' timing domains, namely time-synchronous

streams and timed and untimed event streams. For a component's atomic behavior description, MontiArc supports statecharts as a common behavior description that MontiArc interprets differently with respect to the timing domain, either translating them to Mealy and Moore machines or event automata.

7.5.3 Translation

To use MontiArc's simulation capabilities, SpesML system models are exported and translated to MontiArc component descriptions, including elementary behavior descriptions. Furthermore, SpesML-type definitions are exported and translated to UML/P class diagrams.

As both modeling languages, SpesML and MontiArc, are a frontend to Focus, they exhibit similar concepts. Therefore, the translation from SpesML to MontiArc is mostly straightforward, except for a few minor noteworthy differences.

Model elements of a textual modeling language are referenced by name. They reside in a scope or namespace and must be uniquely identifiable. Which element is referenced depends on the reference location. In contrast, in a graphical modeling language, model elements are often referenced through their object identifier rather than a name in the model. Graphical modeling tools load all models into storage and assume no external tools that modify the models' source files. The object references must be resolved and converted to unique name references when translating from a graphical to a textual modeling language. Since the SpesML, as a graphical modeling language, does not utilize scopes, these must be constructed artificially. The translation, therefore, constructs a scope structure alongside the SpesML containment tree, and the unique name of a model element is identified by its location in the containment tree. The translation replaces object references through qualified names accordingly.

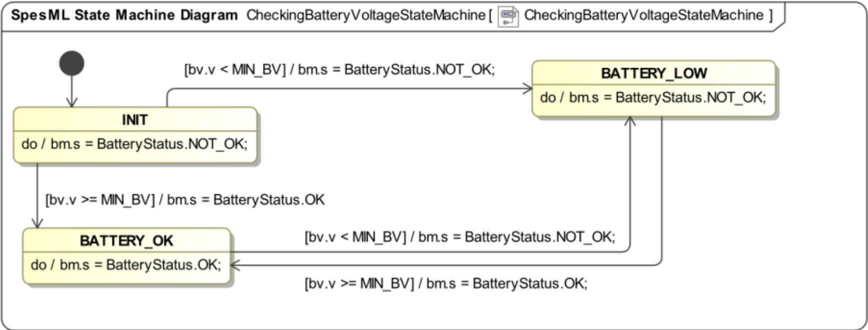


Figure 7-7: The figure shows an example of SpesML behavior specification in form of a state machine. The example is part of the running example. Names are shortened for better readability.

Similarly, to SpesML, MontiArc distinguishes between atomic and composed components. Figure 7-7 gives an example of the translation from SpesML to MontiArc for atomic components using the component specification CheckingBatteryVoltage. Its behavior is defined by the state machine shown in Figure 7-7. Additionally, the component specifies a component variable, an incoming and an outgoing port, called MIN_BV, bv, and bm, respectively. An excerpt of the result of the translation of this component specification is shown in Figure 7-8. The specification begins with the keyword component followed by the component's name (l. 1). The elements of the component are defined inside the curly brackets.

The port declaration (lines 2 and 3) consists of the keyword port followed by the port's direction, type, and name. In the SpesML, ports are of an interface type that consists of channels. A channel has a data type. In MontiArc, ports simply have a data type; there are no interface types. The translation from SpesML to MontiArc removes the interface types. A MontiArc component has a port for each channel of a port of a SpesML component. The type of the port in MontiArc corresponds to the type of the channel, and its name is a combination of the SpesML port and channel name. In the example, the port bv of the example has a channel v of type Integer, which translates to a port bv_v of type Integer in MontiArc.

```
1  component CheckingBatteryVoltage {
2    port in Integer bv_v;
3    port out BatteryStatus bm_s;
4
5    Integer MIN_BV = 10;
6
7    automaton {
8      initial state INIT;
9      state BATTERY_LOW;
10     state BATTERY_OK;
11
12     INIT -> BATTERY_LOW [bv_v < MIN_BV] / bm_s = BatteryStatus.NOT_OK;
13     … // further transitions
14   }
15 }
```

Figure 7-8: The figure shows an excerpt of the component shown in Figure 7-7 in the textual syntax of MontiArc. To avoid redundancy, the excerpt shows only one of the state transitions, the other state transitions translate similarly.

The component variable declaration (line 5) consists of the variable's type, name, and initial value, all directly derived from the SpesML component specification. In contrast to SpesML, the state machine is not a diagram of its own but also an element of the component declaration, here called automaton (lines 7-14). The elements of an automaton, namely states and transitions, are defined in another set of curly brackets. States translate straight forward into state declarations (lines 8-10), a state declaration consisting of the keyword state and the state's name, with the modifier initial defining the initial state. A transition (line 12) consists of the transition's source and target state, followed by the transition's guard and action. In MontiArc, transitions have an implicit order defined by their location in the text file. This order can influence the component's behavior in case of non-deterministic state machines. In SpesML, there is no order of transitions; thus, the translation does not guarantee any order. While the translation of transitions is mostly straightforward, expressions accessing channels in guards or actions are modified to refer to the corresponding ports in MontiArc.

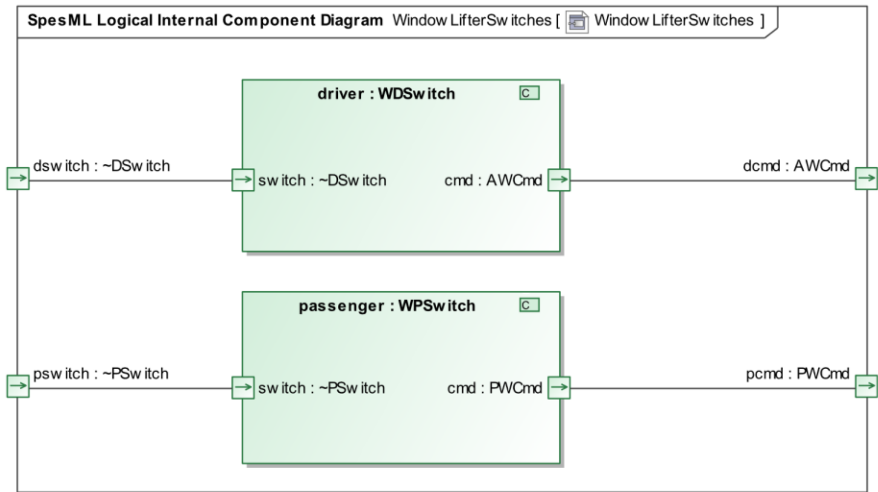


Figure 7-9: The figure shows an example of SpesML composed component. The example is part of the running example. Names are shortened for better readability.

The translation of a composed component follows a similar pattern. Given the SpesML component shown in Figure 7-9 consisting of two subcomponents, the translation results in the MontiArc component shown in Figure 7-10. Again, the inner elements of the component are defined in the curly brackets. Like the automaton of an atomic component, the subcomponents and connectors are part of the component description instead of a separate diagram. Each subcomponent declaration (lines 7-8) consists of a reference to the component declaration and the subcomponent's name. While SpesML component declarations are referenced through object references, in MontiArc they are referenced by name.

```
1  component WindowLifterSwitches {
2    port in DSwicht dswitch_s,
3         in PSwitch pswitch_s;
4    port out AWCmd dcmd_c,
5         out PWCmd pcmd_c;
6
7    WDSwitch driver;
8    WPSwitch passenger;
9
10   dswitch_s -> driver.switch_s;
11   pswitch_s -> passenger.switch_s;
12   driver.cmd_c -> dcmd_c;
13   passenger.cmd_c -> pcmd_c;
14 }
```

Figure 7-10: The figure shows an excerpt of the component shown in Figure 7-10 in the textual syntax of MontiArc.

The connectors (lines 10-13) similarly reference the source and target ports per name. If the referenced port is of a subcomponent, then the port reference is qualified via the name of the subcomponent. While in MontiArc, subcomponents and ports are implicitly ordered by their location in the text file, this order does not affect the component's behavior. SpesML, being a

graphical modeling language, has no order of subcomponents; thus, the translation does not guarantee any order.

MontiArc component descriptions are only an intermediate step of the simulation execution. The simulation plugin further translates the intermediate descriptions to Java and ultimately to byte code. The translation to Java is not shown here and instead can be found in a previous work [44]. While the Java code generated by MontiArc is used as provided, the simulation plugin also generates additional Java files, namely the test drivers that execute the simulation.

7.5.4 Test Execution

With the simulation plugin, a component can execute inside a test case. For the execution of a component, the component needs inputs to its input channels. The test case represents the frame, i.e., the context in which a component executes.

Test cases in SpesML are represented by a separate element kind, named test case component. The element kind separates the system under development from elements only used for testing. Creating a test case component registers that component for test execution.

While test case components are components themselves, they can only have a single outgoing port whose type and name are predefined. This port indicates the result of the test case, reporting success or failure. The test case component can be decomposed like any other component. The most basic test case specification, shown in Figure 7-11, consists of the system under test and an oracle that provides the system's inputs and evaluates the system's outputs, determining the test's success or failure. The oracle is a common component implemented by the systems engineer, or more specifically, by the test case engineer.

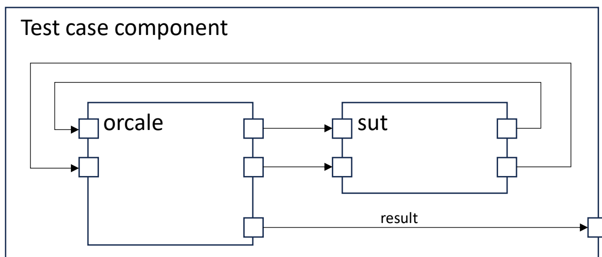


Figure 7-11: The figure shoes the basic concept of a SpesML test case. The test case consists of two subcomponents, the system under test (SuT) and the oracle that provides the system's inputs and evaluates the system's outputs. In case of success or failure the oracle sends a corresponding message over the single output port.

The test case scaffold shown in Figure 7-11 is only one option for defining a test. A test case component can be decomposed arbitrarily; only the outgoing port must be populated. For example, the system under test could be replaced by a conglomerate of various systems to test the interaction of these systems. Similarly, the oracle can be divided into several components.

While the result port of a test case component reports the success or failure of the corresponding test, it is not sufficient to just determine the termination of the simulation execution. In the case of a faulty oracle, success or failure may never be detected, and as components execute indefinitely, a fault can lead to a test that never terminates. The simulation execution, therefore, provides several termination criteria. One criterion is a specific event on a predefined port, namely a success or failure on the result port. Tests in the time-synchronous or

timed event domain terminate after a predefined number of clocks, which can be configured in the plugin settings.

The test case component and all other test components, e.g., the oracle, are also converted into MontiArc components when executing the test. For the test case component, the plugin additionally generates Java code that implements the test driver. This driver realizes the simulation core loop and checks the result port for messages. Furthermore, it implements the test termination and creates the rest report after test execution.

References

[37] MagicDraw Enterprise. 2023. URL: https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/

[38] Regnat, N., Gupta, R., Jansen, N., Rumpe, B.: Implementation of the SpesML Workbench in MagicDraw. Gesellschaft für Informatik e.V. (S. 61-76), 2022.

[39] Regnat, N.: Why SysML does often fail - and possible solutions. Gesellschaft für Informatik e.V., pp. 17–20, 2018.

[40] Hölldobler, K., Kautz, O., & Rumpe, B.: MontiCore Language Workbench and Library Handbook: Edition 2021. Aachen: Shaker Verlag, 2021.

[41] Chomsky, N.: Three models for the description of language. IRE Transactions on Information Theory, 1956.

[42] Broy, M., Stolen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement. Springer 2001.

[43] Ringert, J.O., Rumpe, B., Wortmann, A.: Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. Aachener Informatik-Berichte, Software Engineering, Band 20, Shaker Verlag, Dec. 2014.

[44] Haber, A.: MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems. Aachener Informatik-Berichte, Software Engineering, Band 24, Shaker Verlag, Sep. 2016.

[45] Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer International, 2016.

Henning Femmer
Maximilian Junker
Marcel Goger
Samson Groß
Ralf Hocke
Stefan Setzer

8 Case Studies

8.1 Parking-Lock (Schaeffler)

8.1.1 Domain context

The Schaeffler Group has been driving forward groundbreaking inventions and developments in the field of motion technology for over 75 years. With innovative technologies, products, and services for electric mobility, CO_2-efficient drives, chassis solutions, Industry 4.0, digitalization, and renewable energies, the company is a reliable partner for making motion more efficient, intelligent, and sustainable – over the entire life cycle. The motion technology company manufactures high-precision components and systems for drive train and chassis applications as well as rolling and plain bearing solutions for a large number of industrial applications.
Like other players in this sector, Schaeffler is also confronted with the challenge to retain an efficient and high-quality product development in a field of a steadily rising system and development complexity. A key factor to preserve the success is the continuous optimization of systems engineering processes and methods. Out of this reason, Schaeffler has a high intrinsic interest in collaborating with partners from industry and research to support such research activities on the one hand and on the other hand to get valuable input for the optimization of own development methods.

8.1.2 Case Study Introduction

The case study shows extracts of the development of an *integrated parking lock (IPL), shown in* Figure 8-1*, which is one of different actuator solutions provided by Schaeffler [46]*. The purpose of the *IPL system* is to lock the drive train and prevent vehicle movement during parking mode. To lock the drive train, a pawl is inserted which can hold the mechanical force. Locking/Unlocking requests as well as status responses are communicated via CAN bus between IPL and *external (vehicle) ECUs*.
To operate the system in the context of passenger cars, common challenges of the automotive sector must be met. This includes various requirements of functional safety, operating and lifetime, material, environmental effects, and strict cost constraints.

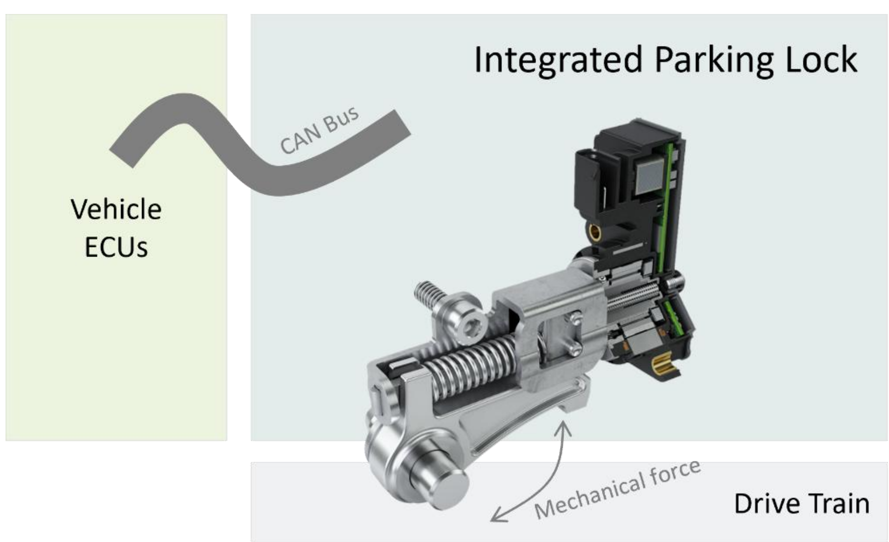


Figure 8-1: Representation of the Integrated Parking Lock and its context

8.1.3 Requirement View

Usually, the development of automotive systems comprises a high number of detailed requirements. To simplify the scope of the Schaeffler Case Study, requirements were reduced to a set of high-level requirements and design constraints. In this way, the case study builds on a simplified case that is as independent from existing technical solutions as possible. An excerpt of high-level requirements can be found in Figure 8-2.
The intended use of the *IPL* in passenger cars comes with several constraints. Certain guidelines and safety standards, such as ISO 26262 [47][46], must already be met during development. Also, the materials applied on the system must comply with current environmental regulations. The main function of the *IPL,* which is preventing the movement of the vehicle during parking, is defined by functional requirements. The functionality is further specified by requirements e.g., the transition time between parking mode and driving mode.
Although the idea of the case study is solution-neutral development, the requirements already define design constrains. Requirements from its context, e.g., other *ECUs* to communicate with, can restrict the solution space already in the early phase of development. Thus, the application of CAN Bus is already specified in the high-level requirements.

8.1.4 Functional View

The functional view (see Section 6.3) is used to specify the functionality of the IPL on a system level. The goal is to define the system functions of the SUD with inputs and outputs at the system boarder of the IPL and to decompose them into subfunctions to get an overview about the different necessary steps in the transformation from inputs to outputs.

#	Name	Text
1	⊟ D 1.0 Automotive Context	The system shall operate in the automotive context of passenger cars
2	P 1.1 Functional Safety	The system shall fulfill the functional safety standard ISO 26262
3	P 1.2 Gross Weight	The system shall be installed in road vehicle with a maximum gross weight of kg
4	P 1.3 Road Vehicle Environmental Conditions	The system shall fulfill the requirements of ISO 16750 Road vehicles - Environmental conditions
5	P 1.4 Ingress Protection Code	The system shall meet the following Ingress Protection Codes in accordance with ISO 20653:
6	⊟ R 2.0 Prevent Road Vehicle Movement	The system shall prevent the movement of road vehicle during parking
7	R 2.5 Manuel Control	The system shall provide a way to manually lock and unlock
8	D 2.6 Drive Train	The system shall lock the vehicle drive train
9	R 2.1 Parking Mode	The system shall lock in status parking mode
10	R 2.2 Vehicle Movement	The system shall not lock in status vehicle movement
11	R 2.3 Movement After Locking	The system shall allow a maximum vehicle movement of mm in status locked
12	R 2.4 Locking Speed	The system shall lock and unlock within second
13	⊟ R 3.0 Communication	The system shall communicate with other ECUs
14	R 3.1 Status Report	The system shall communicate status messages
15	R 3.2 Control Signals	The system shall communicate control signals
16	D 3.3 CAN Bus	The system shall communicate according to ISO 11898 Road vehicles - Controller area network (CAN)
17	P 4.0 Maximum Weight	The system shall have a max. weight of kg
18	D 5.0 Operating Time	The system shall operate during a vehicle lifetime of h
19	D 6.0 Life Time	The system shall be functional over a lifetime of years
20	D 7.0 Material	The system shall consider laws and regulations based on (EC) No. 1907/2006 (REACH)
21	P 8.0 Noises	The system shall not generate any noise above dB (A) according to DIN EN 21683

Figure 8-2: Requirements table (Screenshot of tool implementation)

8.1.4.1 Functional Context

Starting with the functional context, the function *prevent movement of road vehicle during parking* is identified as the main functionality of the *IPL* and will be further refined in the following sections. To understand the context, it is important to consider additional context functions, that are outside of the system boundaries. Five external functions are defined that need to interact with the main function. Three of them relate to the request of *drive mode*, *park mode* or *status report*. The remaining two are used to *lock* and *unlock* by the vehicle drive train (Figure 8-3).

8.1.4.2 Functional Black Box Model

The functional black box view shows all identified functions on system level. In total the *IPL* is composed of three system functions, which can be seen in Figure 8-4. Each system function handles its individual request. If the request triggers a state transition of the *IPL* the locking or unlocking process will be triggered. The system function then responds with the resulting system status.

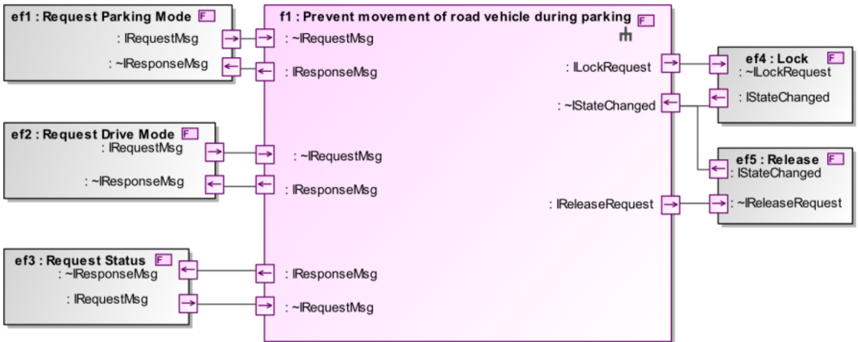


Figure 8-3: Functional context view of the IPL

8.1.4.3 Functional White Box Model of "Enter Parking Mode"

The white box model shows the breakdown of the system function *Enter Parking Mode* by decomposing it into white box functions shown in Figure 8-5. After receiving and processing the *parking mode* request, a command signal will be forwarded to the *Lock Drive Train* function. To prevent vehicle movement, the external *Lock* function is triggered. Based on the response of the *Lock* function, the *feedback* is sent as a *status report* to the initial caller function.

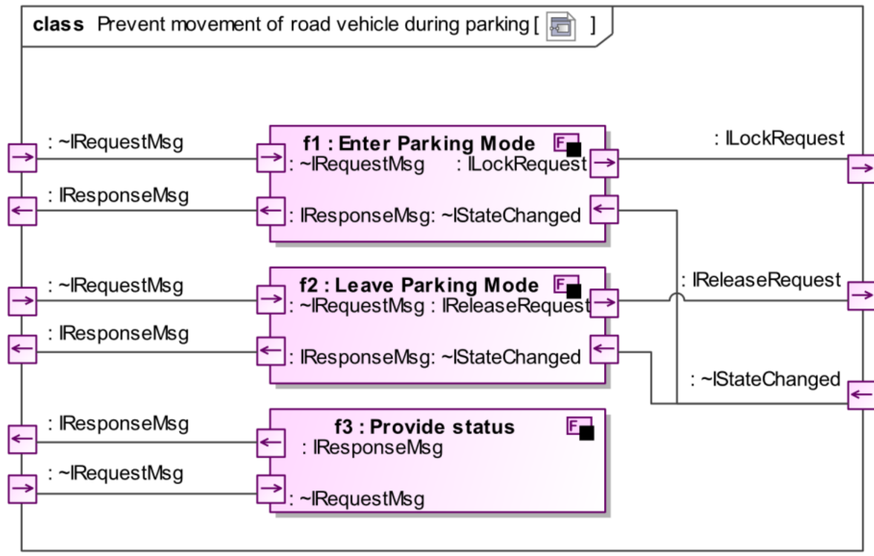


Figure 8-4: Functional Black Box View of the IPL

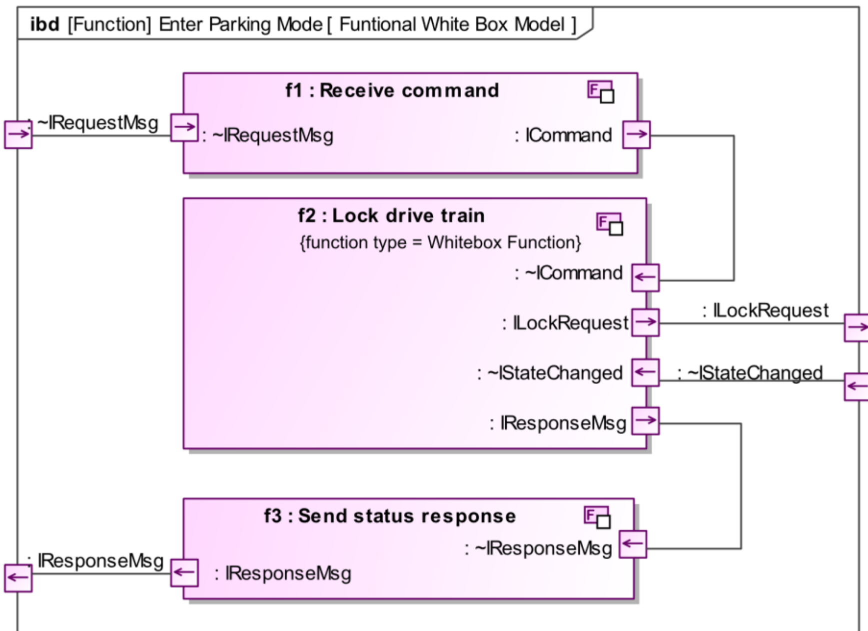


Figure 8-5: Functional white box view of the IPL

8.1.4.4 Mode Model

Because there are no dependencies between the IPL system functions, the Mode Model is not applicable.

8.1.4.5 Evaluation Result

The functional viewpoint enables the decomposition of the system main function to a set of system functions. By applying the concepts of the universal interface model (see Section 4.6), it allows to model the interfaces to functions outside of the system boundary. In this way a functional specification can be created without being influenced by any technical solutions. To add even more details, system functions can be specified by white box functions. This allows deeper understanding of the functional dependencies inside the system.

8.1.5 Logical View

The logical view describes a set of logical elements (logical components) and their interdependencies of the *IPL* to implement the specified behaviors of the functional view. Components which are not restricted by design constraints in the requirements table are still independent from their technical realization.

Although allocations between functions and logical elements can be *x:y*, in the practical usage it is recommended to do the functional decomposition in a way that enables the allocation of a function to exactly one logical element (x:1).

8.1.5.1 Logical Context

The logical context describes how the *SUD* interacts with external systems. The IPL in this example has two relevant actors: An external (vehicle-) control unit with outgoing and incoming communication and a drive train where we have a force transmission interface, shown in Figure 8-6.

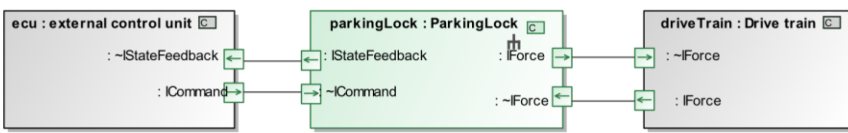


Figure 8-6: Logical context of the IPL

8.1.5.2 Logical Component Structure

The logical system architecture of the *IPL* comprises three logical components, shown in Figure 8-7. Inputs and outputs of the *Parking Lock* component are connected to an input or output of sub-components. Each component interacts with others to realize the specified system functions of the functional view.

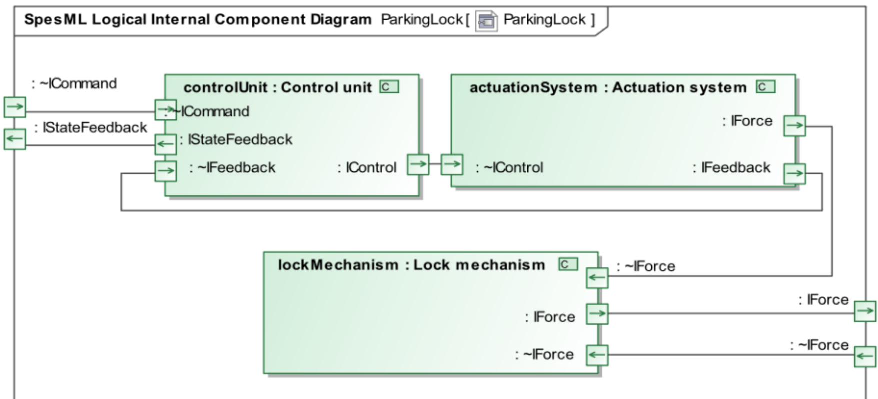


Figure 8-7: Architecture view of the IPL

The view visualizes the *Control Unit* with its external communication interfaces *Command* and *StateFeedback*, to receive and send messages and the Control interface to enable control of the *Actuation System*. If the *Control Unit* receives a new request, it will be converted to a control command and forwarded to the *Actuation System*. The *Actuation System* controls the *Lock*

Mechanism by emitting and retracting mechanical force. The Lock mechanism must create enough force to lock the drive train.
At this point, all logical components contain the required level of complexity and further refinements are not necessary.

8.1.5.3 Behavior

Behaviour can be modelled via SpesML State-Machines (see Section 5.5) linked to its associated logical components. Figure 8-8 shows the behaviour modelling of the *Actuation System* which was already defined as a logical component in Figure 8-7.

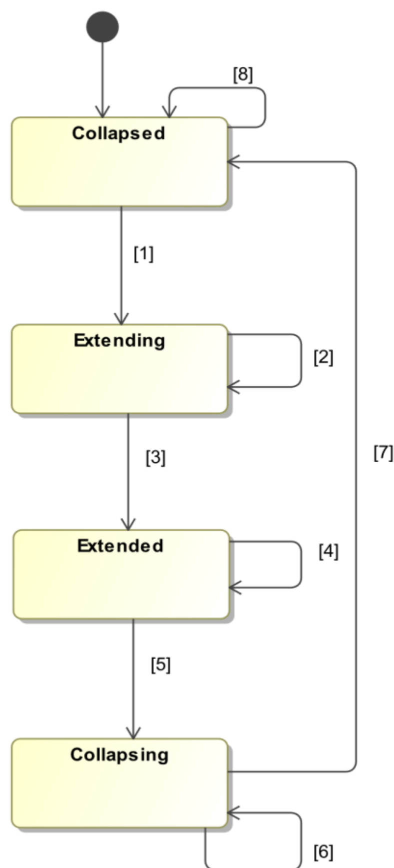


Figure 8-8: Behavior view of the Actuation System (transitions, see Table 8-1)

To specify transitions between states, SpesML provides its own syntax. The diagram offers two different kinds of guard conditions. On the one hand, the transition is triggered by an input received on a specified channel. For example, the transition between *Collapsed* and *Extending* is triggered by *D_STATE* input on the control channel. On the other hand, changes on internal

states will trigger transitions i.e., between *Extending* and *Extended*. The guard condition refers to two value-properties, which are compared. If the *actuation position (act_pos)* is greater or equal than the actuator *maximum position (act_max)* the system state will change to *Extended*. A similar approach applies to events on SpesML transition. The syntax can be used to emit an output on port channels as well as performing status changes. For example, two events on the transition between *Extending* and *Extended* includes the value of emitted force on channel force and returning an *OK* response on the feedback channel.

Table 8-1. State transitions

[1]	[Control.control == CONTROL.D_STATE] / {Transmission.force = (act_pos * d_spring);act_pos += 1; Feedback.feedback = CONFIRMATION.WAITING;}
[2]	[act_pos < act_max] / {Transmission.force = (act_pos * d_spring); act_pos += 1;Feedback.feedback = CONFIRMATION.WAITING;}
[3]	[act_pos >= act_max] / {Transmission.force = (act_pos * d_spring); Feedback.feedback = CONFIRMATION.OK; }
[4]	[Control.control != CONTROL.P_STATE] / { Transmission.force = (act_pos * d_spring); Feedback.feedback = CONFIRMATION.WAITING; }
[5]	[Control.control == CONTROL.P_STATE] / {Transmission.force = (act_pos * d_spring); act_pos -= 1; Feedback.feedback = CONFIRMATION.WAITING; }
[6]	[act_pos > 0] / {Transmission.force = (act_pos * d_spring); act_pos -= 1; Feedback.feedback = CONFIRMATION.WAITING; }
[7]	[act_pos <= 0] / {Transmission.force = (act_pos * d_spring); Feedback.feedback = CONFIRMATION.OK; }
[8]	Control.control != CONTROL.D_STATE

SpesML provides a way to run simulations using state machines to verify the modelled behavior by various test cases. The test case, visualized in Figure 8-9, contains the *Actuation System* isolated from all other logical elements of the *IPL*. The *controller* for *test case 1* sends a predefined sequence of inputs signals to the *Actuation System* while the assert for *test case 1*

compares its output with the expected results. To ensure the intended behavior of a system, multiple test cases are needed.

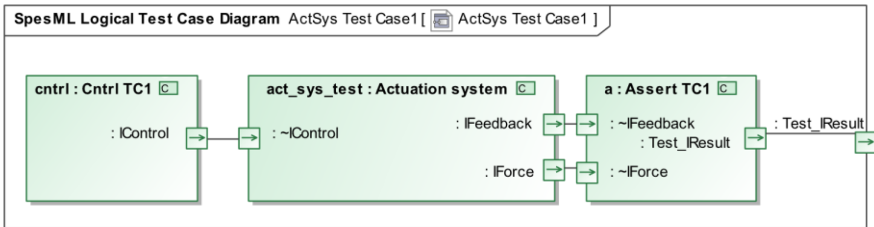


Figure 8-9: Test case to validate the behavior of the Actuation System

8.1.5.4 Evaluation result

The logical view enables the creation of the system architecture without thinking about technical details and restrictions. Logical components can be used no matter how complex the component is. If complexity of the component should be reduced, this can easily be done by decomposition into logical subcomponents. This allows a flexible design of the logical system architecture based on functional view.
When the desired level of detail has been achieved, the model can be extended by behavior models. SpesML provides its own syntax to specify transitions in state machines.
To support readability of model diagrams, complex or messy syntax can be outsourced to executable functions (see Section 5.4). Through to well-formless rules (see Section 6.4), the user is directly alerted regarding errors in the specification, which is a significant relief in the process. Overall, this approach provides a widely known way of behavioral definition such as peculiarities.
Additionally, SpesML offers the Simulation Plug-In to run and test the state machines. This allows the user to get (almost) instant feedback on developed behavior models and helps to avoid errors, even in later changes. At the point in time when the evaluation took place, the visualization possibilities for the simulation results were very limited. The results needed to be analyzed in the generated log files, which takes a lot of time and has a high error potential. This is a possibility for future optimizations.
The logical architecture has the potential to serve as a good and stable base for following technical decisions and the development of different technical variants.

8.1.6 Technical View

The goal of the technical view is, to transform the platform independent models of the logical view to platform specific models. All logical components will be specified by its technical implementation.

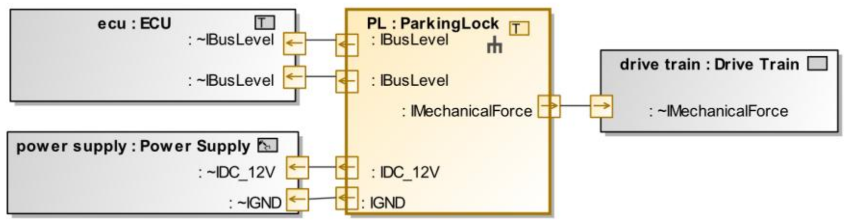


Figure 8-10: Technical context view of the IPL

8.1.6.1 Context

The technical context defines the environment and interfaces the IPL will be integrated. Compared to the logical view the interfaces are more detailed and technical aspects must be considered, that depend on the selected technical components. While the logical communication interfaces *Command* and *StateFeedback* are quite abstract, Figure 8-11shows the technical implementation via *CAN bus*, with its two channels *CAN_H* and *CAN_L*.
In addition, a *Power Supply* has been added to the context view. Through the fact of a 12 V power source, the technical solution is restricted, which was not necessary on the platform-independent solution provided by the logical layer.

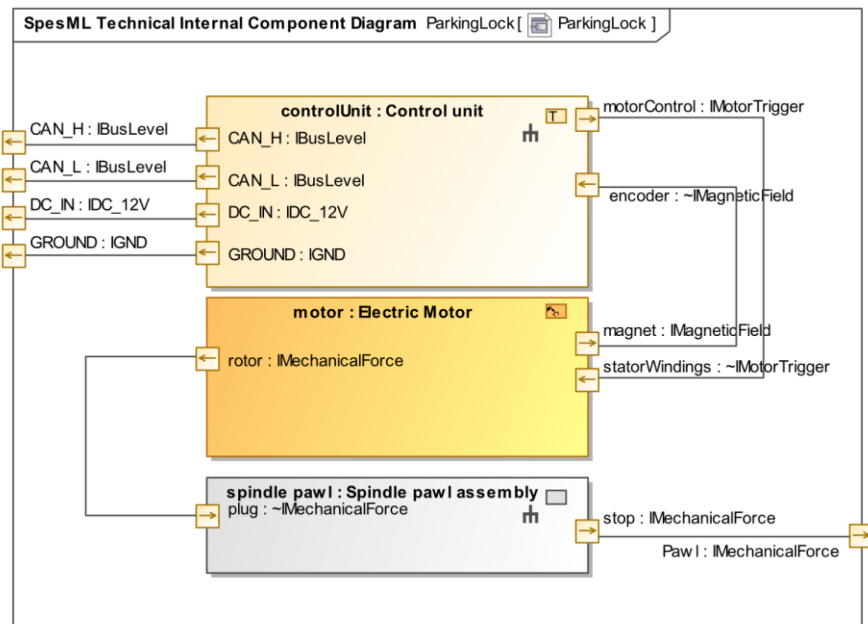


Figure 8-11: Technical architecture view of the IPL

8.1.6.2 Structure

The architecture of the technical realization of the *IPL* can be seen in Figure 8-11.The *Actuation System* (logical component) is realized by an *Electric Motor* (technical component). The *Lock Mechanism* (logical component) is realized by *Spindle Assembly* and *Mechanical Pawl*.
If the technical component can already be assigned to a particular discipline such as mechanic, electronic or software, a specific stereotype can be applied. If not, the generic technical component stereotype is to be used and the decomposition into more specific components is needed.

8.1.6.3 Evaluation Results

After defining a platform independent model within the logical view, the focus of the technical view is to describe the technical implementation including the software execution subsystems. Technical components can be stereotyped as discipline-specific components. However, they can also remain generic and be further detailed in the later process.
The ability to break down the system structure in any level of detail gives the user many options for modelling without being restrictive. Elements that have already assigned to a particular discipline can be further detailed in a domain specific description. While mechatronic engineers can detail in terms of electronic components, software engineers can model *Software Tasks* and integrate it later.

8.1.7 Crosscutting Concepts

8.1.7.1 Traceability

To establish traceability in the system model, traces are used in two different ways. First to link system elements between the functional, logical, and technical views and second to link elements from each view to requirements.
The matrix, shown in Figure 8-12, contains elements from the functional view that are traced to elements from the logical view. Each trace means that the linked function will be realized by its logical component.
The same principle is also applied in the transition from logical view to the technical view. The generated matrix view contains all relevant elements and traces can be created that allocate logical components to technical realizations.
After filling in all traces into the matrices, the tool allows to generate dependency graphs. By selecting an element, the traces across the different views are visualized. Figure 8-13 shows the traces starting from the *CAN transceiver* of the technical view. The *CAN Transceiver* realizes two logical components: *Msg Emitter* and *Msg Receiver*, which in turn implement the functions *Send status response* and *Receive command*.

Legend: R realizes function

3.3 Elements [Model::SpesML:: Logical Viewp	2.3 Elements [Model::SpesML::Functional Viewpoint	2.3.1 System under	Prevent movement o	2.3.2 System Functic	Enter Parking Mode	Leave Parking Mode	Provide status	2.3.3 Whitebox Func	Determine status	Lock drive train	Receive command	Send status response	Unlock drive train	2.3.5 External	Lock	Release	Request Drive Mode	Request Parking Moc	Request Status
3.3 Elements [Model::SpesML:: Logical Viewp			1		3	3	2		1	2	1	1	2		1	1	1	1	1
3.3.1 System under Development			1																
ParkingLock	1	1	R																
3.3.2 Components					3	3	2												
Actuation system	3			3	R	R	R												
Control unit	3			3	R	R	R												
Control Unit Composed																			
Lock mechanism	2			2	R	R													
3.3.3 Subcomponents									1	2	1	1	2						
Actuation controller	2							2		R			R						
Msg emitter	1							1				R							
Msg receiver	1							1			R								
Position Determinant	1							1	R										
Processing Unit	2							2		R			R						
3.3.4 External															1	1	1	1	1
Drive train	2													2	R	R			
ECU	3													3			R	R	R

Figure 8-12: Matrix view to display and create traces between logical and functional view

Besides traces across views there are also traces between elements from any view to requirements. The generated matrix provides the set of requirements and all elements of a particular view. E.g., Figure 8-14 shows the tracing of elements from the logical view to requirements.

Applied to all views, the dependency graph shown in Figure 8-13 is extended by various traces to requirements. The result is shown in Figure 8-15.

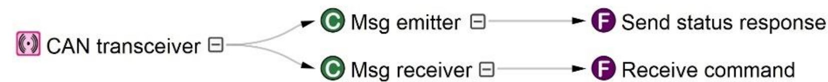


Figure 8-13: Dependency graph to visualizes traces across viewpoints

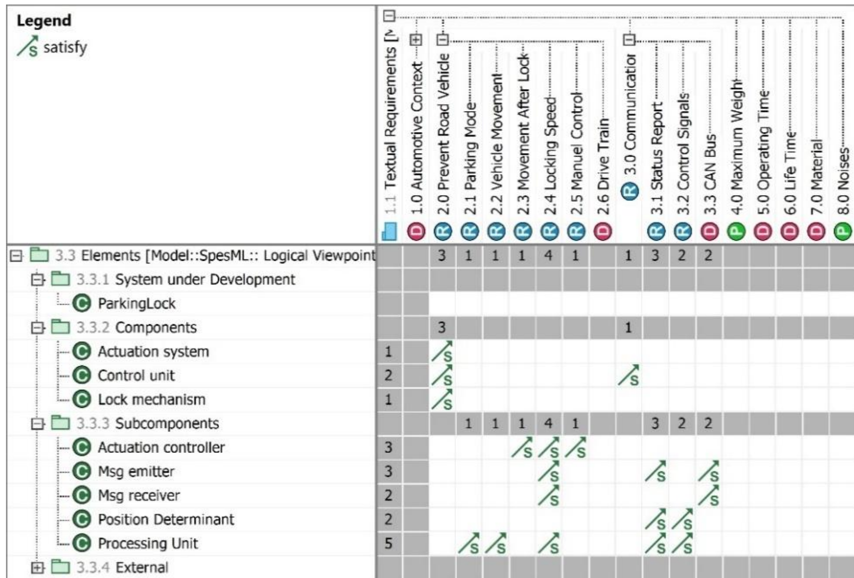


Figure 8-14: Matrix to create and display traces between logical viewpoint and requirements

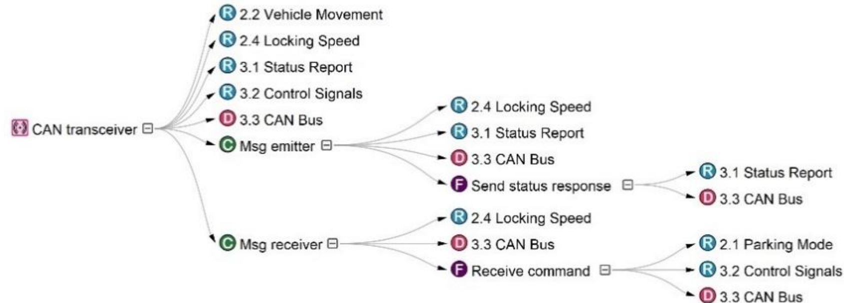


Figure 8-15: Dependency graph including traces to requirements, generated by the SpesML Impact Analyses

8.1.7.2 Impact Analysis

In the automotive industry, changes in requirements or environmental conditions are quite common. Integration of changes into an existing system model is time-consuming and error prone. SpesML Impact-Analysis offers a way to get a quick first insight, on how the change affects the whole model and its elements.

Selecting one or multiple requirements a graph is generated including all related elements that could be affected by the initial change. The listed elements are starting in the functional view, over the logical view up to the technical view. This way the effort of changes can be understood and estimated quickly.

Figure 8-16: Dependency graph generated by the SpesML Impact Analyses

To illustrate the Impact Analysis, the communication via *CAN bus* is to be changed as an example. To generate the dependency graph, the affected requirement *“3.3 CAN BUS - The system shall communicate according to ISO 11898 Road vehicles - Controller area network (CAN)”* [48] is selected and graph from Figure 8-16 is displayed.
By changing this requirement, the functions *Receive command* and *Send status response* need to be examined. These functions are originally realized by the logical components *Msg receiver* and *Msg emitter* which also must be examined regarding necessary changes. The last element in the chain that must be observed for potential changes, is the *CAN Transceiver*.

8.1.8 Sum-up and Overall Evaluation

The increase in product complexity requires new, innovative development methods and tools that support the developers as good as possible and reduce manual efforts as well as failure potentials. There is no doubt that the future of development will be more and more defined by model-based and formalized, machine-readable information. The challenge is the combination of method, language, and tool solutions while realizing an efficient usability for developers. SpesML has proven that this is possible, and already provides an initial set of a well-defined model-based development methodology, including a language profile and tool plugins that improve the tool usability and increases productivity of engineer productivity.
The basic Viewpoint structure fits very well with the common development structures in the systems engineering context. Addressed concerns by the models of the SPES Viewpoints cover many aspects which must be considered in the automotive development context.
As SpesML is based on UML/SysML, it already implements many best practices e.g., regarding traceability, which are already widely used in the industry. Developers who are experienced with model-based engineering can start quickly without intensive trainings on the base SysML language elements. Through to well-formedness rules, the SpesML Plugin detects many syntax errors and provides feedback to the user in real time. It not only prevents mistakes and frustration in modeling, but also makes it easier for new developers to get started.
Another method to gather feedback on system models is using the simulation plugin provided by SpesML. It opens the possibility of running model-based test cases even at the early stages of development. Especially in implementing later changes in the system model there is a traceable assurance for system safety.

8.2 Simplified Radiography System (Siemens Healthineers)

8.2.1 Domain context

Siemens Healthineers AG (listed in Frankfurt, Germany: SHL) pioneers breakthroughs in healthcare. For everyone. Everywhere. As a leading medical technology company headquartered in Erlangen, Germany, Siemens Healthineers and its regional companies are continuously developing their product and service portfolio, with AI-supported applications and digital offerings that play an increasingly important role in the next generation of medical technology. These new applications will enhance the company's foundation in in-vitro diagnostics, image-guided therapy, in-vivo diagnostics, and innovative cancer care. Siemens Healthineers also provides a range of services and solutions to enhance healthcare providers' ability to provide high-quality, efficient care. Further information is available at www.siemens-healthineers.com.

8.2.2 Case Study Introduction

A large part of the Siemens Healthineers portfolio is dedicated to X-ray based medical imaging systems. Within this portfolio segment, radiography is still the most used imaging technique worldwide. Every clinical institution has to offer it. Therefore, a radiography system was chosen as the Siemens Healthineers industry example. As such a system has a huge complexity and could not be modeled in detail within the scope of the project, a twofold approach was followed: On the one hand, the focus of modeling was set on covering all relevant aspects of a radiography system on a higher level. For the sake of demonstration, this system has been reduced to the bare functional minimum, the so-called *simplified radiography system*. On the other hand, a selected component, the X-ray collimator, was modeled with all its relevant aspects in detail. This approach is depicted in Figure 8-17. In a real-world development project setup, the responsibility for the radiography system model and the collimator model would be split between dedicated cross-functional teams. Such a model collaboration requires a proper separation of modeling scopes. Therefore, two models have been used. The model of the simplified radiography system integrates the artifacts of the collimator model by employing the "project usage" capability of MagicDraw. The resulting model hierarchy, as well as the resulting containment tree structure, is shown in Figure 8-18.

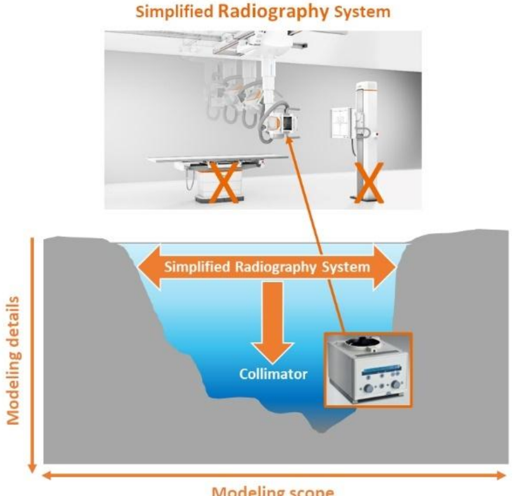


Figure 8-17: Modeling scope for the simplified radiography system and the collimator

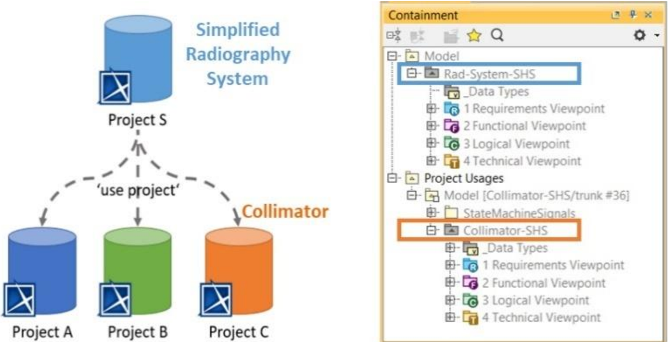


Figure 8-18: Model structure (left) and corresponding containment tree structure (right)

As stated above, the responsibility of the individual models used in the project will be within dedicated teams, or even outside of the company in case of models provided by a third-party supplier.

8.2.3 Requirement View

The requirement view allows collecting all stakeholder needs and constraints to the cyber-physical system to be developed. They describe any boundary conditions to be met to

successfully realize the system of interest, such as normative requirements (e.g., in the case of medical devices the IEC 60601).
The requirement view was derived on the collimator only, as it was the major modeling focus. The approach for the simplified radiography system would be identical and would not have provided any additional learnings for the application of the SpesML plugin.
For better clarity, the requirements for the collimator are grouped into topic-specific folders, as shown in Figure 8-19, with a detailed view on the collimator performance requirements. Using the requirement classifications implemented in the SpesML plugin, one can easily identify the different types of requirements.
Collimator leakage radiation (item #30) is an example for a quality requirement being demanded by a normative standard. Beside quality requirements, the complete table provides examples for design constraints, interface, and functional requirements.

#	Name	Text	Requirement Type	Documentation
1	Requirements			
2	Regulatory			
11	User Experience			
22	Collimator Performance			
23	D Collimation lamella thickness	Lamella thickness (Pb-Gleichwert) = 3+0.2 mm	Design Constraint	
24	D Collimation lamella material	Lamella material lead purity >= 97%	Design Constraint	
25	Q Collimator Inherent filtration	Inherent filtration >= 1.00 mm Al / 75 kV IEC 60522:2003	Quality	engineered category of IOS25010: performance efficiency Expected test results see requirement
26	Q Collimation repetition accuracy	Repetition accuracy <= +/- 1 mm)	Quality	engineered category of IOS25010: performance efficiency Expected test results see requirement
27	Q Collimation maximum Opening	Maximum Opening >= 450 x 450 mm	Quality	engineered category of IOS25010: performance efficiency Expected test results see requirement
28	Q Collimation minimum Opening	Minimum Opening <= 30 x 30 mm	Quality	engineered category of IOS25010: performance efficiency Expected test results see requirement
29	Q Collimation max. Positioning time	Max. Positioning time <= 1 s	Quality	engineered category of IOS25010: performance efficiency Expected test results see requirement
30	Q Collimator Leakage Radiation	Leakage Radiation (CFR21 & IEC 60601-1-3) < 500 μGy/h @150 kV	Quality	Test setup is defined in CFR21
31	Hardening Performance			
41	Interface			
47	Integration			
61	Reliability			
65	Transport			

Figure 8-19: Performance requirements for the collimator ("Collimator Performance").

8.2.4 Functional View

The functional view is used to specify the functionality of the simplified radiography system and the collimator on their individual system levels. The goal is to define the system function(s) of the system under development with inputs and outputs at the system border and to decompose

them into subfunctions to get an overview of the different necessary steps in the transformation from inputs to outputs.

8.2.4.1 Functional Context

The functional context is used to identify all context functions and stakeholders, which interface to the simplified radiography system, as well as to the collimator. As the collimator is part of the radiography system, the functional context of the collimator is a subset of the simplified radiography system functions and its context.

Functional Context of the Simplified Radiography System

A radiography system is usually installed in the radiology department of a hospital. In our example, the main function is to acquire a medical image and store it in the *PACS* (Picture Archiving and Communication System). To do so, the main system function needs information about the patient being provided by the *RIS* (Radiology Information System) as well as input from the *MTA* (Medical Technical Assistant). There is also a functional interaction with the patient whose body is being imaged with the help of X-rays. All these functional interactions of the context with the simplified radiography system are shown in Figure 8-20.

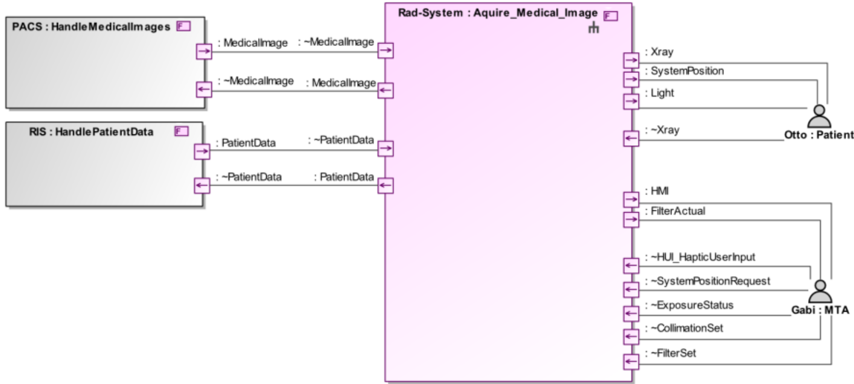


Figure 8-20: Functional context of the simplified radiography system

Functional Context Collimator

In the first step, the interaction of the collimator function with external functions of the integrating system was mapped in the functional context (see Figure 8-21). This defines with which stakeholders and functions of the radiography system the collimator will interact with and how these will use the collimator functionality.

The main purpose of the collimator is to reduce the X-ray exposure of the patient by limiting the X-ray field of view to a minimum. Physically this is done by partially blocking the X-ray beam with lead lamellas. Therefore, as shown in Figure 8-21, the function *CollimatorFunction*

has an X-ray input, which interfaces to the external function *Generate X-ray*, providing the X-ray beam, and an X-ray output, interfacing directly to the patient.
The collimator can be controlled remotely by the *RAD System Control* function using *its CollimationSet* input. The collimator signals back to the System Control function the actual collimation position using the interface *CollimationActual*.
Furthermore, the collimator can be controlled locally by the MTA using the *CollimationSet* input, which allows adjusting the field of view manually by the support of a light field being provided by the collimator, which indicates the X-ray field of view.
The collimator further allows hardening the X-ray beam using different metal filters. The correct filter can be set either by the Rad System Controller via the *FilterSet* interface or locally by the MTA using the second *FilterSet* interface. In both cases, the collimator signals back the filter status via the *FilterActual interfaces.*

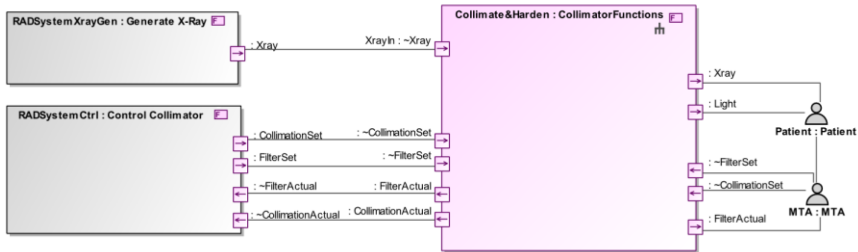


Figure 8-21: Functional context of the collimator

8.2.4.2 Functional Black-Box Model

The functional black-box model shows all identified functions on the system level of the simplified radiography system as well as of the collimator.

Functional Black Box Model of the Radiography System

The functional black box view shows all identified functions on the system level. In total, the main function *Acquire_Medical_Image* consists of five system functions, as shown in Figure 8-22. The system must be positioned, configured, and controlled. Additionally, the X-rays must be generated and collimated. After passing through and being attenuated by the patient, the image relevant X-ray photons are converted to a medical image.

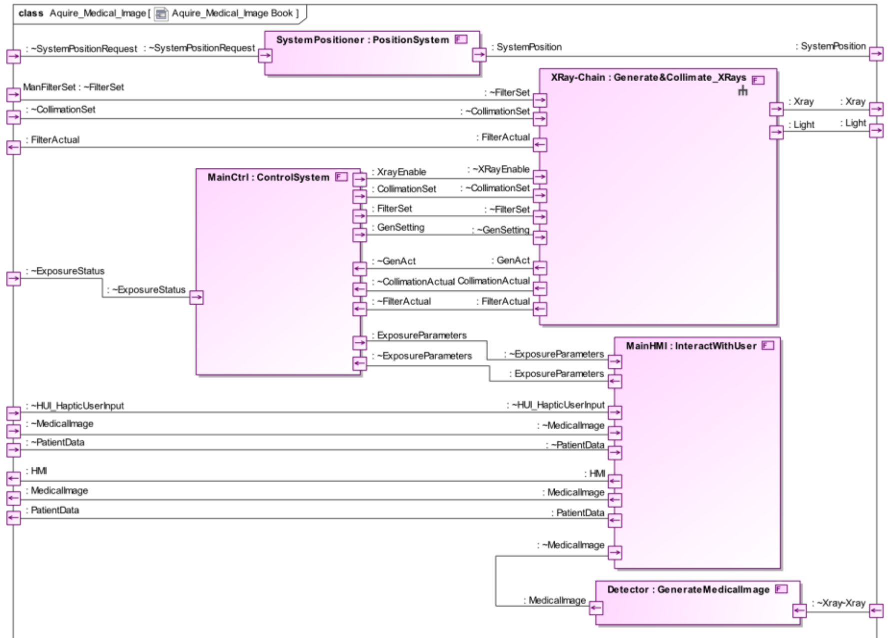


Figure 8-22: Black Box functional view

Functional Black Box Collimator

The collimator provides two main functions, as shown in Figure 8-23. First, the incoming X-rays will be hardened by the *Harden X-Ray Beam* function before the beam will be shaped to its final form by the *Collimate X-Ray Beam* function. All functional inputs and outputs of the collimator shown in Figure 8-21 could be connected to the according input and outputs of these two functions. As both functions can be sufficiently specified on this level, there is no need to further decompose these functions using the functional white box model.

8.2.4.3 Functional White Box Models of the Simplified Radiography System

The white box model shows how to realize the behavior of a system function by decomposing it into multiple white box functions. Figure 8-24 gives a detailed view on the system function *Generate&Collimate_XRays*. It consists of the two functions *GenerateMedicalXRayPhotons* and the *CollimatorFunction*.

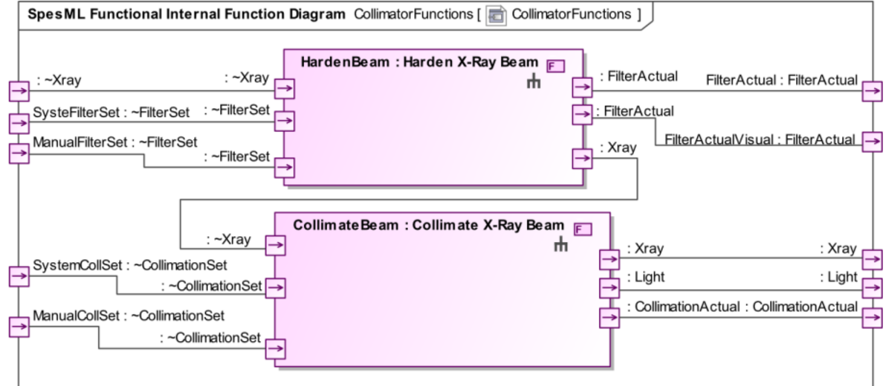


Figure 8-23: Functional black box model of the collimator

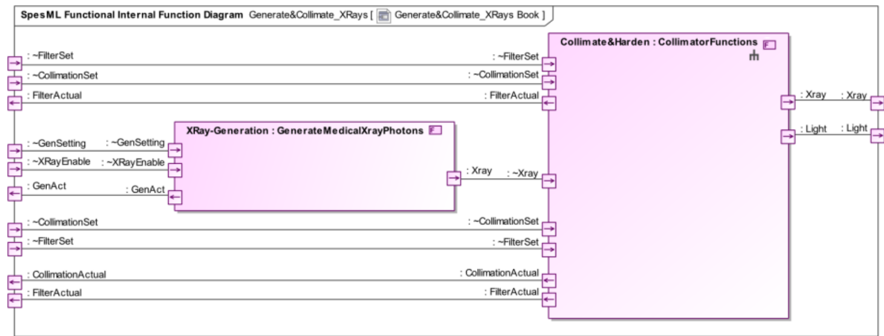


Figure 8-24: White box model of the system function "Generate&Collimate_XRays"

8.2.4.4 Mode Model

The behavior of the simplified radiography system and its components can be described by state machines. A central control unit hosting the main radiography system state machine triggers changes in the component state machines by sending corresponding messages to the individual components. Additionally, messages sent by the components to the central control unit trigger changes in the main system state machine. In this setup, the concept of Modes has not been needed to model the behavior of the system and its components.

8.2.4.5 Remarks

The functional view describes the main functionality of the system of interest as perceived from the outside and how these functions interact with elements in the functional context. While the SpesML plugin has the possibility to model state machines on this functional level, we believe that extending the modeling capabilities by sequence diagrams could be beneficial to model

different dynamic interaction scenarios of the system functions with their context. Following the concepts of the universal interface model, a state machine allows to precisely model and verify the system behavior, the use of sequence diagrams would further allow modeling integration scenarios in order to provide input for the validation of the system behavior in its integration context.

8.2.5 Logical View

The logical view describes the structuring of the system under development to implement the behavior being specified in the functional view.

8.2.5.1 Logical Context

The logical context describes how the system under development interacts with external logical elements.

Logical Context of the simplified radiography system

The logical context shown in Figure 8-25 describes how the simplified radiography system interacts with external logical systems. External IT systems provide information about the patient and details about the image to be acquired. These IT systems also store medical X-ray images for archiving and documentation purposes. Additionally, the medical technical assistant needs to interact with the system to set up and control it [10].

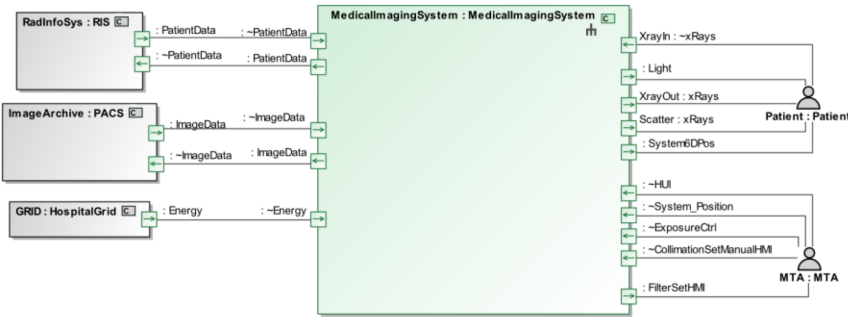


Figure 8-25: Logical context of the simplified radiography system

[10] To function properly, the simplified radiography system needs to be provided with electrical energy, being realized by a connection to the hospital grid.

Logical Context Collimator

Following the approach of the functional view, a collimator context diagram was first modeled in the logical view (see Figure 8-26).

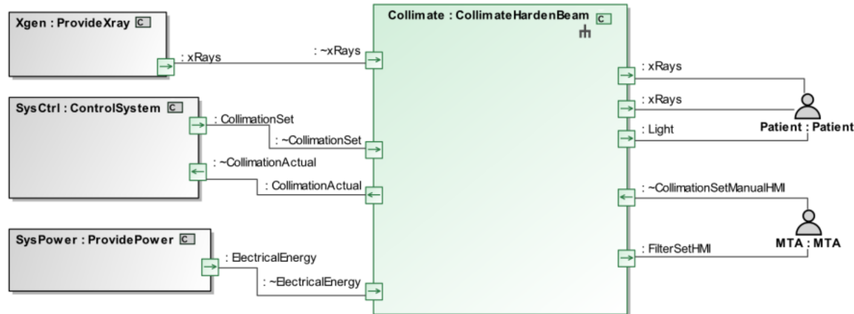


Figure 8-26: Logical context of the collimator

The diagram shows the logical *ProvideXray* and *ControlSystem* components. Further, the diagram shows a *ProvidePower* logical component, which represents an enabling logical component, typically being not explicitly requested by the user, but which is essential technology wise.

8.2.5.2 Logical Architecture

The logical architecture shows the internal logical components of the system under development and how these components interact with each other to realize the system's functions.

Logical architecture of the simplified radiography system

The logical architecture of the simplified radiography system is shown in Figure 8-27. The complete system is controlled by the s*ystem-controller,* which also gets inputs from the user via the *MainHMI*. It is also controlling the X-ray chain which is responsible for generating the medical X-rays.

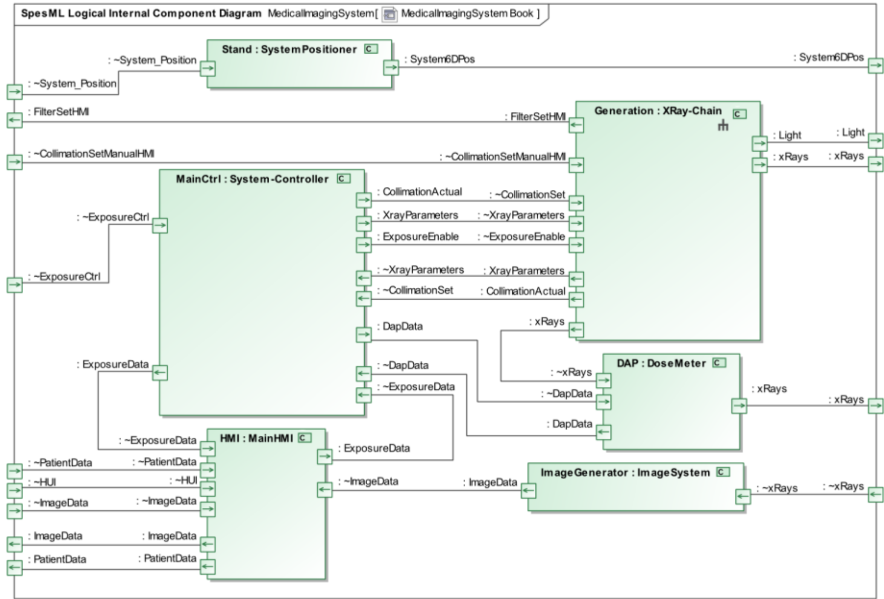


Figure 8-27: View of the logical architecture of the simplified radiography system

The collimator is part of the X-ray chain; therefore, this logical component was decomposed further. On this level of detail also the components responsible for *DriveXrayGeneration* and *GenerateXrays* show up, the latter one directly interacting with the collimator (see Figure 8-28).

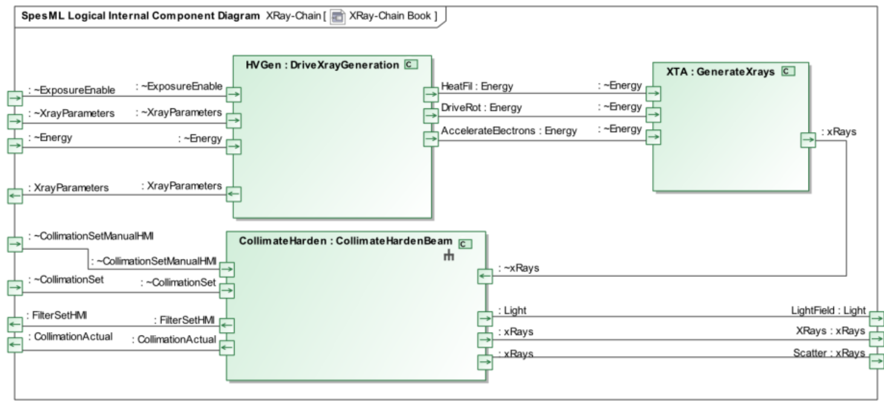


Figure 8-28: Detailed view of the logical component "XRay-Chain"

Logical Architecture of the Collimator

Figure 8-29 shows the top-level logical structure of the collimator. The behavior of the collimator is modeled by the logical component *CollimatorSystem_Com*. This logical component is connected to the logical component *System-Controller* from the Radiography-System and by this enables sending and receiving control messages. Furthermore, it models the

logical component *Interface2User*, which realizes the local user interface. The main functions of collimating and hardening the X-ray beam are modeled by four logical beam modification components, *HardenBeam*, *LightModule* and two instances of *CollimateBeam*, one for the collimation in x-direction and one for the y-direction. Further, the logical component *ShieldScatter* represents a safety function, which allows to shield the scattered radiation being generated by the beam modification components. For showcasing the energy distribution, a stakeholder specific diagram (shown in Figure 8-30) was created, which only focuses on this cross-cutting logical interaction. It is worth to note that even if an information is not shown in a diagram, it is still available in the model. By this, diagrams typically only show stakeholder specific views, whereas the complete model is within the containment tree of MagicDraw.

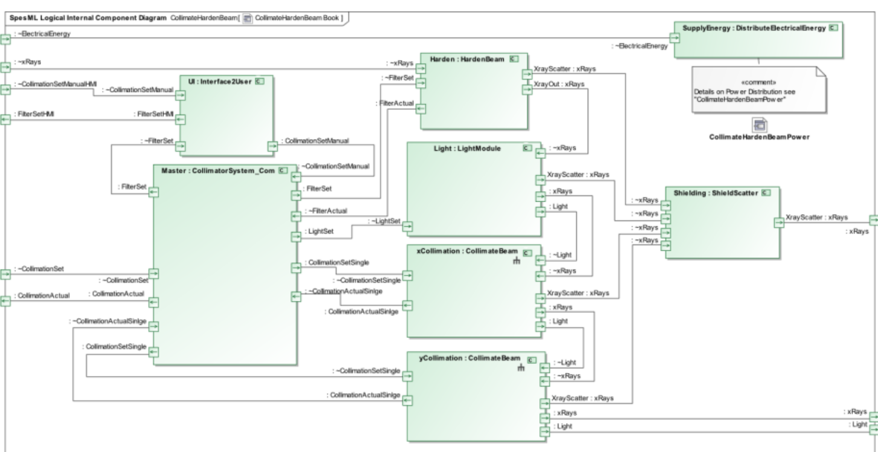


Figure 8-29: View of the logical architecture of the collimator

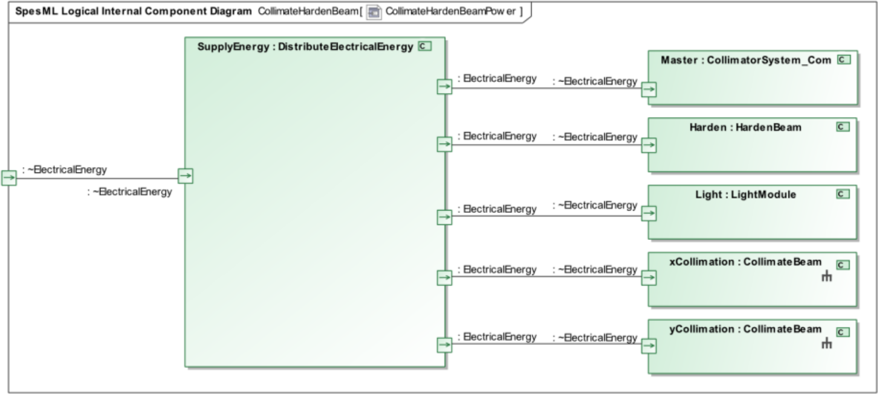


Figure 8-30: Stakeholder specific view of the logical power distribution of the collimator

The task of the system architecture is to decompose a system into smaller elements. This holds true until a level is reached where the elements could be uniquely assigned to the development disciplines electronics, mechanics, and software.

The logical component *CollimateBeam* was decomposed further, in order to showcase a seamless handover to the development disciplines. This logical component needs to be instantiated twice, one time for x- and one time for y-direction. The detailed structure of the type of this logical component is shown in Figure 8-31. On a top level this logical component must model the electronic, mechanic and software aspects of a single collimation axis.

The logical component *ControlCollimationAxis* realizes the central control function for this single axis. It is worth to mention, that each collimator function (hardening, collimation, light) works independently from each other. As it will be further shown in section 8.2.5.3, this independency demands independent state machines for the according logical components. In our case, this approach allows to set independently the filter, turn on the light or move any of the collimation axis in parallel. In this sense the central *CollimatorSystem_Com* realizes only the system and user interaction described by its state machine. Any command being received is forwarded for example to the component *ControlCollimationAxis*, where it is processed. Thus, the *ControlCollimationAxis* must take care for the control software to control a stepper motor, the electrical driver for the motor, the stepper motor itself and the mechanical movement unit with the lead blades.

The logical component *SystemCoordTranslation* takes care for the coordinate translation between the system coordinates and the local stepper motor coordinate system. In short, it translates metric coordinates into a step-based coordinate system.

The *ControlBladePosition* controls in real time the positioning of the system. It takes care of the acceleration and deceleration of the lead lamellas, as well as for the constant movement and steady state of them. Acceleration, deceleration, constant movement, and steady state are different states which again demand an independent state machine for this logical component.

The logical component *AbstractMotorCtrlHW* takes care of the mapping of software parameters, typically being stored in registers, to real physical parameters. This means, that such logical components typically show a mixture of software (registers) and hardware (electrical signals) inputs and outputs.

The outputs of the *AbstractMotorCtrlHW* are the inputs to the logical component *DriveMotor*, which provides the required electrical pattern to turn the electrical motor clock- or counterclockwise. The logical component *MoveBlades* transforms the electrical energy into a mechanical momentum, which finally leads to a defined position of the lead lamellas, absorbing the X-rays and the light.

This detailed decomposition will allow a one-by-one tracing of logical components to domain specific technical elements (electronics, software, mechanics), as is shown in section 8.2.7.1 .

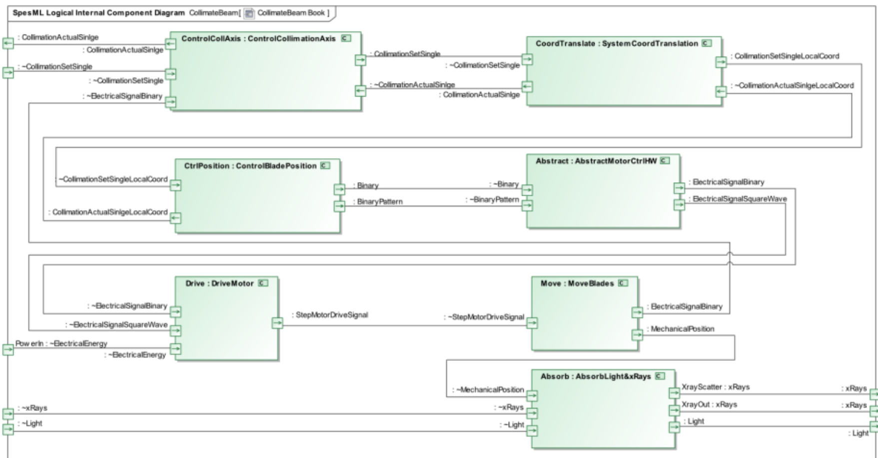


Figure 8-31: Detailed logical architecture of one collimation axis

8.2.5.3 Behavior

The behavior of the logical components is modeled via SpesML state machines.

Behavior of the simplified radiography system

The behavior of the simplified radiography system is controlled by the component *System-Controller* and modeled by the state machine shown in Figure 8-32. When the system is switched on, it enters an *Initialization* state where all elements of the system are configured and are performing self-tests. If no errors are reported over the corresponding interfaces, the system enters the *Stand By* state. In this state the user can either enter a service state (used for maintenance and repair of the system) or enter the *Setup Acquisition and process result* state by initiating the corresponding message exchange over the interfaces. Here all necessary parameters for acquiring a medical image are set. The imaging process itself happens in the *Acquire Image* state. Additionally, there is a *Shutdown* state used to power down the system in a controlled fashion. During any of the operational states' errors can occur. Receiving such an event will change the system state machine into the *Error* state. Leaving this state results in a complete re-initialization of the system.

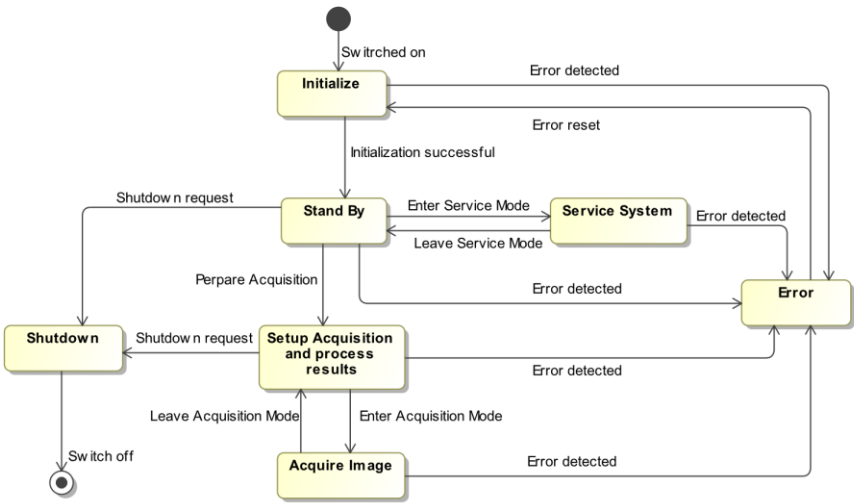


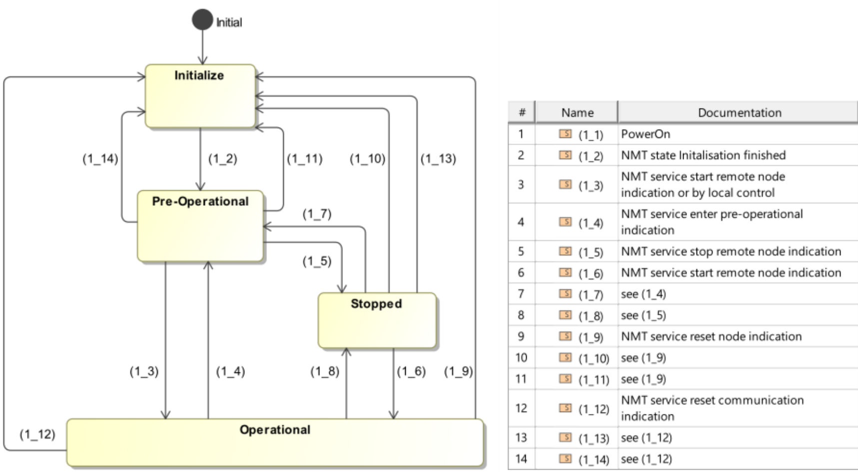
Figure 8-32: Central system state machine of the simplified radiography system

Behavior Collimator

The behavior of the collimator is described by the state machine of the component *CollimatorSystem_Com.* Additionally, the behavior of the collimator components responsible for hardening the beam, light generation and collimation axis control are described by individual state machines.

Figure 8-33 shows the state machine of the logical component *CollimatorSystem_Com* (see Figure 8-29), which models the collimator main initialization, as well as the communication to the system and the coordination of the internal state machines. The state machine and its state transition guards follow the Can Open specification CiA301 [49] ("CANopen application layer and communication profile") and thus ensures standardized-behavior of the collimator. After powering up, the collimator enters the state *Initialize*, in this state the *CollimatorSystem_Com* component is exchanging messages with the collimator internal components to check if all of them are in the operational state. If all components signal back that they are in their idle states, the collimator state changes to *Pre-Operational.*

By receiving the corresponding message from the main system controller component, the state *Pre-Operational* is changed to *Operational*, in which the collimator will accept standardized control commands.

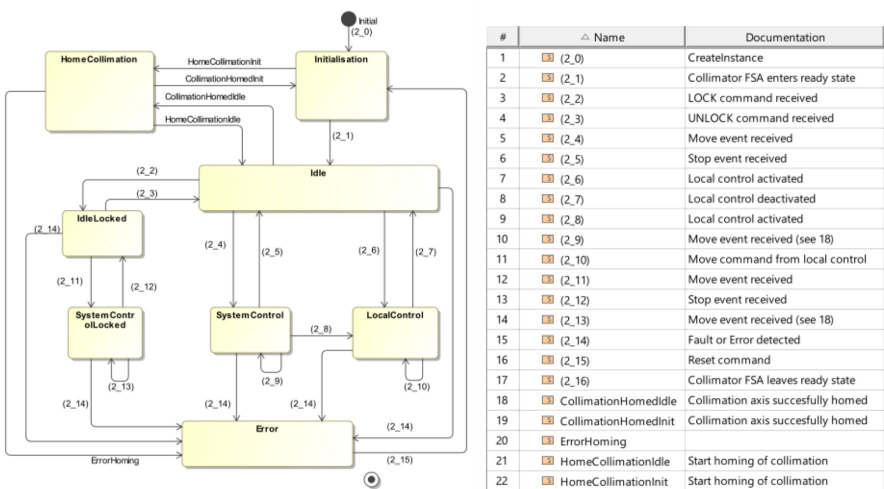


#	Name	Documentation
1	(1_1)	PowerOn
2	(1_2)	NMT state Initalisation finished
3	(1_3)	NMT service start remote node indication or by local control
4	(1_4)	NMT service enter pre-operational indication
5	(1_5)	NMT service stop remote node indication
6	(1_6)	NMT service start remote node indication
7	(1_7)	see (1_4)
8	(1_8)	see (1_5)
9	(1_9)	NMT service reset node indication
10	(1_10)	see (1_9)
11	(1_11)	see (1_9)
12	(1_12)	NMT service reset communication indication
13	(1_13)	see (1_12)
14	(1_14)	see (1_12)

Figure 8-33: Central collimator state machine for initialization and communication of the collimator

As already mentioned, both collimation axes, as well as the X-ray beam hardening and the light module, act independently in parallel. Thus, the behavior of these components is modelled by individual state machines.

Figure 8-34 shows the state machine of the logical component *ControlCollimationAxis* being part of the logical architecture of the component *CollimateBeam (*see Figure 8-29)



#	Name	Documentation
1	(2_0)	CreateInstance
2	(2_1)	Collimator FSA enters ready state
3	(2_2)	LOCK command received
4	(2_3)	UNLOCK command received
5	(2_4)	Move event received
6	(2_5)	Stop event received
7	(2_6)	Local control activated
8	(2_7)	Local control deactivated
9	(2_8)	Local control activated
10	(2_9)	Move event received (see 18)
11	(2_10)	Move command from local control
12	(2_11)	Move event received
13	(2_12)	Stop event received
14	(2_13)	Move event received (see 18)
15	(2_14)	Fault or Error detected
16	(2_15)	Reset command
17	(2_16)	Collimator FSA leaves ready state
18	CollimationHomedIdle	Collimation axis succesfully homed
19	CollimationHomedInit	Collimation axis succesfully homed
20	ErrorHoming	
21	HomeCollimationIdle	Start homing of collimation
22	HomeCollimationInit	Start homing of collimation

Figure 8-34: Collimator state machine for one collimation axis

In the three states *SystemControlLocked*, *SystemControl* and *LocalControl* movement commands can be sent over the connecting interfaces to the logical block *ControlBladePosition* (see Figure 8-31) triggering changes within its state machine. The corresponding state machine is shown in Figure 8-35.

Again, this state machine starts with an *Initialize* state. After successful initialization, a state change into the *Idle* (stopped) state is triggered. In this state movement commands of the logical block *ControlCollimationAxis* are being processed.

In order to act on a valid position request, the state machine transitions between different states for accelerating, deaccelerating or moving the axis with a constant speed.

After receiving a valid positioning request, the mechanics is first accelerated, moves constantly afterwards and is decelerated at the end.

It is worth to note that during the movement the control logic of the collimator accepts commands being communicated by the logical block *ControlCollimationAxis*, allowing to move to a new position.

Figure 8-35 further shows an error state, which will not be further explained in detail. In addition to the state machines mentioned so far, further state machines are required to model the

collimator behavior. The tracing of the logical components, and by this their state machines to corresponding SW task will be discussed in section 8.2.7.1.

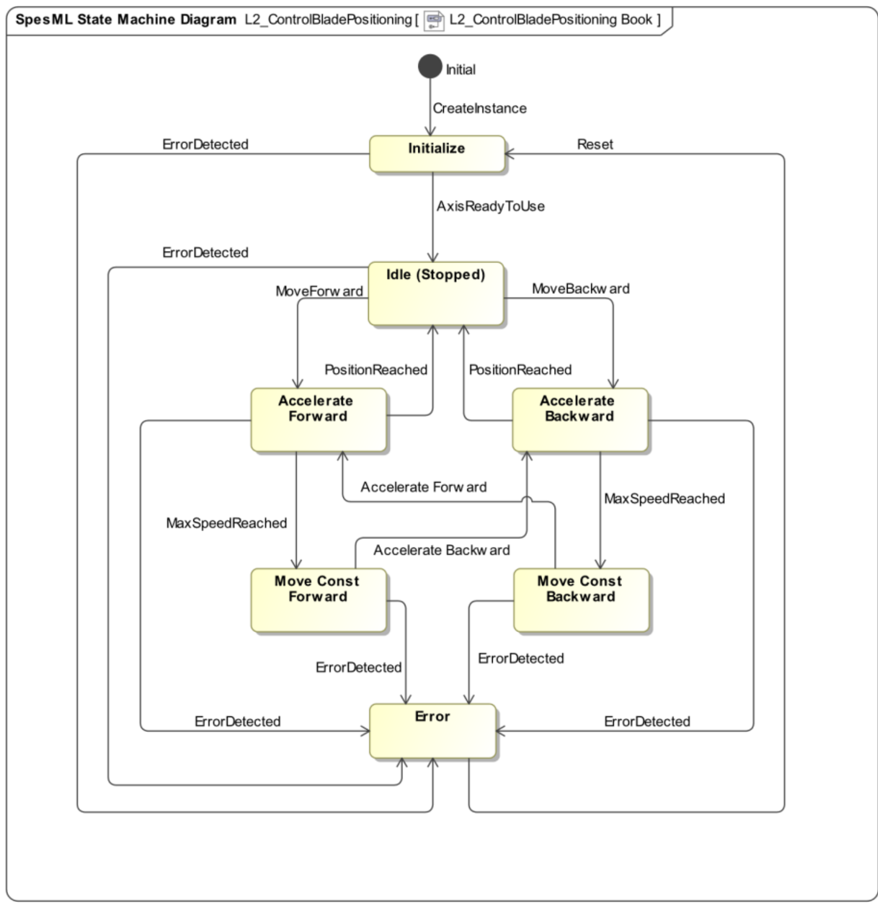


Figure 8-35: State machine for moving a collimation axis

Test-Case for State Machine Collimator Light State Machine

The behavior of logical components in SpesML is modelled by associated state machines. To tests the simulation capabilities of the SpesML plugin, a simple example was chosen, namelythe state machine inside the collimator being responsible for switching on and off the light within the collimator, as shown in Figure 8-36. The light itself passes through the collimation elements

and provides the user with a visual indication of the area which will be exposed to X-ray photons.

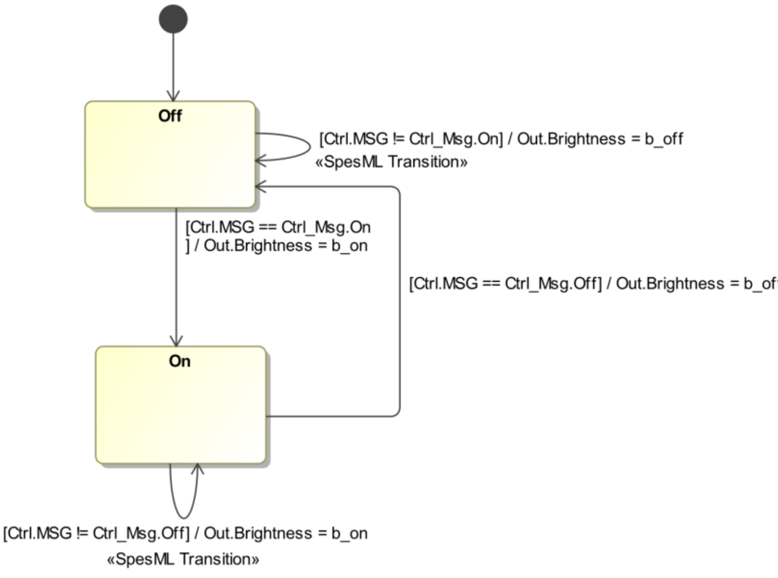


Figure 8-36: Light state machine of the collimator

The transitions between the two states are controlled by the value of the incoming control messages, namely *Ctrl_Msg.On* and *Ctrl_Msg.Off*, acting as the guard condition. The corresponding effects is a brightness value of *b_off* or *b_on* Lumen on the output of the logical element.
To test this state machine, the testcase shown in Figure 8-37 was created. The logical *Stimulus* element injects control messages into the *Testee* hosting the light state machine. The *Observer* on the other hand is aware of the control messages and is probing the output of the *Testee*. By correlating the expected light brightness value with the requested state described by the control messages, the *Observer* can analyze, if the behavior of the *Testee* state machine works as expected and sets the binary output value *of the type IResult* accordingly.

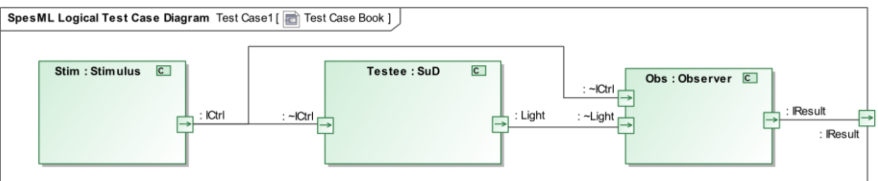


Figure 8-37: Testcase architecture

8.2.5.4 Remarks

The logical view shows the decomposition of the system into fine grain logical components as well as the interaction between these logical elements. Depending on the chosen abstraction level, the logical architecture will reflect the impact of technology choices for the realization of the logical elements. In the example of the collimator, the decision to move the collimation blades with stepper motors instead of a servo motor manifests itself in the underlying control schema and therefore also in the logical architecture of the control components and its interfaces. The universal interface model concept provides a very precise way of specifying the inputs and outputs of individual interfaces by using channels with dedicated data types. Such a precise description ensures that elements of the architecture are compatible interface wise, which is very important when, as in our case, the complete model consists of sub-models being created by different individuals (or teams) in parallel. Additionally, it is the base line for simulating the logical architecture. Being able to set up and successfully run a simulation of a logical architecture with the underlying element state machines increases the maturity level of the architecture significantly.
Lastly, from a practical point of view, the introduction of sequence diagrams to the logical viewpoint could help modeling the dynamic interaction, showing the timely sequence of data flow between elements of the logical architecture. Sequence diagrams would allow to model and showcase typical interactions between components whose behavior is precisely described. Additionally, these sequence diagrams could form the baseline for integration testing.

8.2.6 Technical View

The goal of the technical viewpoint is to transform the (mainly) platform independent models of the logical viewpoint to platform specific models. All logical components will be hosted by actual technical implementations.

8.2.6.1 Context

The technical context describes how the system under development interfaces with its external technical elements and stakeholders.

Technical Context of the simplified radiography system

Figure 8-38 shows the technical context of the simplified radiography system which is typically installed in the radiology department of a hospital. Obviously, the system needs to be connected to the mains grid and to the corresponding IT systems in order to be able to function properly. Additionally, a purely mechanical facet has been introduced as well, namely the mechanical interface needed to mount the simplified radiography system to the ceiling of the room. Modeling this aspect correctly is crucial for a successful integration of a system in its supersystem environment and therefore should not be overlooked.

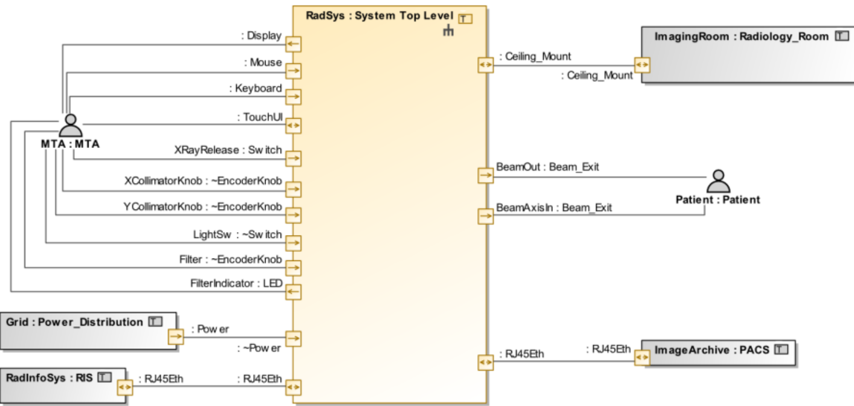


Figure 8-38:Technical context of the simplified radiography system

Context Technical View Collimator

Figure 8-39 shows the technical context of the collimator. The logical block *ProvideXray* from Figure 8-26 is realized by the technical component *XrayTube*, *ControlSystem* by the *SystemController* and *ProvidePower* by a *PowerSupply*. Further the context diagram shows the stakeholders MTA and Patient and how they interact technically with the collimator. For example, the MTA sets the collimation axis by turning an *EncoderKnob.*

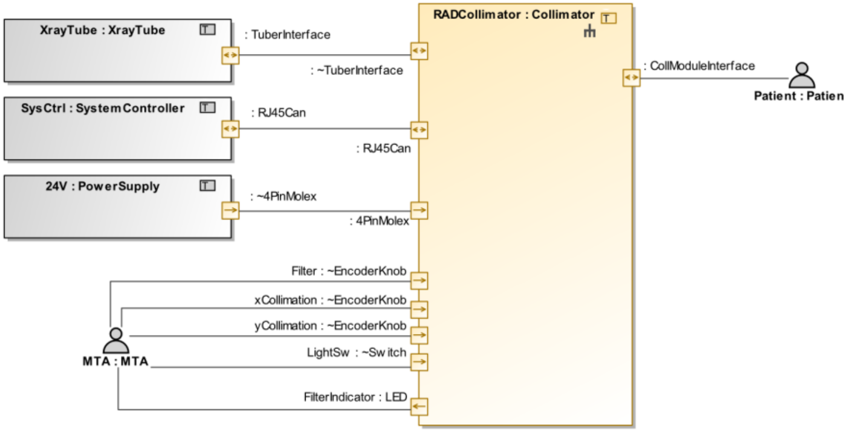


Figure 8-39: Technical context of the collimator

8.2.6.2 Technical Architecture

The technical architecture shows the internal technical elements of the system under development and how they interact with each other to realize the logical architecture and by this the system functions.

Technical Architecture of the simplified radiography system

The details of the technical view of the collimator are shown in Figure 8-4. The *HV Generator* is supplied with electrical energy by the *PowerDistribution* unit. Its job is to convert the electrical energy into a high voltage pulse driving the *xRayTubeAssy* in which electrons are generated, accelerated, and finally stopped on a rotating anode. During this process X-rays are generated, which are hardened and collimated afterwards inside the *Collimator*. In order the measure the applied X-ray dose to the patient, a DAP-Meter is used. The *XrayTubeAssy*, *Collimator* and *Dap-Meter* are directly connected to each other. This whole assembly is mounted to the *TubeStand*, which can position the assembly in the room, to align the X-ray source with the body part of the patient to be imaged. While passing through the body of the patient, the X-rays are attenuated and are subsequently detected by the *Detector* which converts them into electrical signals holding the information of the X-ray image. The *RadSystemController* is the main controller in the system, orchestrating the interaction and timing dependencies of the elements within the system. In order to interact with the system, the medical technical assistant has two main user Interfaces. One is the *TUI*, an HMI panel directly mounted next to the *XrayTubeAssy*. Additionally, there is an *ImagePC*, a conventional workstation with display, keyboard, and mouse. This workstation runs the software needed to register the patient information, to parametrize the simplified radiography system and to display the medical X-ray images. It also interfaces with the hospital network to retrieve and store the relevant data.

Technical Architecture of the Collimator

The details of the technical view of the collimator are shown in Figure 8-41. The central software execution subsystem *ControlModule* implements the communication with the system and controls the electromechanical drives of the beam hardening and collimation units. The user can interact with the collimator via the user interface *UIModule*, which contains rotary encoders for the aperture and filter movements, as well as a switch for the light and a LED indicating the filter position. Figure 8-41 further shows the collimator *Covers* (mechanical element) and the electromechanical module *HardeningModule*, which realizes the hardening functionality, the *LightModule*, which provides the collimation light field as well as two instances of the *CollimationModule*, one if it being instantiated for the x- and one for y-direction.

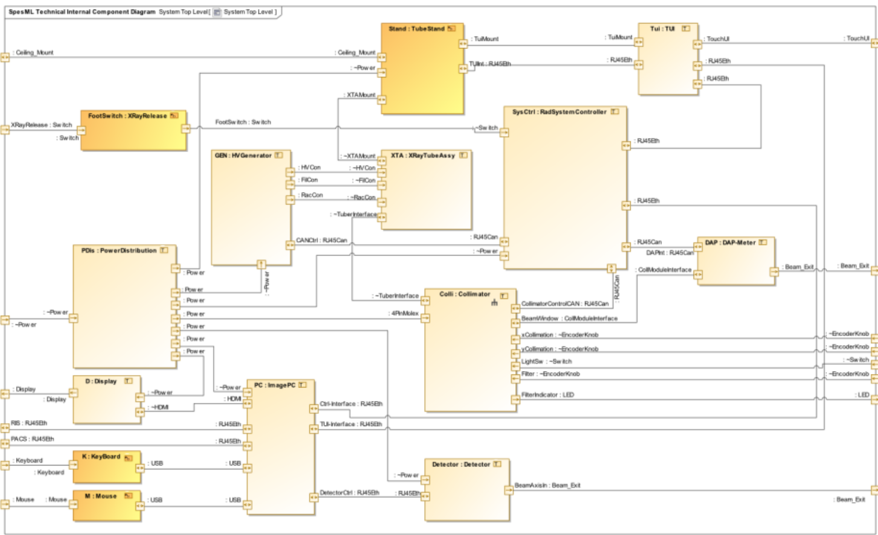


Figure 8-40: Technical view of the simplified radiography system

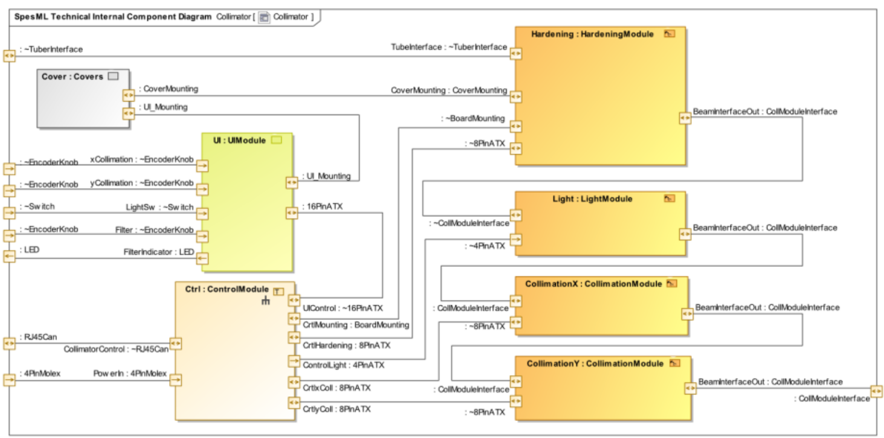


Figure 8-41: Technical view of the collimator

The *software execution subsystem* (*ControlModule*) was modeled in detail to reflect the interaction between the software and the hardware of the collimator.

Since the *software execution subsystem* (*Controller* in Figure 8-42) is typically not able to directly operate electrical elements with significant electrical power, driver components are used between consumers and controllers (see Figure 8-42). For example, the typical output of a microcontroller does not provide enough power to directly drive a stepper motor. For this purpose, the hardware driver *StepMotCtrl* in Figure 8-42 is used. This driver expects two digital control input signals, one providing the direction of the movement and one providing the step

pattern. Based on this input, the driver provides a two-phase output signal with enough electrical power to drive a stepper motor. Further the *StepMotCtrl* implementation provides an output to the controller representing the signal of the light barrier for homing purposes.
Within Figure 8-42 the software execution subsystem shows the external interfaces of the microcontroller being used as execution platform for the software.
Following Figure 8-43 the *software execution subsystem* consists of an electrical and a software part. The electrical part represents the controller to be used (microcontroller) to execute the software. The software is represented by the task architecture, which is shown in detail in Figure 8-44.

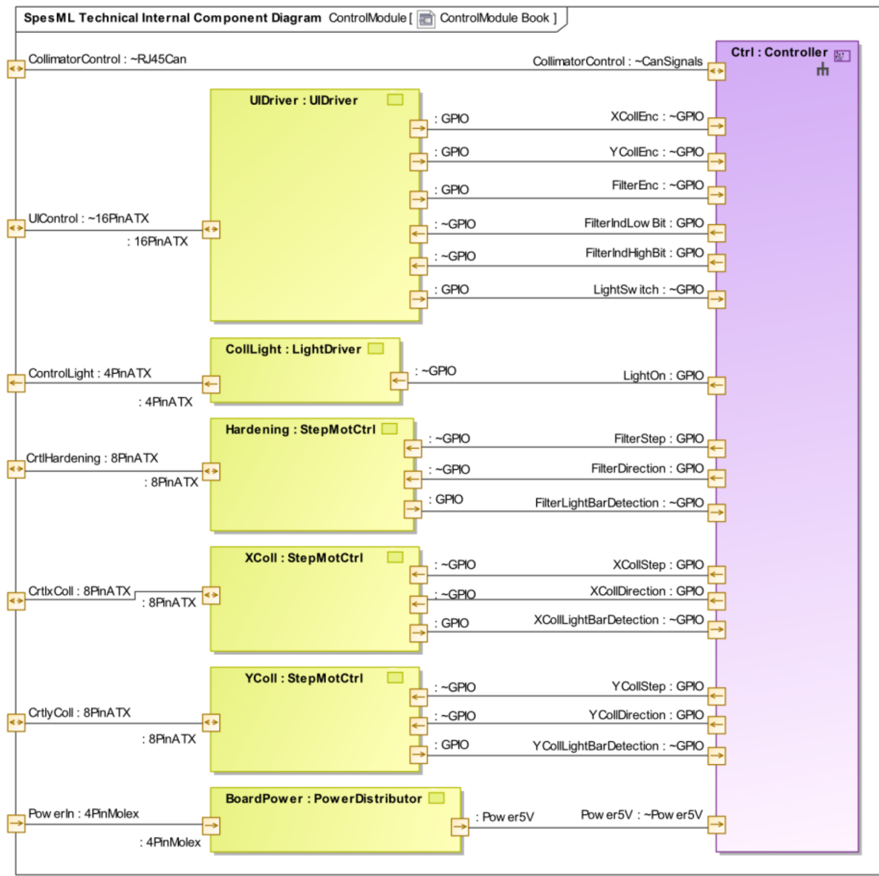


Figure 8-42: Software execution subsystem of the collimator

As shown in Figure 8-43, the electrical signals of the controller are controlled by an execution component on which a task architecture is executed.

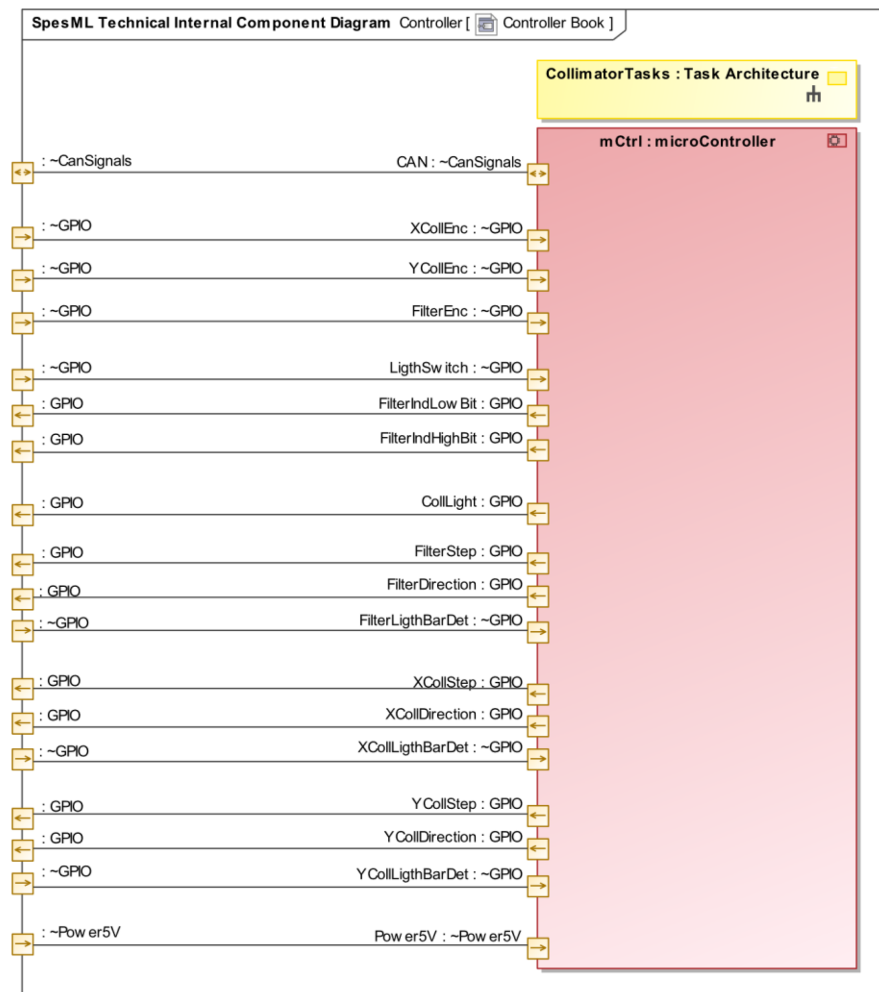


Figure 8-43: Execution Component and Associated Task Architecture of the collimator

The task architecture consists of individual tasks that implement well-defined functionalities (see Figure 8-44). To identify separated independent tasks, one guideline is to identify logical blocks being modeled by an individual state machine. Typically, these logical blocks behave independently from other logical blocks and are only loosely coupled to each other.

Within the collimation example we can identify the collimator communication, two collimation axes, the light, and the hardening filter to behave independently from each other over time. The user for example can change the hardening filter, while the x-collimation is still moving. This independent behavior is modeled by assigning logical components to a dedicated task, which later will be executed independently on the software execution platform. Following this approach seven tasks, being shown in Figure 8-44, could be identified to realize the logical model.

Figure 8-44 further shows the interfaces between the software tasks, as well as the interfaces to the electronic device. For example, the output *Direction* of the task *ControlHardenBeam* directly controls the hardware output *FilterDirection* of the controller. To reflect the tracing between the software task output and the electrical pin of the microcontroller, a tracing matrix, as shown in Figure 8-45 is used.

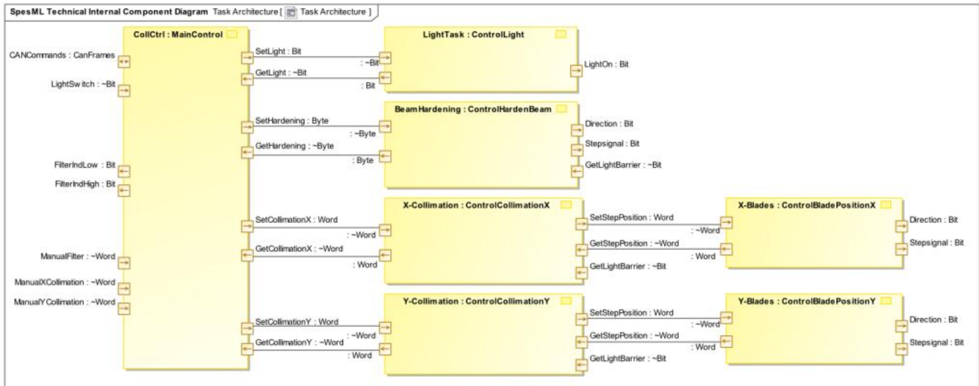


Figure 8-44: Task architecture of the collimator control software

Legend
A = allocate port to

	microController [Collimator	in +FilterEnc : ~GPIO	in +FilterLigthBarDet : ~GI	in +LigthSwitch : ~GPIO	in +Power5V : ~Power5V	in +XCollEnc : ~GPIO	in +XCollLigthBarDet : ~GI	in +YCollEnc : ~GPIO	in +YCollLigthBarDet : ~GI	inout +CAN : ~CanSignals	out +CollLight : GPIO	out +FilterDirection : GPIC	out +FilterIndHighBit : GPI	out +FilterIndLowBit : GPI	out +FilterStep : GPIO	out +XCollDirection : GPIO	out +XCollStep : GPIO	out +YCollDirection : GPIO	out +YCollStep : GPIO
Software [Collimator-SHS::Technical Viewpoint		1	1	1		1	1	1	1	1	1	1	1	1	1	1	1	1	1
ControlBladePositionX																1	1		
ControlBladePositionY																		1	1
ControlCollimationX							1												
ControlCollimationY									1										
ControlHardenBeam			1									1			1				
in +GetLightBarrier : ~Bit	1		A																
in +SetHardening : ~Byte																			
out +Direction : Bit	1											A							
out +GetHardening : Byte																			
out +Stepsignal : Bit	1														A				
ControlLight											1								
MainControl		1		1		1		1		1			1	1					

Figure 8-45: Tracing of task ports to microcontrollers ports (Port To Execution Allocation Matrix)

8.2.6.3 Remarks

The SpesML plugin allows to distinguish between different types of technical components, namely mechanical, mechatronic, and electrical/electronic components. This is important to be able to assign the realization of these elements to domain-specific teams, like mechanical engineers or PCB designers.
Additionally, there is a complete set of elements being dedicated to modeling the software execution components, consisting of an execution platform running a software task architecture consisting of individual software tasks. This allows software experts to model their part of the technical architecture.
The mapping capability of the SpesML plugin allows mapping the elements of the logical architecture to hardware and software elements respectively, providing valuable insight into the technical realization concept of the system and providing the baseline for impact analysis.

8.2.7 Crosscutting Concepts

8.2.7.1 Traceability

One task of the system architect is to decompose the system into logical blocks, which could be deployed uniquely to an element within the technical architecture. Doing so, the party being responsible for the implementation of this element exactly knows the functionality to be realized within this element. In case the logical block will be realized by a dedicated development discipline (SW, HW, Mechanics), the according element can be handed over to the according discipline specific team or individual. In case the logical block to be realized will still have to be implemented by different disciplines, a further decomposition is required until the level of a single discipline is reached.
An example of a discipline specific deployment is shown in Figure 8-46. The SW specific logical blocks *ControlCollimationAxis* and *SystemCoordTranslation* will be realized by the SW tasks *ControlBladePosition*, whereas the SW specific logical blocks *ControlBladePosition* and *AbstractMotorCtrlHW* will be realized by the SW task *ControlCollimation* (see also Figure 8-31 and Figure 8-44) . The logical block *DriveMotor* will be realized by the electronic device *StepMotCtrl* on the *ControlModule*, whereas *AbsorbLight&xRays* as well as *MoveBlades* are realized by the mechatronic element *CollimationModule*. According Figure 8-46 every logical block is uniquely deployed to a technical element.

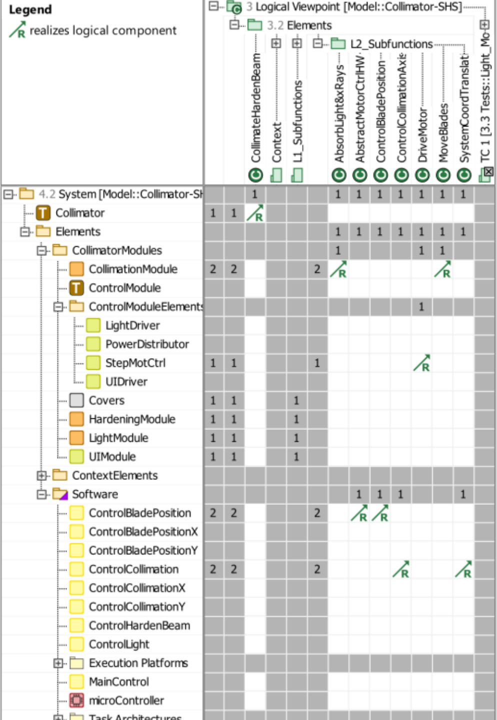


Figure 8-46: Unique deployment of logical blocks to discipline specific technical elements

8.2.7.2 Impact Analysis

Even in the healthcare industry, changes to requirements are quite common. Integration of changes into an existing system model is time-consuming and error prone. The impact analysis of the SpesML plugin offers a way to get a quick first insight on how the change affects the whole model and its elements.

Selecting one or multiple requirements, a graph is generated including all related elements that could be affected by the initial change. The listed elements are covering the functional, logical view and technical view. This way the impact of changes can be understood and assessed easily. Figure 8-47 shows the traceability for the design requirement “*Filter 1*” of the hardening performance. Following the trace links, it can be easily identified that the function *Harden X-Ray Beam* and accordingly the logical blocks *HardenBeam* and *ShieldScatter*, being realized by the *HardeningModule* and the *Covers,* are impacted in case of any change of the requirement “*Filter 1*”.

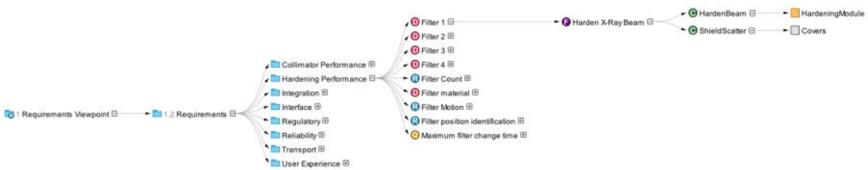


Figure 8-47: Dependency graph from requirements to technical element

8.2.8 Final Thoughts

Trying to introduce the concept of model-based systems engineering to industrial development departments is a challenging endeavor as it affects the way how products are developed fundamentally. Based on our own experience, just providing the development organization with an UML or SysML modeling tool and some basic language training is not sufficient. Developers are experts in their individual development domain, but they are by far no modeling experts. This results in the creation of ambiguous models as UML or SysML do not provide a clear guideline how to model aspects of a system architecture consistently. The resulting models are not always helpful for communication between cross-functional teams and the reuse potential of such models is significantly limited. Therefore, a modeling methodology approach as realized in SpesML together with its implementation in the MagicDraw plugin is a step forward. The SpesML plugin is a domain specific language (DSL) implementation and by this guides the user to build models, e.g. by clearly separating the problem space (requirement and functional view) from the solution space (logical and technical views). Additionally, the plugin allows to create only the appropriate modeling elements that match the actual modeling view. Furthermore, automatic continuous execution of well-formedness rules in the background gives immediate visual feedback to the users if the model does not comply to the SpesML methodology.
Today, models being created in an industrial environment usually only cover selected aspects of a cyberphysical system by focusing on dedicated hardware or software architecture models only. The SpesML approach allows to model all aspects of a cyperphysical system by clearly reflecting the interaction of hardware and software components to realize the functionality of the system.
The capability to run simulations increases the maturity level of logical models. MontiArc [50] was chosen as the simulation framework, a solution which is not widely known in industrial applications. Therefore, extending the simulation capabilities to interface with Simulink could further increase the application range for the SpesML-Plugin in industrial applications.

8.3 Anomaly Detection System (Qualicen)

The following section describes the application of SpesML on an anomaly detection system. After a general overview over domain and case study, the chapter describes the different views according to the SpesML methodology. A discussion of the case study concludes the section.

8.3.1 Domain Context

Qualicen is a company that focuses on improving engineering methods with the help of AI methods and model-based engineering. This case study focuses on an anomaly detection system (ADS). An anomaly represents a variation of the system behavior relative to the defined normal behavior. Anomalies can surface in any kind of system (not just control systems) and can originate from the system itself (e.g. due to physical wear) or from external influences (e.g. an intruder trying to compromise a system). An ADS is a system that is used by Qualicen to demonstrate technology and which was discussed in detail with industrial partners in previous communications.

One of the challenges of the ADS is its crosscutting nature. On the one hand, the systems for which the system should detect anomalies are often control systems for production or transportation. On the other hand, the ADS itself is focusing on data processing and machine learning to analyze sensor data to find anomalies. Hence, there are different system types involved within the context of the ADS.
This cases study aims to evaluate whether mode-based systems engineering based on SPES and SpesML can cope with such systems. The goal is to model the system on a high level, from requirements down to the technical architecture. However, the study does not aim at modelling algorithms or detailed data processing rules. The main target is to identify the relevant functions, components and an appropriate technical realization and to be able to identify impacts on the architecture in case of changing requirements.

8.3.2 Case Study Introduction

The system under development of this case study is an anomaly detection system. The goal of this system is to detect when a target system is not operating normally. The detection is performed in a two-stage process. In the first stage (benchmarking stage), the ADS records sensor values of the system while it is operating normally. In the second stage (monitoring phase) the system uses the recorded sensor values to issue a warning when the target system is operating *differently* from the benchmarking stages (with a definition of different provided by the appropriate choice of the machine learning approach). The training stages and the benchmarking stages may also be interleaved to allow for incrementally updating the current system benchmark. Additionally, to benchmarking and monitoring, the ADS supports engineers to gather information about the cause of an anomaly by providing means to query the recorded sensor values in various ways.

The target system can be any system whose operation is monitored by a number of sensors. Examples are production systems, logistic systems, transportation systems, information systems etc. Sensor data may not only be numerical measurements but also, for example, operation logs or similar data.

The ADS is deployed on a potentially distributed set of execution units to allow for speedy analysis of the received sensor data and at the same time provide interfaces to interact with users and external applications.

The ADS is currently not a commercial product of Qualicen but rather an application to demonstrate Qualicen-owned technology. Parts of the system have been built as prototypes before and simulations of an example target systems (“Cooling System”) existed before the project.

8.3.3 Requirement View

The requirements for anomaly detection are described in two levels of detail:

- Level 1: Stakeholder Requirements
- Level 2: System Requirements

The stakeholder requirements describe high-level capabilities that the ADS must fulfil, for example the ability to create benchmarks or continuous monitoring. A SpesML Requirements Table captures the stakeholder requirements. SpesML Requirements Packages group requirements with similar topics (e.g. Benchmarking). As the stakeholder requirements are high-

level capabilities, they were classified as requirement type “Capability”. Figure 8-48 shows an excerpt from the stakeholder requirements.

Criteria
Scope (optional): Stakeholder Requirements ... Filter:

#	Name	Status	Text	Requirement Type
1	Stakeholder Requirements			
2	Benchmark			
3	Benchmark Trigger	reviewed	When triggered by the administrator the AD shall create a benchmark of the sensor data of the SWS.	Capability
4	Client Interface			
5	Client Interface	reviewed	The AD shall provide a user interface for control and configuration.	Capability
6	Control commands	reviewed	The AD shall receive control commands via the client-interface.	Capability
7	Data Querying	reviewed	The AD shall allow to retrieve stored data with a query.	Capability
8	Configuration			
9	Benchmark Configuration	reviewed	The AD shall allow to configure which data to include in a benchmark.	Capability
10	Monitoring Configuration	reviewed	The AD shall allow to configure which data to monitor against which benchmark.	Capability
11	Monitoring			
12	Anomaly Detection	reviewed	While monitoring, the AD shall detect anomalies within the sensor data.	Capability

Figure 8-48: Excerpt of the stakeholder requirements

The system requirements were derived from the stakeholder requirements. A second SpesML Requirements Table was used to capture the system requirements and again used requirements packages for grouping. The system requirements were classified as either “Functional” or “Quality”. Figure 8-49 shows a section of the system requirements. In a second step, the quality requirements were used as a starting point and further functional requirements were derived from them. These requirements were documented in the same requirements table as the other system requirements.

There is a tracing link between the stakeholder requirements and the system requirements in order to document which system requirement addresses which stakeholder requirement. There are further tracing links connecting quality requirements and derived functional requirements. Section 8.3.7will provide details on the traceability.

#	Name	Status	Text	Requirement Type
1	System Requirements			
2	Functional Requirements			
3	Reporting			
37	Detect Anomalies			
38	Describe Monitoring Source	reviewed	When the system receives a monitoring source it checks wether the given source is part of an available benchmark and marks the given source for the currently configured monitoring.	Functional
39	Available Benchmark	proposed	The system checks if the given source is part of an available benchmark.	Functional
40	Monitoring Source	proposed	The system marks the given source for the currently configured monitoring.	Functional
41	Describe Monitoring Benchmark	reviewed	When the system receives a monitoring benchmark, it checks whether the given benchmark is compatible with the given monitoring source, if any monitoring source has been given, and selects the benchmark for the currently configured monitoring.	Functional
42	Evaluation of Monitoring Benchma	proposed	The system checks if the received monitoring benchmark is compatible with the given monitoring source, if a monitoring source is given.	Functional
43	Benchmark Selection	proposed	The system selects the received benchmark for the currently configured monitoring.	Functional

Figure 8-49: Excerpt of the system requirements

8.3.4 Functional View

The functional view is broken down into the sections functional structure, containing white-box and black-box models, mode models, and finally a brief evaluation.

8.3.4.1 Structure

In the functional view, the functional requirements are implemented in the form of black-box and white-box functional models. These models describe the functions and sub-functions of the anomaly detection system. Figure 8-50 shows the functional black-box model with the system functions "Manage Benchmarks" (Controls the creation and updating of benchmarks), "Detect Anomalies" (Performs monitoring and creates warnings) and "Reporting" (Allows to query the stored data).

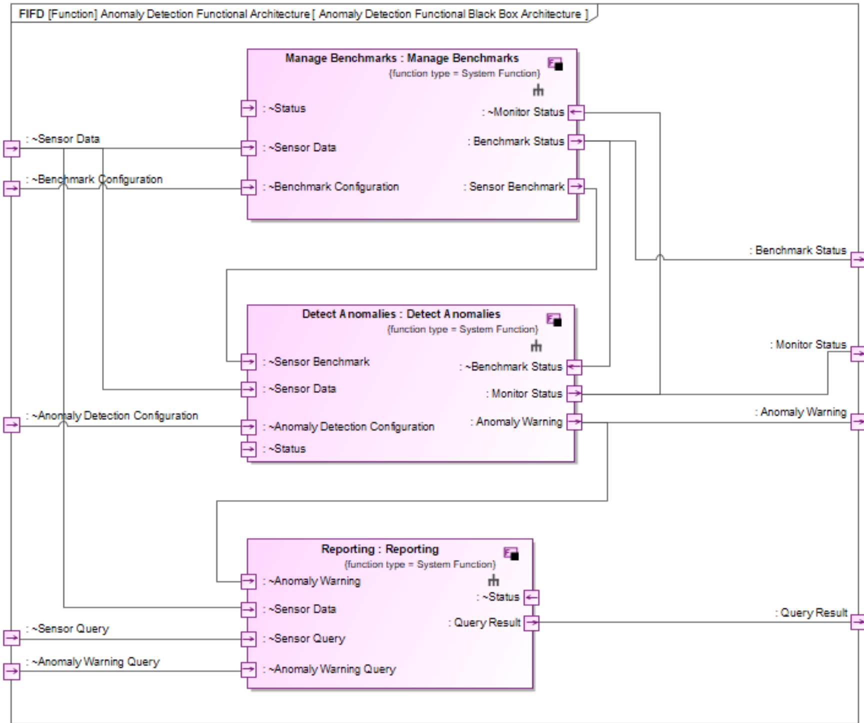


Figure 8-50: Functional black-box model of the ADS

The black-box functions are in turn broken down into functional white-box models containing the white-box functions that together realize the black-box function. The functional white-box model hence shows a glass-box view of a black-box function. Figure 8-51 shows an example of the white-box model for the black-box function "Reporting". In the white-box model, it is visible

that the reporting function handles storage and retrieval of various data, such as sensor data and warnings.

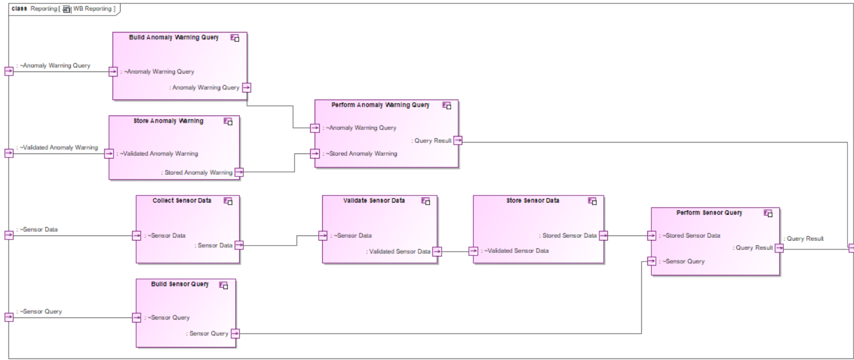


Figure 8-51: Functional white-box model of the reporting function

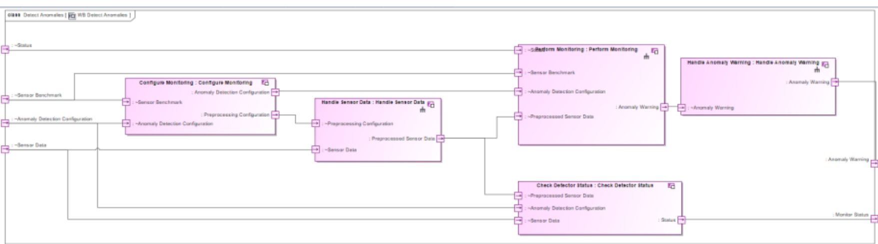


Figure 8-52: Functional white-box model of the monitoring function

In the example for the reporting function the white-box functions are not further decomposed. However, in our case study this was necessary for the other black-box functions. Figure 8-52 shows the functional white-box model of the monitoring function. Here, three out of the five white-box functions are further decomposed into sub-white-box function (visible by the “fork” icon on the upper right corner of a function).

8.3.4.2 Mode Model

In the functional black-box architecture, there are internal "mode" channels in addition to the channels to the system boundary that model interaction between black-box functions. The channels "Benchmark Status" and "Monitor Status" are of particular importance, as the internal status of the system is decomposed here. A SpesML mode model was created for these channels. The corresponding mode model shows the possible transitions of the benchmark and monitoring status. Each state corresponds to a value of the enumeration datatypes of the two mode channels.

The mode model for the benchmark and monitoring status is shown in Figure 8-53: Mode model of the ADS

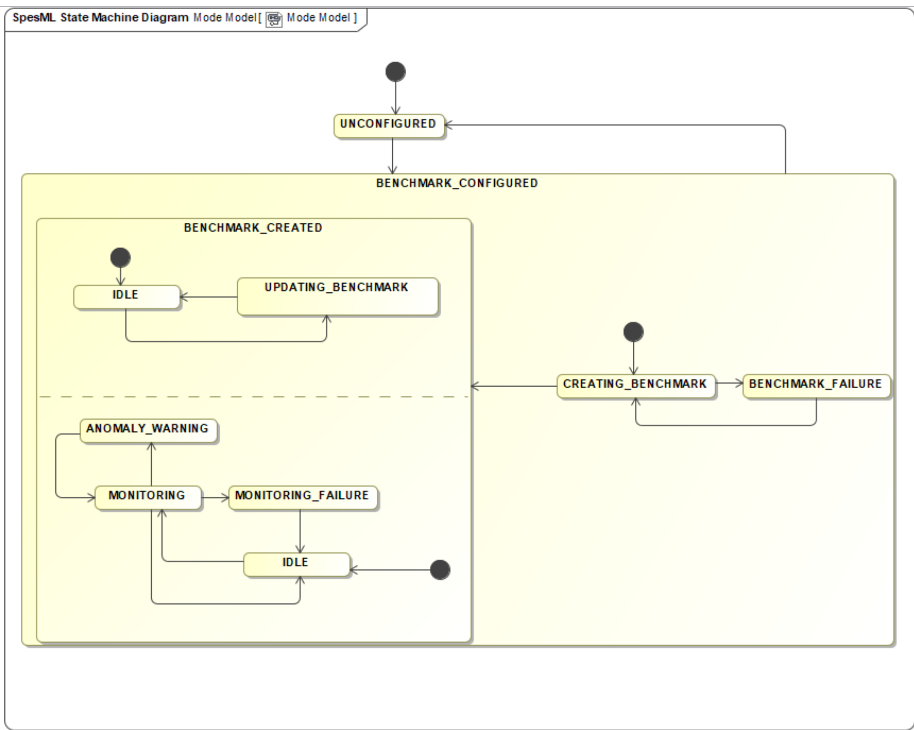


Figure 8-53: Mode model of the ADS

8.3.4.3 Evaluation Result

In case of the functional view, the model captures the functional structure of the ADS system with the SpesML modelling elements. By following the suggested process of defining black-box functions and refining those with white-box functions a structured functional specification was obtained.

The model furthermore provides a high-level view of the mode behavior of the system by converting internal mode channels to a mode model in systematic way.

In this case study, a model for the functional context was omitted. As the readers will see in the next section, the model instead contains a context model for the logical view. Furthermore, behavior models for the functional white-box functions were not included.

8.3.5 Logical View

The logical view of our model describes the context, the logical structure and decomposition, the logical behavior and, finally, again a brief evaluation.

8.3.5.1 Context

Figure 8-54 shows the logical context of the ADS, including two stakeholders (administrator and surveillor) and three other systems in the context. These are:

- the monitored system, where anomalies are to be detected,
- the configuration system controlling configuration of the ADS, and
- the monitoring system, which uses the warnings of the ADS.

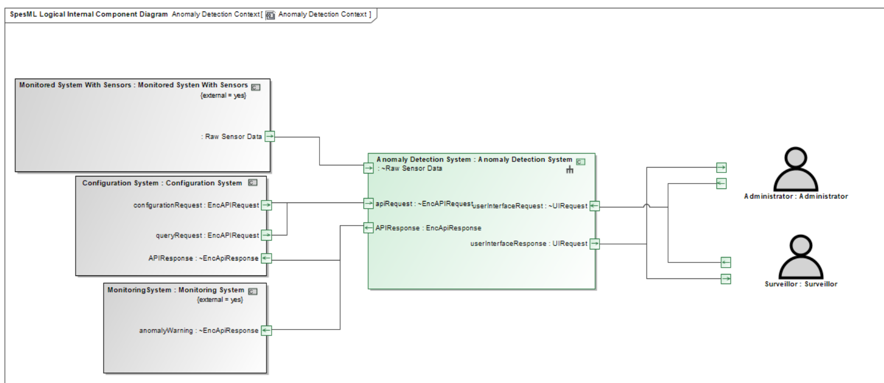


Figure 8-54: Logical context of the ADS

8.3.5.2 Structure

The components of the logical view realize the defined system functions. Logical components can be successively decomposed into sub-components until a unique assignment of white-box functions to components is possible. Figure 8-55 shows the logical view of the ADS at the top level. It consists of six components in total. All but one of these components shown are decomposed into further sub-components (again identifiable by the fork icon on the top right corner of a component).

As can be seen, the structure of the logical view is very different to the structure of the (black-box) functional view, as the logical view emphasizes other aspects of the system.

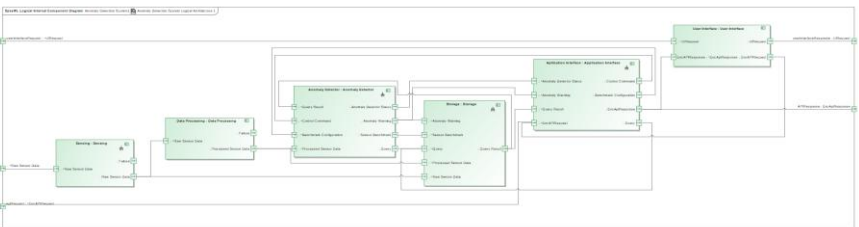


Figure 8-55: Logical View of the ADS – Top level

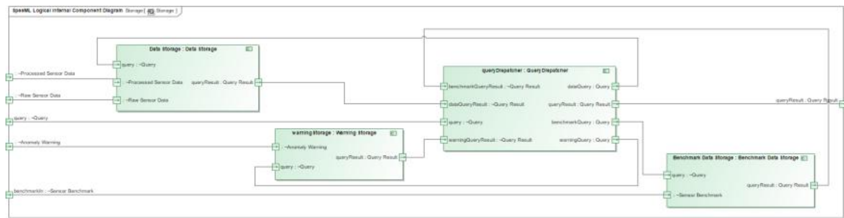


Figure 8-56: Logical decomposition of the storage component

As mentioned above, one of the top-level logical components (Anomaly Detector) was singled out as a subsystem and started a new development process and model with the Anomaly Detector as a starting point. In the SpesML workbench, a new model for the anomaly detector subsystem was created and further detailed in the new model. To this end, subsystem requirements and a subsystem logical architecture were created as well. The ADS model references the subsystem model.

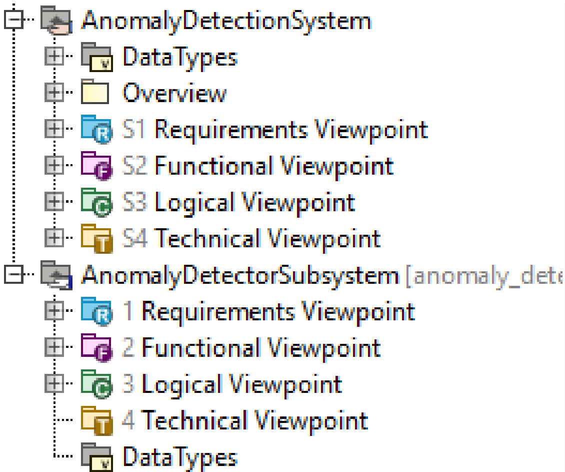


Figure 8-57: Models for the anomaly detection system and anomaly detector subsystem, again containing all views

8.3.5.3 Behavior

Components that are no longer broken down further can be provided with a behavior in the form of a state machine. It describes how such an atomic component produces outputs based on the input it receives. An example of such a state machine is shown in Figure 8-58. This state machine controls the status of benchmarking and monitoring (similar to the functional mode model). The

states refer to the modes of operation (e.g. Benchmarking). Based on the input it receives, it changes states and sends signals to other components.

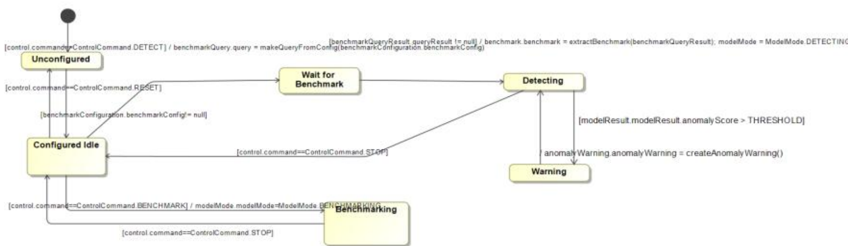


Figure 8-58: Example state machine for the model controller component

The logical components and the behavioral state machines were designed in such a way that a simulation can be carried out. For this purpose, a test system with mock-ups was developed, which provides the ADS with inputs, monitors the outputs and checks them for correctness. Figure 8-59 shows the test setup. The behavior of the overall system results from the composition of the behavior of the components, according to the Universal Interface Model.

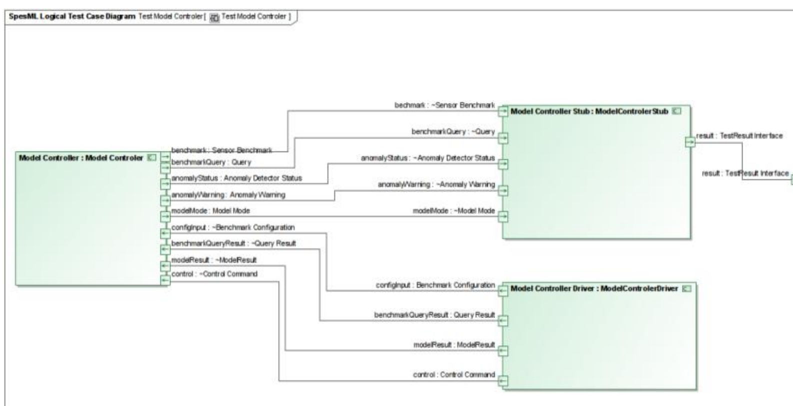


Figure 8-59: Logical test setup with the model controller on the left (System-under-Test) and environmental components on the right

8.3.5.4 Evaluation result

The case study was able to model a solution architecture for the ADS that realizes the functions defined in the functional viewpoint. One challenge was that a large part of the overall behavior of the ADS addresses data processing and only a small part addresses control. As state machines are more suitable for control parts of the behavior, the case study only models selected parts of the system's behavior.

SpesML provides means to describe data processing behavior (using executable functions; however, it was out of scope of the evaluation of this case study.

8.3.6 Technical View

The technical view shows the realization of the system in the form of technical components. In the general case, this may include mechatronic components, for example. In the technical view we would then break down the system into mechatronic and software components and could model the software subsystem on a further level of granularity. However, the anomaly detection system is a purely digital system and contains only the necessary execution and communication components in addition to software. Therefore, in this case study the technical view does not contain any mechatronic components and therefore we do not explicitly identify a software subsystem. The resulting hardware architecture, which shows only these execution components and networks, is shown in Figure 8-60. The hardware architecture consists of four types of execution components, the Client Interface Server to execute the tasks that relate to providing an external user interface, the System Control Server for the overall coordination tasks, the Data Server for data storage and preprocessing, and the Analysis Node for the actual sensor data analysis. Additional to the execution hardware, the model contains network elements for data and control messages.

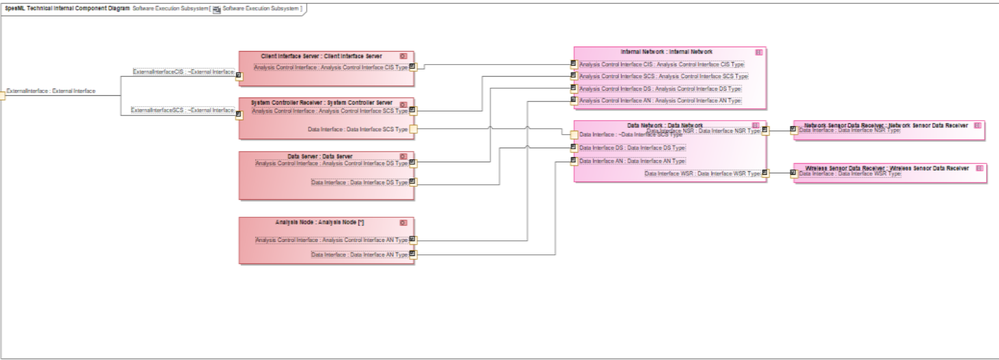


Figure 8-60: Hardware architecture of the ADS

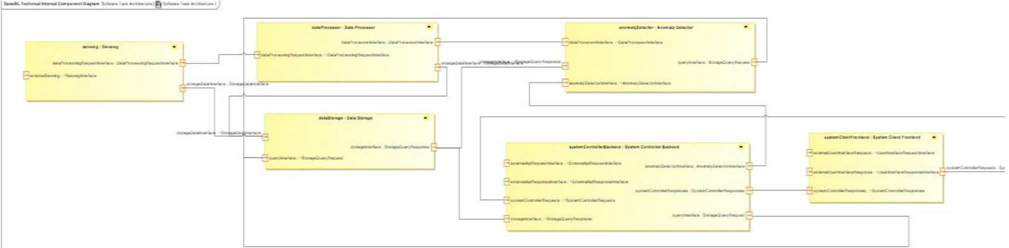


Figure 8-61: Task architecture of the ADS

Furthermore, the ADS model contains a task architecture. This represents the software tasks of the system and their interconnection. Components of the logical architecture are mapped to tasks of the task architecture in the context of tracing. See Figure 8-61 for an overview of the task architecture.

In order to relate the tasks of the task architecture to the execution components of the hardware architecture, appropriate mappings were created. In fact, in SpesML there are two types of mappings, one relating tasks to execution components, and another one to relate task ports to ports of the execution hardware. The former shows which software tasks are deployed (and therefore will get executed) on which execution hardware. The latter shows which technical interfaces and network equipment will be responsible to transmit messages that are exchanged via the task ports. This case study contains both types of mappings. Figure 8-62 shows the mapping of tasks to execution components and Figure 8-63 shows the port mappings.

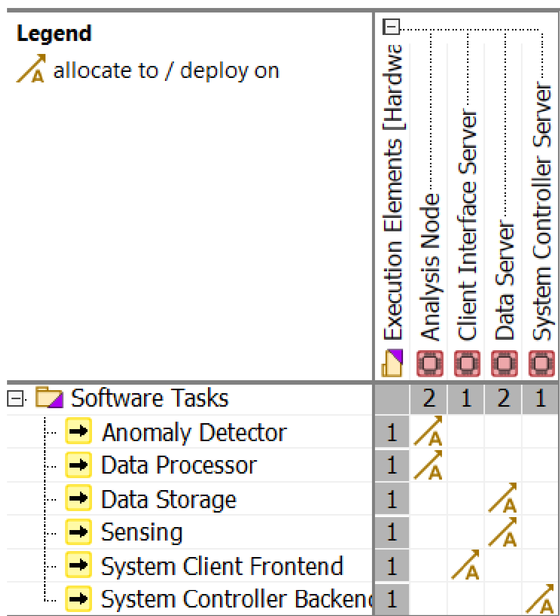


Figure 8-62: Mapping of tasks to execution components of the technical architecture

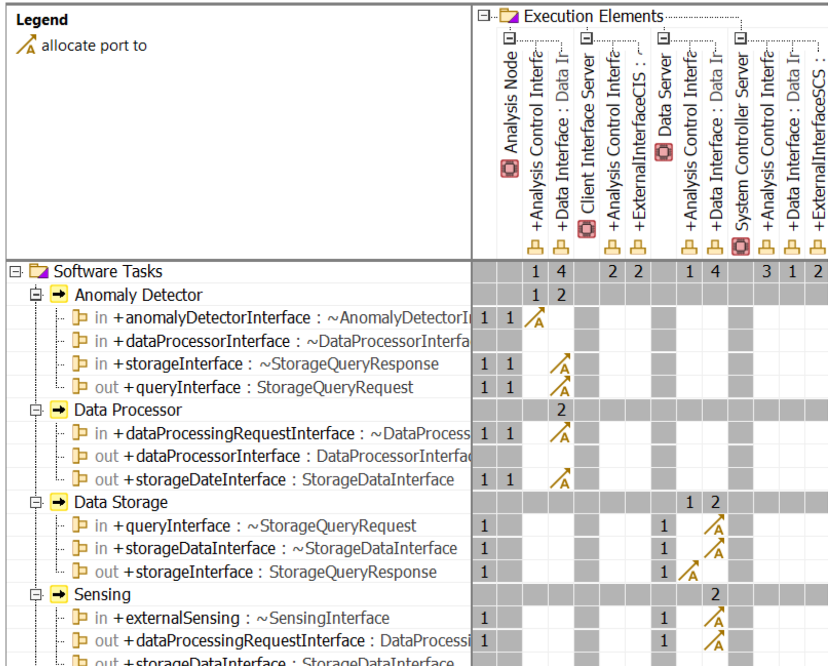


Figure 8-63: Mapping of task ports to ports of the execution hardware.

8.3.6.1 Evaluation Results

This case study shows that it is possible to create a technical architecture suitable for the ADS. This included a model of the software tasks that realize the logical components, as well as a hardware architecture containing execution hardware and networks. Through the deployment mappings it was possible to relate the tasks of the task architecture to the execution components. The same holds for the ports.

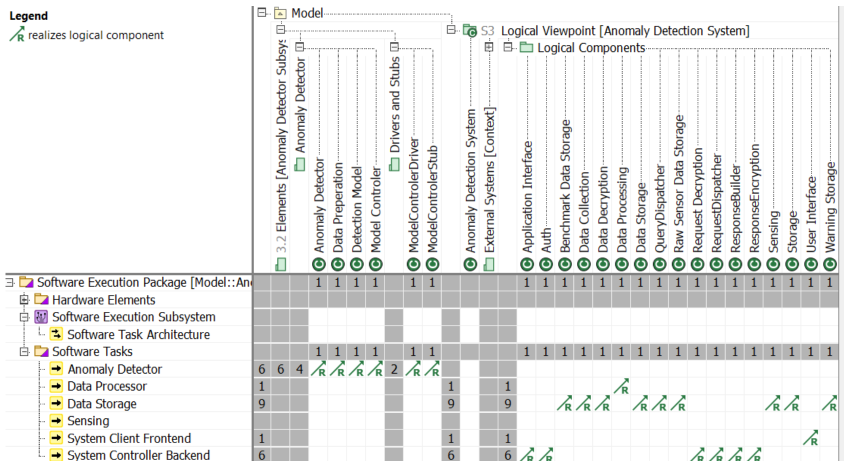


Figure 8-64: Mapping of logical components to tasks of the task architecture

8.3.7 Crosscutting Concepts

This section describes aspects that touch various views. In particular, the section gives a dedicated report on traceability topics. It also describes an impact analysis to evaluate to which degree the SpesML model eases handling changing requirements.

8.3.7.1 Traceability

This case study establishes traceability links that allow to follow from a requirement via the functional and logical view to the technical architecture. As a first step, traceability links inside the requirements view were created. Here, stakeholder requirements are traced to detailed system requirements and quality requirements are traced to derived functional requirements (see Figure 8-65 and Figure 8-66 for excerpts from these matrices).

Figure 8-65: Tracing matrix relating stakeholder and system requirements

Legend
derive

	Model	Quality Requirements [Anomaly	Application Security	Forgery Protection	Monitoring Resources	Sensor Data Loss	Sensor Data Security	Tolerance to Sensor Faults	System Requirements [Anomaly	Functional Requirements	Validity	Benchmarking Configuratio	Query Result Data Validity
Model			1	1		3	1	2					
Functional Requirements [Anomaly Detection System::Req			1	1		3	1	2					
Data Collection						1							
Sensor Data Collection													
Sensor Data Storage	1	1				derive							
Detect Anomalies										4			
Manage Benchmarks										1			
Reporting										5			
Security			1	1		2	1	2					
Anomaly Warning Validation	1	1						derive					
Authorization	2	2	derive			derive							
Encryption	3	3		derive		derive	derive						
Sensor Data Validation	1	1						derive					

Figure 8-66: Tracing matrix relating quality requirements to derived functional requirements

Furthermore, functional requirements were traced to the functions that realize these requirements. In our case, the model contains only connections of the functional requirements to the white-box function with the finest granularity that participated in realizing the requirement. Hence, every functional requirement needs to be traced to a function, and, conversely, every function should be traced to at least one requirement, otherwise either a requirement is missing, or a function is superfluous. See Figure 8-67 for the tracing matrix between requirements and functions.

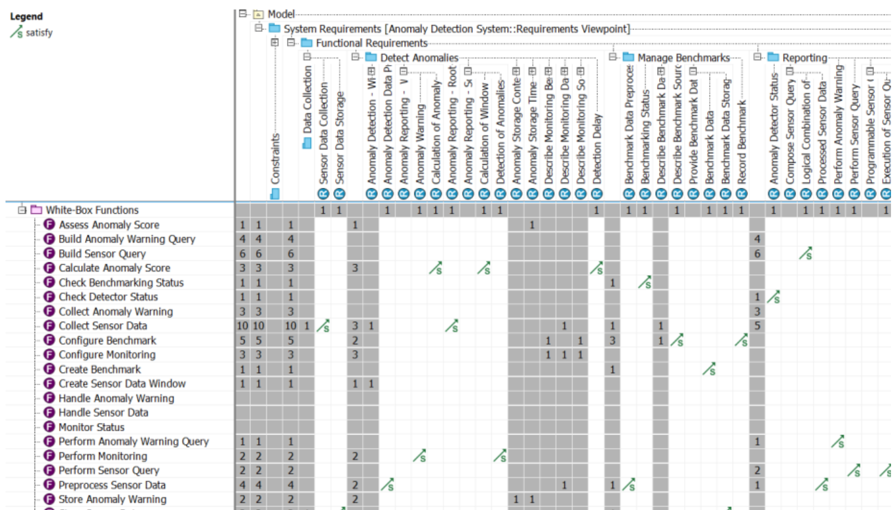


Figure 8-67: Tracing matrix between requirements and functions

The next steps in the traceability chain are the trace links from functions to logical components, as well as from logical components to technical components. In our case, all logical components represent software components (which are deployed on hardware). Hence, all logical components were traced to tasks of the technical task architecture.

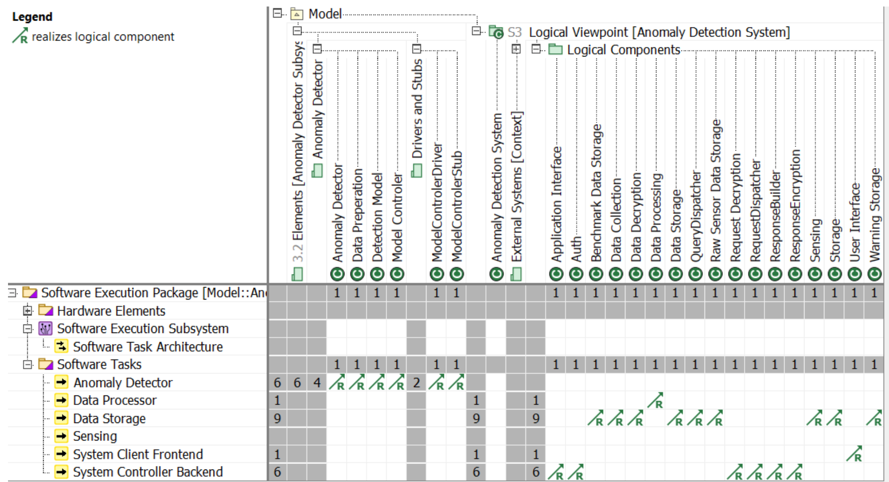


Figure 8-68: Traceability from logical components to tasks

8.3.7.2 Impact Analysis

The SpesML metamodel and the tool provide extensive possibilities for linking model elements with each other (see previous section). However, this linking is not an end in itself, but should

ease to consistently identify model parts that need to be changed in the case of changing requirements.
In order to evaluate the extent to which the SpesML method and tool support typical change scenarios, an impact analysis was carried out. Three change scenarios were used to simulate the impact analysis of a requirement change and to analyze how subsequent changes can be executed.
The three change scenarios were:
1) Deleting a stakeholder requirement
2) Modify a stakeholder requirement
3) Adding an additional stakeholder requirement.

For each scenario, a concrete example (e.g., a requirement to be removed) was used, with the understanding that the changes would be non-trivial and would include edge cases, such as a constraint that is not related to system requirements and functions.
For each of the examples, the tracing links were first used to determine which other requirements, functions and components are potentially affected by the change. For all affected elements, the consequence was then determined (for example, adaptation of an interface). The consequences can in turn lead to further changes.
The implementation of the impact analysis has clearly shown that with the help of the SpesML tooling, changes can be systematically implemented within the framework of this case study and the model can be brought back into a consistent state. However, the scenarios considered certainly do not cover all possible effects - for example, architectural changes that do not result from changes in requirements have not been considered. This would require both broader and deeper subsequent evaluations.

8.3.8 Sum-up and Overall Evaluation

This chapter describes a case study with the goal of modeling an anomaly detection system using SpesML method and tooling. Our goal was to model the system on a high level of abstraction avoiding algorithmic details and data processing rules.
As ADS has only a small portion of control behavior and a large portion of data processing behavior, it was challenging to cover this using state machines. Extended concepts involving executable functions were available in the tooling. Nevertheless, as the aim of this case study was to model the architecture and not so much the data processing and algorithms this did not impact our overall evaluation result.
The case study shows that it was possible to create a model ranging from the high-level stakeholder requirements via detailed requirements, functions, components up to the technical realization. Moreover, using the SpesML Tracing relationships it was feasible to connect related model elements. The evaluation of the SpesML model in an impact analysis shows that the methodology enables to precisely simulate changing requirements and easily identify downstream changes.

References

[46] M.Goeckler, P.Biegert, M.Grethel: Next-Generation Actuator Systems, Digital conference book 2022 (Schaeffler), https://www.schaeffler.com/en/media/dates-events/kolloquium/digital-conference-book-2022/actuator-systems/

[47] ISO 26262:2018: Road Vehicles – Functional Safety, International Organization for Standardization, 2018.

[48] ISO 11898:2015: Road Vehicles – Controller Area Network (CAN), International Organization for Standardization, 2015.

[49] CiA 301 version 4.2.0: https://www.can-cia.org/groups/specifications/

[50] MontiArc: https://www.se-rwth.de/research/Software-Architecture/

Bernhard Rumpe
Andreas Vogelsang
Sebastian Voss

9 An Outlook based on Spes, Crest, and SpesML

To support the introduction of a permanent and sustainable Model-based System Engineering approach for development, the workbench described in this book has been developed in SpesML. This workbench supports the SPES methodology based on the SpesML modeling language and uses the underlying FOCUS semantics, thus achieving a significant improvement of the SpesML model elements' semantics. SpesML thus provides a substantial clarification of the SysML modeling standard and has a clear focus on the modeling concepts that are relevant for the specification of distributed software-intensive systems. Thus, the widely used modeling language SysML has been expanded and specified by methodologically urgently needed components.

The BMBF has kindly funded the SpesML project, to develop a tool prototype that consistently supports the correct semantics of both the modeling language SpesML and the SPES methodology. The results from SpesML now form a good basis for further aspects that will be of decisive importance for productive use, especially the use of models over a life cycle.

The results are an enabler for (see also Figure 9-1):

- Modeling of advanced topics
  - Advanced analyses with tool support in the areas of consistency, security and safety, performance, or resilience
  - Joint development of cyber-physical products and their digital twins
  - Integration of AI-based components into an overall system structure
  - Derivation methods for sensor data collection based on the SpesML system models
  - Monitoring and optimization of systems at runtime (Model-Based DevOPs)
- Fostering reuse
  - Re-use of modularly developed, general-purpose models
  - Building up reusable libraries and thus making development knowledge explicit that was only implicit before
- Standardization and tooling
  - Semantically sound standards will lead to improved software tools
  - Convenience and increased efficiency in tooling

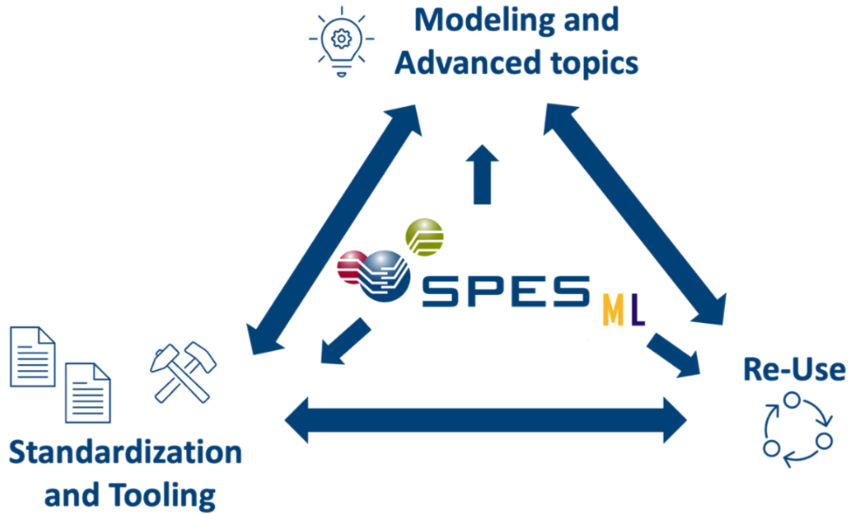


Figure 9-1: Showcase of the hardware for the WLS

9.1 Modeling and analyses of advanced topics

Model-based Systems Engineering using the SpesML workbench support in the construction of semantically rich models. To support the full lifecycle of a model, this enables future advanced analysis possibilities. To maintain the consistency of modeling artifacts, and support security, safety, and performance capabilities of the system model, SpesML provides the necessary foundation. On top of this foundation, technical and modeling capabilities can be added.
The joint, model-based development of cyber-physical products and their digital twins must become a dynamic and symbiotic relationship at the forefront of technological innovation. In such an integrated approach, physical products, such as machines or devices, are designed and manufactured in parallel with their model-based digital counterpart, namely the digital twins. A digital twin needs to know the internal structure, desired behavior, etc., and thus must be strongly based on the models of the CPS. Furthermore, it stores lakes of data acquired during operation and offers a large set of services to connect systems to systems of systems as well as for the interaction with controlling humans. Digital twins thus are complex systems themselves and therefore synergetic development in parallel is useful. Derivation methods for sensor data collections and aggregation based on the SpesML system models are a key enabler.
These techniques will allow rigid monitoring and relate the observed behavior to desired (modeled) behavior and thus many forms of optimizations of systems at runtime (Model-Based DevOps). The resulting synergy allows for a multitude of benefits, from early-stage product testing and optimization to predictive maintenance and performance monitoring. By bridging the gap between the physical and digital realms, organizations can harness the power of data analytics, machine learning, and IoT technologies to enhance product quality, streamline operations, and drive continuous improvement. The joint development of model-based cyber-physical products and their digital twins will make strong contributions to a new era of

efficiency, innovation, and agility in various industries, from manufacturing and healthcare to transportation and beyond.
Machine learning is also increasingly being used in safety-critical systems (e.g., for object recognition). Models help to describe and analyze the complex interaction of components with and without machine learning. This is the only way to release and certify safety-critical systems with machine learning components in the medium term. SpesML offers an excellent basis for this. The modeling theory can be extend-ed to include probabilistic elements that are necessary to describe ML components. The SpesML language and tools need to be adapted accordingly.

9.2 Fostering reuse

The construction of general, or in-house libraries of reusable basic components will drastically increase the efficiency of development processes, similar to reusable libraries in programming ecosystems. SpesML allows this because the language has a modular structure. It allows encapsulation and composition at all hierarchical levels and is particularly capable of reflecting the modularity of the system in the modularity of the models. This is also reflected in the SPES methodology and is better developed than, for example, in the SysML itself, which supports the composition of models only informally.

9.3 Standardization and Tooling

The SpesML prototype, with its well-developed and understood principles, is a suitable basis for realizing hardened, well-consolidated tools. General tool manufacturers should pick up these results and refine their generic tools with a SpesML profile. This is especially important for smaller companies to adopt the SpesML results as early as possible. As we have already seen in the SpesML project, individual larger companies are also able to customize their existing tool chains by specific DSLs that implement the concepts developed in SpesML.

9.4 Successful introduction of SpesML

In addition to leveraging the benefits of the SpesML modeling language and workbench, we need support and strategies to successfully implement SpesML in companies. Especially if a company does not start from scratch, it is crucial to align the MBSE implementation with the current methods used in development and the envisaged goals when introducing MBSE. With the SPES Maturity Model, we have a tool that can support the introduction.

9.5 Conclusion

Systems Engineering in high-wage countries, such as Germany or even the whole Europe, needs effective development methods to remain competitive. This is even more important as we here have built up excessive regulatory bureaucracy in legislation, but also in development. Reuse, modularity, based on a model-based approach is an essential element here and the SPES methodology and its SpesML modeling language are a very helpful contribution from our point of view, which now needs to be hardened and further industrialized.

Appendix

Partner

Foqee GmbH

Foqee GmbH was founded in 2017 as a consulting company for the definition, adaptation, improvement and application of software and systems development processes in regulated markets, such as aerospace, medical technology, pharmaceuticals and finance. The main focus is on the automotive sector with the system and software development process standards AutomotiveSPICE, ISO 26262 and ISO 21434.
Foqee's services include (1) development and rollout of process frameworks for standard-compliant software and system development, tailored to the needs of the customer organization; (2) development and rollout of tools to support process application and project monitoring; (3) support of developers in their daily project work by training and coaching, as well as with analyses, reviews and audits.
Foqee's customer base includes SMEs as well as DAX enterprises. Worldwide, more than 50 development projects are currently being carried out supported by solutions by Foqee.

fortiss GmbH

fortiss GmbH is the research institute of the Free State of Bavaria for software-intensive systems, based in Munich. The institute currently employs about 130 people who cooperate in research, development and transfer projects with universities and technology companies in Bavaria, Germany and Europe. fortiss is organized in the legal form of a non-profit limited liability company. The shareholders are the Free State of Bavaria (as majority shareholder) and the Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V..
The mission of the company according to its statutes is research and development in the field of digital technologies, in which software plays a significant role, as well as the dissemination or transfer of the resulting knowledge to interested companies, authorities, research institutions and individuals. The focus is on application-oriented research in the areas of software and systems engineering, AI engineering and IoT engineering with the goal of enabling the controllable development of a new generation of increasingly autonomous and also decentralized software systems through the integration of model-driven software development with data-driven programming of AI.

Qualicen GmbH

A Leading Company for Systems and Software Engineering with a Focus on Requirements and Test Engineering.
Qualicen GmbH is a dynamic and innovative company for systems and software engineering, specializing in the optimization of requirements and testing. Founded in 2015 as a spin-off from the renowned Technical University of Munich (TUM) Chair of Software & Systems

Engineering, under the leadership of Prof. Dr. Dr. h.c. Manfred Broy, Qualicen draws upon a strong academic foundation, combining research expertise with practical industry experience.

Connecting Industry and Research: Qualicen actively participates in industrial projects and offers tailor-made services, customized to meet the specific needs of its clients. The services provided include audits, training, process improvement, tool selection and development, as well as continuous quality assurance. This enables companies to optimize their system and software development processes and enhance the quality of their products.

Diverse Customer Base: Qualicen's customer portfolio spans various industries, including automotive companies and suppliers, aerospace and defense, as well as insurance, public sector projects and a wide range of other software manufacturers. Qualicen's cross-industry expertise allows them to develop individual solutions that address the specific requirements and challenges of each client.

Proprietary Innovative Tools: Qualicen has not only gained recognition through its first-class consulting services but also through the development of its own innovative tools that revolutionize the software development process. These include the "Qualicen Requirements Scout" and the "Qualicen Test Scout," which enable developers to systematically review and improve requirements and tests for quality. Additionally, Qualicen has developed the open-source solution "Specmate," enabling systematic and partially automated test derivation.

Continuous Advancement: Qualicen consistently stays at the forefront of technological development and invests continuously in research and development. Their close collaboration with universities and other research institutions allows them to identify current trends and best practices and integrate them into their services and products. This ensures that Qualicen's clients always benefit from the latest and most effective solutions.

RWTH Aachen University – Chair of Software Engineering

The research focus of the Chair of Software Engineering at RWTH Aachen University, headed by Prof. Bernhard Rumpe, lies in the definition and improvement of methods and development of tools for efficient software engineering. Research areas include model-based software engineering (MBSE), generative software development, and model-driven digitization of system development. The broad knowledge of this research is documented in over 200 publications and has been successfully evaluated in various industrial projects in the domains of embedded systems, autonomous vehicles, IoT, smart buildings, energy, robotics, and cloud systems. Projects included requirements elicitation, as well as function, version, and variant modeling of software and hardware architecture to support analysis and synthesis activities. In addition, the chair has a profound theoretical knowledge of modeling languages and their semantics, and with the language workbench MontiCore, a tool for the compositional development of modeling languages.

Schaeffler Technologies AG & Co. KG

The Schaeffler Group has been driving forward groundbreaking inventions and developments in the field of motion technology for over 75 years. With innovative technologies, products, and services for electric mobility, CO_2-efficient drives, chassis solutions, Industry 4.0, digitalization, and renewable energies, the company is a reliable partner for making motion more efficient, intelligent, and sustainable – over the entire life cycle. The motion technology company manufactures high-precision components and systems for drive train and chassis applications as

well as rolling and plain bearing solutions for a large number of industrial applications. The Schaeffler Group generated sales of EUR 15.8 billion in 2022. With around 84,000 employees, the Schaeffler Group is one of the world's largest family-owned companies. With more than 1,250 patent applications in 2022,
Schaeffler is Germany's fourth most innovative company according to the DPMA (German Patent and Trademark Office).

Siemens AG – Corporate Technology

Siemens is a technology group that is active in nearly all countries of the world, focusing on the areas of automation and digitalization in the process and manufacturing industries, intelligent infrastructure for buildings and distributed energy systems, smart mobility solutions for rail transport, and medical technology and digital healthcare services. Siemens comprises Siemens Aktiengesellschaft (Siemens AG), a stock corporation under the Federal laws of Germany, as the parent company, and its subsidiaries. Our Company is incorporated in Germany, with our corporate headquarters situated in Munich. As of September 30, 2022, Siemens had around 311,000 employees. As of September 30, 2022, Siemens has the following reportable segments: Digital Industries, Smart Infrastructure, Mobility and Siemens Healthineers, which together form our "Industrial Business" and Siemens Financial Services (SFS), which supports the activities of our industrial businesses and also conducts its own business with external customers.
Siemens Technology is the central R&D department of Siemens and thus has a key role to shape the future of our products. Some 2,100 employees, including 1,700 researchers and developers as well as 350 patent experts work in more than 150 research sites in Europe, America and Asia. The main Siemens Technology locations are in Germany, Austria, the US, China, Portugal and India. Siemens Technology acts as a strategic partner to support the executive units of Siemens. In consequence the main research focus is on future technologies for industry, infrastructure, mobility and healthcare. Nevertheless, Siemens Technology is well connected with partners from universities and other companies and often takes a leading role in joint funded research projects. In all our business areas we experience the current trend that the digital world moves into our real world - what e.g., used to be a solid model can now become a simulation in your VR goggles. This digital transformation brings about countless opportunities for an - attractive and sustainable - future and will dominate the research activities for the years to come.

Siemens Healthineers AG

Siemens Healthineers AG (listed in Frankfurt, Germany: SHL) pioneers breakthroughs in healthcare. For everyone. Everywhere. As a leading medical technology company headquartered in Erlangen, Germany, Siemens Healthineers and its regional companies are continuously developing their product and service portfolio, with AI-supported applications and digital offerings that play an increasingly important role in the next generation of medical technology. These new applications will enhance the company's foundation in in-vitro diagnostics, image-guided therapy, in-vivo diagnostics, and innovative cancer care. Siemens Healthineers also provides a range of services and solutions to enhance healthcare providers' ability to provide high-quality, efficient care. In fiscal 2022, which ended on September 30, 2022, Siemens Healthineers, which has approximately 69,500 employees worldwide, generated

revenue of around €21.7 billion and adjusted EBIT of almost €3.7 billion. Further information is available at www.siemens-healthineers.com.

Technical University of Munich – Chair of Software Engineering

TUM is one of the leading universities in Germany. TUM's top performances in research and education, interdisciplinary studies, and talent promotion stand out. Strong alliances with businesses and scientific institutions across the world play a part in this. TUM was one of the first "Universities of Excellence" of the nationwide Excellence Initiative and impressed this cooperative in 2006 with its concept of "TUM. The Entrepreneurial University".
The Department of Informatics attaches high importance to a close link between scientific research and study. The systems and tools developed there, are always being tested by students and research staff in a practical deployment.

University of Cologne

The University of Cologne is one of the oldest and largest universities in Germany. With its six faculties covering a broad spectrum of disciplines and its internationally outstanding research profile areas, it enjoys an excellent reputation for its academic achievements and high standard of undergraduate and graduate education. The Faculty of Mathematics and Natural Sciences with 186 professors and more than 12,000 students offers a wide range of modern study programs, that are closely coupled with its wide variety of outstanding research activities.
The chair for Software and Systems Engineering at the Institute of Computer Science, led by Prof. Dr. Andreas Vogelsang, is actively doing research in the areas and Model-based Systems Engineering, Requirements Engineering, Software Engineering & Machine Learning, Explainable Intelligent Systems, Research Software Engineering, and Data-driven Systems Engineering.
Prof. Vogelsang was a leading researcher in several BMBF-funded MBSE research projects. In the completed BMBF project "CrESt" the focus was on model-based development of interconnected, collaborative embedded systems. The BMBF project "SPEDiT" involved developing an introduction strategy for model-based development aligned with the goals of companies. The chair has significantly contributed to the SPES Modeling Framework, a globally unique, scientifically grounded modeling approach that connects all phases of Model-Based Systems Engineering.

Author Index

B

Bayha, Andreas
fortiss GmbH
Guerickestr. 25
80805 Munich, Germany

Bergemann, Sebastian
fortiss GmbH
Guerickestr. 25
80805 Munich, Germany

Böhm, Dr. Wolfgang
Technical University of Munich
Department of Informatics
Boltzmannstr. 3
85748 Garching, Germany

Broy, Prof. Dr. Dr. h.c. Manfred
Technical University of Munich
Department of Informatics
Boltzmannstr. 3
85748 Garching, Germany

D

Drux, Florian
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

F

Femmer, Prof. Dr. Henning
Qualicen GmbH
Am Berkelbogen 12
48653 Coesfeld, Germany

G

Goger, Marcel
Schaeffler Technologies AG & Co. KG
Industriestr. 1-3
91074 Herzogenaurach, Germany

Groß, Samson
Schaeffler Technologies AG & Co. KG
Industriestr. 1-3
91074 Herzogenaurach, Germany

Gupta, Rohit
Siemens AG
Foundational Technologies
Friedrich-Ludwig-Bauer-Str. 3
85748 Garching, Deutschland

H

Hocke, Dr. Ralf
Siemens Healthcare GmbH
Siemensstr. 3
91301 Forchheim, Germany

J

Junker, Dr. Maximilian
Qualicen GmbH
Am Berkelbogen 12
48653 Coesfeld, Germany

K

Koch, Dr. Walter
Schaeffler AG
Industriestr. 1-3
91074 Herzogenaurach, Germany

P

Pfeiffer, Mathias
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

Philipps, Jan
foqee GmbH
Kirchenstr. 86
81675 München, Germany

R

Regnat, Nikolaus
Siemens AG
Foundational Technologies
Friedrich-Ludwig-Bauer-Str. 3
85748 Garching, Deutschland

Rumpe, Prof. Dr. Bernhard
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

S

Schmalzing, David
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

Setzer, Dr. Stefan
Siemens Healthcare GmbH
Henri-Dunant-Str. 50
91058 Erlangen, Germany

Vogelsang, Prof. Dr. Andreas
University of Cologne
Software and Systems Engineering
Sibille-Hartmann-Str. 2-8
50969 Köln, Germany

Voss, Prof. Dr. Sebastian
FH Aachen / qwitto GmbH
Software & Systems Engineering
Eupener Str. 70
52005 Aachen, Germany