Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

METHODOLOGICAL

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

*Robert Deckers[1,2] | Patricia Lago[1]

[1]Vrije Universiteit Amsterdam, The Netherlands
[2]Atom Free IT, Heeswijk-Dinther, The Netherlands

To enable the people involved in a software development process to communicate and reason close to their area of knowledge, we are investigating and engineering a method that formalizes and integrates knowledge of multiple domains into domain models and into specifications in terms of those domain models. We follow an action research approach, starting with a diagnosis phase, in which we have previously defined a set of method objectives, and performed a systematic literature review. During action planning we defined how we are going to develop the method –called MuDForM. This paper reports on the methodical support for using a domain model as terminology to define other specifications, and feature specifications in particular. During action taking, we defined an initial version of the method and set up case studies. During the evaluation phase, we performed a case study to validate how well the method helps in the specification of processes and to realize the case-specific objectives of the customer. The case study pertains to the formalization of the ISO26262 standard for functional safety in the automotive domain. The created models are explained to the involved experts to ensure their consistency with the original text. We found that MuDForM is suitable to systematically formalize processes described in natural language, such that the resulting process models are fully expressed in terms of domain concepts and concepts from outside the domains and processes of interest. Further, during the specifying learning phase, we have extended our method with concepts, steps, and guidelines for grammatical analysis, for formalization of constraints, and for the specification of processes.

KEYWORDS:
Method engineering, Language engineering, Domain modeling, Feature modeling, Process modeling, Model based engineering, ISO26262

1 | INTRODUCTION

Since the 1980s, several methods for domain modeling have been proposed[1,2,3]. Domain models can be used in the development of a system in various ways, *e.g.*, as a basis to derive a software design[4], or as the terminology for other specifications, like a requirements specification[5] or functional specification[3].

This work introduces an integral domain-oriented modeling method, called Multi-Domain Formalization Method (MuDForM), which provides support for the creation of domain models, and for the creation of models that are defined in terms of a domain model, called *domain-based models* (see Figure 1). MuDForM provides steps, concepts, and guidelines for an analysis and modeling process, which starts with a knowledge source, like a (domain) text or (domain) expert. This paper explains the part of MuDForM for domain-based feature modeling, and its application in a case study.

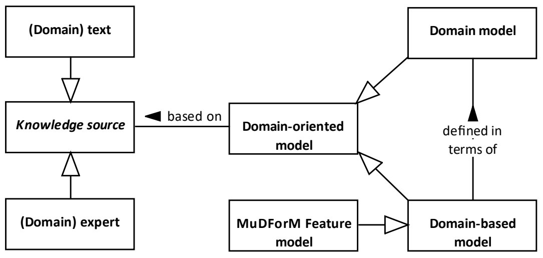


Figure 1 Context of a domain-based model (UML class diagram)

The rest of this section describes the problem, our contribution, and target audience. Section 2 explains in more detail what we try to achieve with our research, and how this paper fits. Section 2 also clarifies what we mean with the terms domain and feature. Section 3 explains the research methodology. Section 4 describes the foundation of MuDForM, which is required to understand the support for specifying features in terms of a domain model, defined in Section 5. These two sections are based on the complete MuDForM definition (metamodel, method steps, viewpoints, and guidelines)[6], of which the phase from input text to initial model is published previously[7]. Section 6 reports on a case study in which MuDForM is applied to formalize process specifications of the ISO26262 standard[8]. Section 7 reflects on MuDForM 's feature modeling support via the results from the case study. Section 8 discusses the threats to validity of this work. Section 9 discusses related work, and Section 10 concludes the paper and presents suggestions for future work.

1.1 | Problem Statement

Deckers and Lago report in a systematic literature review (SLR)[9] that methods for domain-oriented specifications are mostly limited to creating a domain specification (DS), like a domain-specific language (DSL) or domain model (DM), and do not incorporate steps and guidance for applying a created DS. There are many publications about the usage of a specific DS as well, *e.g.*, the security domain model from Firesmith[5], and the many examples from van Deursen *et al.*[10]. However, they only provide support for the DS at hand, in the form of an underlying model and a notation, and mostly not provide detailed steps and guidelines. The KISS method for Object Orientation[3] appears to be the only approach that has an explicit modeling phase and concepts for applying a domain model in functional specifications. But the KISS method does not provide a metamodel, fine-grained method steps, and detailed guidelines. **With MuDForM, we aim to offer methodical support, including steps and guidelines, for the creation of DMs, as well as for the creation of models in terms of DMs, called domain-based models.** Section 2.1 explains in more details our motivation for the work presented in this paper.

Although we do not know of literature that explicitly states the need for methodical support for applying a DS in making other specifications, we found several papers related to this topic. There is literature about evaluating the usability of created DSLs, *e.g.*, Barivšić *et al.*[11,12,13]. They state that the evaluation of DSLs does not happen often. Furthermore, Gabriel *et al.*[14] state that the community in software language engineering does not systematically report on the experimental validation of the languages it builds. So, it is not recorded if a created DSL is usable in practice. Gray *et al.*[15] write that "poor documentation and training [of a DSL]" is one of the 10 reasons why DSLs are not used more frequently in industry. Völter[16] states that it is important to communicate to users how a specific DSL works, and observes, that in non-scientific domains, domain experts, although they are the targeted modelers, are often not the ones that make models with the DSL. Instead, the domain expert pairs up with the DSL developer to apply the DSL, or the DSL developer does all the specification based on discussions with the domain expert. We see this as a symptom of that **it is not straightforward for a domain expert to make models with a DSL that is created by someone else.**

We, the authors, have observed the same phenomenon regarding DS developers and domain experts in many industry projects. A DS is defined, but only the DS developers know its exact semantics and know how to use it as a language for other specifications. The targeted users lack this knowledge, and hence, might not apply the DS correctly in making specifications with them. Without methodical knowledge about the use of a DS, they also do not know what steps to take, how to make good modeling decisions, and what information to retrieve from domain experts or from the input text, or how to organize the specification process. The lack of methodical support, in particular clear steps and guidelines, leads to an unpredictable and difficult to organize specification process, which results in specifications of which the quality is only controllable by validation, and not by design. We think this **lack of methodical guidance is one of the most important shortcomings in the literature on domain-oriented methods**, which is subscribed by Gray *et al.*[15] and Barivšić *et al.*[13].

The SLR[9] also concluded that none of the found methods on domain-oriented modeling has a complete method definition (*i.e.*, covering underlying model, notation, steps, and guidance). Some approaches provide a metamodel and a notation, and others provide high level steps, sometimes guidance, and sometimes the use of an existing language like UML. The lack of an underlying model makes it hard to have an unambiguous well-defined interpretation of models, and difficult to separate the semantics from the syntax. If there is a language defined, but no steps and guidance, then modelers must be experienced in the language. Otherwise, the specification process becomes unpredictable.

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

1.2 | Contribution and Audience

This paper has two main contributions. First, it presents methodical support for using domain models as a coherent vocabulary to make specifications, such that that support is an intrinsic part of our method, which also covers the creation of domain models. Practitioners may use the support as guidance for applying a domain model as a structured vocabulary for other specifications. The benefits of the support are the predictability and the manageability of the specification process, because the method steps (see Figure 5 and 6) form the basis for planing the specification activities. Furthermore, the provided guidelines (see Appendix A) help making decisions throughout the steps of the analysis and modeling process. The latest version of the complete MuDForM definition is available via the MuDForM Github repository[6].

Method developers may use the description of the support as an example of how to extend a domain modeling method with a part for using a domain model to create other specifications. This facilitates bridging the gap between the literature that considers a domain model as a decomposition of a product line, like in FODA[17], and the literature that considers a domain model as a structured vocabulary to define other specifications, like the security domain model from Firesmith[5] and the many examples from van Deursen *et al.*[10]. As such, this contribution might also help researchers and method developers to understand the mentioned literature gap. It must be clear that the methodical support in this paper is an integral part of MuDForM, and hence, is most suited when the domain specification that is used to create the domain-based specification, is also created with MuDForM.

As a second contribution, the paper presents the validation of the methodical support in an industrial case study, which covers both the creation of a domain model (published by Khabbaz Saberi[18]), and the creation of a domain-based feature model (the latter being the focus of this paper). Researchers may use the case study to understand the methodical support. Practitioners may use it as an example of how to create feature specifications in terms of a domain model, or, as in the case study in Section 6, process models .

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

2 | BACKGROUND: MuDForM VISION AND TERMINOLOGY

To understand the reason behind the work that is reported here, we explain what we aim to achieve with MuDForM (Section 2.1), and what we mean with the concepts *domain* (Section 2.2) and *feature* (Section 2.3)

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

2.1 | MuDForM objectives

Based on our vision and experience with domain modeling, architecture, and model driven development, we have defined a set of objectives for the development of MuDForM. We introduce them shortly to justify some characteristics of the method definition in Sections 4 and 5. The objectives are:

O1 In any system development process, there are people that have concerns about different aspects and that take decisions about different aspects. A specification method should support the **distinction of multiple domains**, and their specification in DMs or DSLs, to deal with the multitude of aspects in a development process. Moreover, a specification method should offer multiple mechanisms, *e.g.*, composition, consistency rules, transformation, or weaving, to integrate DSs, and domain-based specifications.

O2 There is no limitation to what kind of aspects can be relevant in system development. A specification method should therefore be **independent from any domain** or system, and a method user (modeler) should not need any prior knowledge about the domain or system that is being specified. In practice, domain modeling is mostly used for the application domains of targeted systems, or for design aspects of software. We think domain modeling should also be applicable to quality domains, such as reliability, security, usability, or functional safety, like in the case study of Section 6.

O3 The knowledge of people about particular aspects can be seen independently from any specific (software) system, and is potentially usable in multiple systems. A specification method should reflect this and support **self-contained specifications** that are independent from their application in a specific system specification. To use specifications in different contexts, *i.e.*, to build other specifications, they should be composable, interpretable, and translatable.

O4 In our notion of domain, a DM captures what can happen and what can exist in a domain, and a system specification or a feature specification are about what shall happen and what shall exist in a system and its context. A specification method should support this distinction, *i.e.*, **distinguish descriptive domain specifications from prescriptive domain-based specifications**. The latter is the main topic of this paper.

O5 Most domains and systems are not only about entities with a state, but also about change. A method should therefor support both the **the specification of state** of a domain at a certain moment **and the specification of change** of state over time. In other words, specifications should address things that exist, things that happen, and how these things are related. This perspective is similar to the notion of structural (static) properties and behavioral (dynamic) properties in UML[19].

O6 Almost all people, including domain experts, use natural language to convey their knowledge and decisions. It is used in many documents in any system development process. A specification method should support the **transformation of** knowledge stated in **natural language** into specifications in an unambiguous specification language. Preferably, such specifications should themselves be translatable into natural language. The purpose of this support is to minimize loss of semantics and better mutual understanding in the communication between modelers and domain experts. The MuDForM vision is to have method concepts that are close to human cognition. Natural language is a starting point for the method concepts, because it has evolved over thousands of years to support communication between people.

O7 A specification method should be **engineered**, which means it should support specification consistency and completeness, elicitation of knowledge, and traceability of modeling decision, and have a comprehensive definition. According to Kronlöf[20] a method definition should provide:

- An underlying model, *e.g.*, metamodel, core model, or abstract syntax, which forms the foundation for the semantics of a specification.
- An explicit notation, possibly used in different viewpoints. All the viewpoints of the method should be defined in terms of the concepts of the underlying model.
- Explicit method steps that guide the modeler through the viewpoints.
- Guidance for taking steps and making specification decisions.

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

We will refer to these objectives in the explanation of MuDForM in Sections 4 and 5 when applicable. Though, not all objectives are addressed, because this paper only covers a part of MuDForM, *i.e.*, the making of prescriptive domain-based models (as stated in objective O4).

2.2 | Domain

In the literature, we found two different notions of domain model. The notion that we use is that of a specification space, analogous to a domain in the mathematical sense. The term "domain" refers to an area of knowledge or activity and a DM describes what can happen (behavior) and what can exist (state and structure) in a domain, or in other words, what can be controlled and managed in a domain. A DM is the foundation for a shared lexicon in communication between stakeholders, and can serve directly as a structured vocabulary for making other specifications, or form the underlying model of a DSL. For example, a model of the banking domain expressed in a UML class diagram, can be used directly in other UML diagrams, or can serve as the abstract syntax of a DSL. A DM is not intended to express what should happen, does happen, is likely to happen, or has always happened in the domain, because we assign those aspects to different types of specifications, like a system, application, feature, or context specification. The knowledge captured in a DM is not limited to a specific way of working in the domain nor to a specific system that operates in the domain. In general, approaches for DSLs like[21,22,23], comply with this notion of DM.

The other notion in the literature is that a domain is a collection of related systems, often forming a product line or family. Accordingly, a DM defines a set of (system) features that are common in the domain of interest. This notion is used for example by FODA[17]. According to this notion, a DM can only be made with a set of systems or features in mind. In the notion that we adhere to, one can talk about the concepts in a domain independently from any feature or product. ODM[2,24] follows this notion too, but speaks of descriptive and prescriptive aspects of a domain. We go a step further, and state that the descriptive aspect is covered by the domain concept, and the prescriptive aspect is covered by the feature concept.

2.3 | Feature

Lee *et al.*[25] identified in 2002 different meanings in literature of the term feature. ODM[24] defines it as a distinguishable characteristic of a "concept" (*e.g.*, artifact, area of knowledge, etc.) that is relevant to some stakeholders (*e.g.*, analysts, designers, and developers). MuDForM follows this definition, and covers the internal specification of features. Accordingly, a MuDForM feature model specifies the desired characteristic, *i.e.*, describes what must happen (behavior) and what must exist (state). A MuDForM feature model is defined in terms of domain models. Namely, the concepts captured in a domain model can be used as types of elements in a feature model. For example, a specific feature model describes how the traffic lights at a crossing must be positioned above driving lanes, and how they must behave, which could be used as a requirements specification of a traffic light control system. In this case, the domain model describes that you can turn the lamps in a traffic light on and off, that you can hang lights above driving lanes, and that a crossing can have several driving lanes. As such, the domain model serves as a specification space for the feature model.

As a result, our notion of feature model differs from the definition in some other literature[17], where a feature model shows the relation between several features of a product line. Often features correspond with high-level functions of a system, and as such a feature model can be seen as a functional decomposition of the product line. MuDForM uses the feature structure viewpoint, as explained in Section 5.2.1, to show the decomposition of a feature in sub functions, or sub features. Just as a MuDForM domain model is described in multiple views, a MuDForM feature model is also described in multiple views. The feature structure viewpoint is similar to what some other methods call a feature model.

3 | RESEARCH METHODOLOGY

This section describes the research methodology we have applied to gather the results presented in this paper. Given the problem statement and the above definitions of domain and feature, we aim to address the following research questions:

(RQ1) How should methodical support for making feature specifications be integrated in a method that also supports the creation of domain models? The answer is provided in Section 4, in terms of how the modeling concepts and method steps fit with MuDForM's other modeling concepts and method steps.

(RQ2) What methodical support can be given for the specification of a feature, such that it is defined in terms of domain models? The answer is provided in Section 5, in terms of modeling concepts, method steps, and guidelines for the steps.

The development of MuDForM started as a project in which experience from industry practice is captured and made tangible in a method vision and definition, followed by a phase in which the method is applied to cases and adjusted based on case findings. The approach can be categorized as action research according to the description by Petersen *et al.*[26], which we use as the basis for explaining our study, resulting in the phases of Diagnosis, Action planning, Action taking, Evaluation, and Specifying Learning. The following explains it to the extent that is relevant for the MuDForM part presented in this paper.

Diagnosis.

Based on our experience with modeling, architecture, and model driven development, we have defined a vision on software development, which is shortly mentioned in the introduction of Section 2. The vision is further refined in the form of the method objectives, defined in Section 2.1. When we started to work on MuDForM, we already knew of specification approaches from several books on domain modeling and domain-specific languages[4,3,23,27,28,22]. We observed that those books did clearly not address all the MuDForM objectives. So, we performed a systematic literature review[9] with the same objectives as starting point. We identified some clear gaps in the existing literature, which should be bridged to achieve the objectives. Especially, the use of natural language processing, dealing with multiple domain models, and the use of DMs and DSLs to make other specifications are topics that are hardly addressed in existing literature. The latter corresponds with objective O4 and is the main topic of this paper.

Action planning.

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

We have chosen UML for the representation of the metamodel and method flow. We also considered using metamodels like Ecore[29], MOF[30], and GOPPR[31]. The advantage that these are more strictly and precisely defined than UML, did not outweigh the familiarity of UML and the availability of good tooling. Moreover, we only use a small subset of UML's modeling concepts in the MuDForM metamodel and method flow. For the sake of readability, we have chosen to use natural language for the specification of the guidelines. Though, we have expressed the guidelines in terms of the MuDForM metamodel as much as possible.

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

Action taking.

We defined the initial versions of the metamodel, method flow, and guidelines. The metamodel and method flow are the result of our 25 years of experience in creating, using, and managing domain models, mainly with UML[19]. We have created and used domain models in the context of domain analysis, requirements engineering, functional design, software architecture, process modeling, and test specification, in various business domains. We have started to record and generalize our experiences, and work them out in detail since we started the MuDForM research program in 2015. We actively manage the metamodel[1] and the method flow in an UML model with the tool Enterprise Architect[32].

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

Evaluation.

Together with experts from the customer, we applied MuDForM to a subset of the ISO26262 standard[8], which led to the results described in Section 6. Typically, each analysis or modeling step started with an explanation of the relevant viewpoints and guidelines to the involved experts. We recorded examples of major analysis and modeling decisions, which included the identification of used guidelines, and the creation of new guidelines. We did not record all decisions for the sake of manageability and project speed. Namely, the industrial partner had constraints regarding the duration and involvement of personnel. Logging all decisions would have decreased the process pace too much. During the case study, we also refined the method steps and their descriptions, and came up with new viewpoints and guidelines, partially based on feedback of the involved people. Moreover, the recorded analysis and modeling decisions were regularly discussed and (re)specified. We also asked people to review the method definition, i.e., the metamodel, method flow, and guidelines, on which we reflect in Section 7.1.

After the modeling process, the recorded model was presented and explained to the employees of the industrial partner. This involved verbalizing the model in natural language, which makes it easy to check if the model corresponds with the original text. We were not actively involved in the usage of the model at the industrial partner's site. However, we received feedback on the usage of the model, which is discussed in Section 7.5 and verified by the industrial partner. We reflected on the case study from the perspective of the research questions and related work on feature modeling, which is addressed in Sections 7.1 through 7.4.

Specifying Learning.

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

[1]How the metamodel is constructed is a research in itself, and topic of a future paper. The latest version is available though[6].

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

4 | MUDFORM FOUNDATION

This section describes the foundation of MuDForM (Multi-Domain Formalization Method), which forms the framework for the feature modeling part described in Section 5.

MuDForM is based on the KISS method for Object Orientation [3]. The major extensions that this paper covers are the guidance for grammatical analysis for features, the guidance for identification and specification of elements in feature models, the method steps for feature modeling, and extra viewpoints for feature models.

MuDForM is defined according to the guidelines of Kronlöf as explained under objective O7 in Section 2.1. This has resulted in a method definition [6] with the following ingredients: (i) a metamodel containing classes, attributes, associations, specializations, and constraints, which define the modeling concepts and their relations, and (ii) a method flow containing steps, guidelines, and viewpoints, which guide the modeling process. Furthermore, the case study (Section 6) uses UML [19] syntax and semantics to the extent that it fits with the MuDForM metamodel, in order to benefit from its familiar notation and available tool support. An explanation of the diagrams' syntax is given in cases that deviate from standard UML notation.

Section 4.1 explains the overall MuDForM modeling process. The method part about the phase from text to initial model is previously published [7]. Section 4.2 explains how features models relate to other parts of a MuDForM compliant model. Section 4.3 concludes the MuDForM foundation with an introduction to the different types of specification elements that MuDForM offers for the specification of features.

4.1 | MuDForM Modeling Process

Figure 2a shows the high-level steps of the MuDForM modeling process[2]. Feature modeling is present in all four steps, and is explained in detail in Section 5. The steps to create a MuDForM model are:

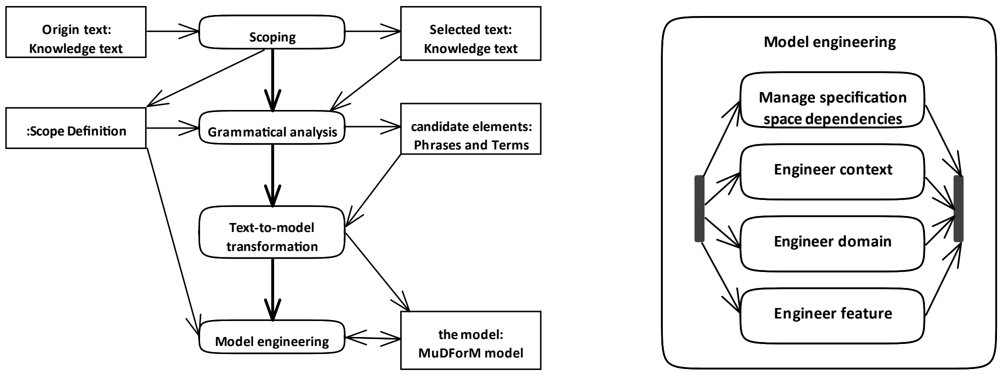
1. **Scoping**: the scope of the targeted model is specified by defining the purpose, the boundaries, and the input text that is selected from the knowledge source. The knowledge source is often an existing document, or a document that is created from interviews with (domain) experts. For each piece of text, a domain expert is appointed to provide missing information and assist with inconsistencies. The goal of scoping is to have relevant input for the modeling process, in order to (i) prevent unnecessary modeling work, (ii) detect other relevant input, and (iii) keep the model and the modeling process manageable.

2. **Grammatical analysis**: the input text is analyzed and transformed into a set of phrases with terms that are candidate elements for the model. The goal of this step is to maximize the knowledge elicitation from the source, and to make the resulting model traceable to the input, which supports objective O7. This method step itself supports the realization of objective O6.

3. **Text-to-model transformation**: the specification spaces, which form the top-level structure of a model (see Section 4.2), are identified, and the phrases are transformed into pieces of model, which each are allocated to one of the identified specification spaces. This transformation is the transition from working with text to working with models, and supports the traceability aspect mentioned in objective O7.

4. **Model engineering**: the model is completed and inconsistencies are solved by following the method steps and guidelines, and iterating over the different views. Domain experts are involved to answer questions and decide about missing concepts and model conflicts. The goal is to acquire an unambiguous specification that meets the MuDForM objectives. Model engineering consists of a step to **manage the dependencies between the specification spaces**, and three steps for **engineering** the different types of specification spaces, *i.e.*, **contexts, domains, and features** (see Figure 2b). Section 5.2 explains the feature engineering part of model engineering. There is no restriction to the order of the engineering steps. But typically, the focus is on engineering a domain or feature, and capture in a context the needed concepts that are not in the domain or feature of interest. When the focus is on engineering a feature then existing domain models and feature models might be used to specify the behavior of the functions in the feature, which also validates to used models. Often during feature engineering, new domain model elements and context model elements are identified. They have to be added to the domain or context models, to assure that all the concepts used in the feature model are properly defined. Otherwise, the feature model would be inconsistent and ambiguous.

[2]For readability, the begin- and end-nodes are omitted from the activity diagrams that depict the method flow in Sections 4 and 5. The begin step is the step without incoming control flow. The end step has no outgoing control flow.


Origin text: Knowledge text
Scoping
Selected text: Knowledge text
:Scope Definition
Grammatical analysis
candidate elements: Phrases and Terms
Text-to-model transformation
Model engineering
the model: MuDForM model
Model engineering
Manage specification space dependencies
Engineer context
Engineer domain
Engineer feature

(a) MuDForM outline **(b)** Model engineering

Figure 2 Generic steps of MuDForM modeling process (UML activity diagram)

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

When capturing the knowledge from a text in the model is the focus, the first three steps do cover feature, domain, and context simultaneously. Feature modeling becomes a separate step when the feature-specific viewpoints are made, which is in the Model engineering phase, and just before that when the initial models are created at the end of the Text-to-model transformation (see Figure 5).

4.2 | MuDForM model structure

The top-level structure of a MuDForM model consists of a composition structure of specification spaces as depicted by the MuDForM metamodel fragment in Figure 3. A specification space contains specification elements, which can be a specification space of the same type, or one of the subclasses of specification element. Each type of specification space has its own specific types of elements, which will be listed in Section 4.3. MuDForM uses specification spaces (similar to UML packages) as containers for the specification elements that make up domains, features, or contexts. The separation of models in separate specification spaces supports objective O1. A specification space may depend on other specification spaces with the following constraints:

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

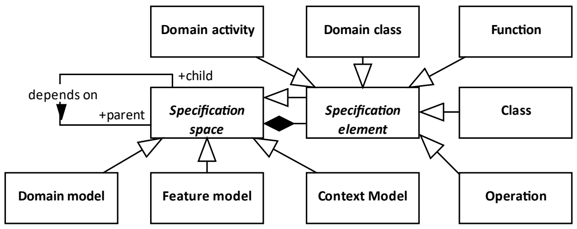


Figure 3 Specification spaces (UML class diagram)

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

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

4.3 | MuDForM Specification Elements

MuDForM offers different types of specification elements, as depicted in Figure 3. The type of specification space, *i.e.*, domain, feature, or context, determines which types of specification elements are allowed, and what is their semantics. All three different specification spaces have concepts to specify state, concepts to specify change, and concepts to specify the relation between state and change, which supports the realization of objective O5. Besides the concepts that are specific for a type of specification space, almost all specification elements can have attributes and specializations, and constraints attached to them. We list the different specification concepts below and clarify them with examples from a made up `banking domain`.

Domain models contain the following types of concepts:

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

Feature models contain the following types of concepts (see Figure 4):

- Functions are specification elements and behavior elements. They specify what must happen when the function is active. A feature is also a function, *i.e.*, the top-level function of a feature model. Some functions can be activated from outside the feature, and some functions are called by other functions. Examples: `Withdraw cash at an ATM, Transfer money on your phone, Open an account at the bank (office)`.
- A function can use other behavior elements, which can be other functions, domain activities, and operations. The usage of a behavior element in a function structure is called a function sub-behavior. Some sub-behaviors are a function event, *i.e.*, it is generated by the function or the function can react to it. Typically, one tree view is created with all the sub-behaviors of all functions of the feature. It has the feature as the root and is called the feature structure (see Section 5.2.1). Examples: the function `Transfer money on your phone` uses the domain activity `to Transfer money from an Account to an Account` and uses the function `Select an account in your banking app`.
- Functions can have attributes, which have a (context or domain) class as a type. Typically, one view is created with all the function attributes and all the function events, which is called the function signature, because it depicts how the function is seen from outside. Examples: the function `Withdraw cash at an ATM` has an attribute `withdraw amount` with type `Money amount`, an attribute `the ATM` of type `Cash register`, and an attribute `used card` of type `Bankcard`.
- Function lifecycles describe what shall happen when a function is instantiated. This so called the control flow is specified in a process algebra style[33], *i.e.*, in terms of sequences, selections, concurrency, and iterations of function steps. Function steps are typed by a function sub-behavior. For each function step is specified which function attributes are participating in it. Examples: the function `Withdraw cash`

at an ATM has functions the steps verify card, enter amount, validate amount for account, and then reject withdraw, or dispense cash.

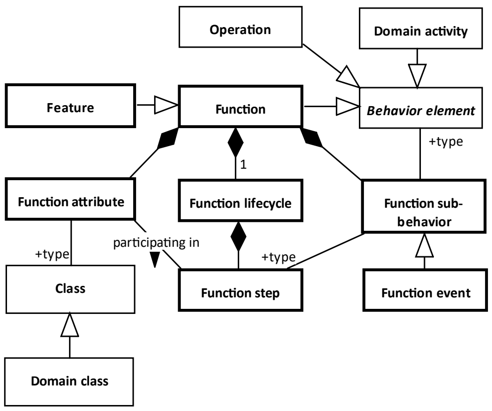


Figure 4 Feature modeling concepts (thick edge) related to other modeling concepts (thin edge) (UML class diagram)

Context models contain the following types of concepts:

- Classes are types that define possible values (without a changing state). Examples: the class Money amount, which has a an attribute currency and an attribute amount, *i.e.*, a positive number with two decimals, or the class IBAN, which represents the European wide definition of bank account numbers.

- Operations are types that define possible manipulations of values or comparisons between values. Examples: the operation Calculate interest on an account, or the operation Check if a string is a valid IBAN.

Context models typically contain two categories of concepts. First, physical quantities like length, time, power, speed and their operations. Second, concepts whose definition is not determined by the owners the domains and features of interest, like name, address, phone number, or an operation to determine the postal code of an address. These concepts might be needed to specify elements in domains and features, but their life (state changes) is not interesting. By explicitly defining needed concepts in a context model, the specifications of domains and features have no implicit semantics, which supports the realization of objective O3.

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

5 | FEATURE MODELING

This section presents the parts of MuDForM that pertain to feature modeling. Following the outline from Section 4.1, Section 5.1 explains the method steps scoping, grammatical analysis, and the transformation from text to the initial model. Section 5.2 explains the method steps of feature engineering, which is part of the model engineering step depicted in Figure 2b.

As explained in Section 2.1, guidelines are defined to help making analysis decisions and modeling decisions. Appendix A presents the guidelines that are relevant for making feature models. Each guideline has a name, a description, and refers to the method steps in which it is applicable. The guidelines are ordered by the method steps.

Figure 5 details the method steps for each of the major steps presented in Figure 2a. For the sake of clarity, we annotated the steps in the figure with notes containing references to the section in this paper. The numbers after MD indicate the section in which the step is explained. (The numbers after CS indicate the corresponding sections of the case study.)

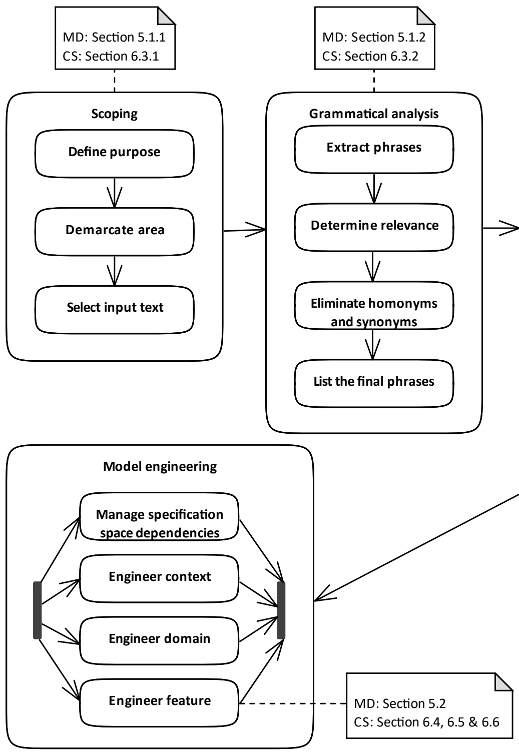


Figure 5 MuDForM method flow (UML activity diagram, MD: method definition, CS: case study)

5.1 | From Modeling Initiation to Initial Model

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

5.1.1 | Scoping

The steps of scoping are:

1. **Define purpose**: specify who the user/customer of the resulting specification is, and what they want to do with it.
2. **Demarcate area**: Name concepts that are in scope, and concepts that are out of scope.
3. **Select input text**: explicitly state which pieces of text (from a document) are the starting point for the specification process. A text can be the result of an interview with a (domain) expert. Typically, an expert is involved for each selected piece of text, to answer questions that arise during analysis and modeling.

5.1.2 | Grammatical analysis

The steps of grammatical analysis are:

1. **Extract phrases** from the selected input text and format them according to one of the phrase types: interaction structure phrase, static structure phrase, state structure phrase, and conditional phrase. (The explanation of these terms is out of scope for this paper. Table 1 in Section 6.3.2 shows some examples.) Typically, each sentence from the input text leads to one or more extracted phrases. The extracted phrases form a decomposition of the original sentence, and are processed in the next method steps, in which they can change in terminology or structure due to modeling decisions. Based on those decision, it is possible to rewrite the original sentence at any time during the modeling process, in order to check if the model still expresses the initially intended meaning. When no explicit input text is used, then sentences can be elicited directly from domain experts.

2. **Determine the relevance** of each extracted phrase from the perspective of the defined scope. Discard phrases that do not fit the scope definition, or that are duplicates.
3. Check all phrases for **homonyms and synonyms**, and **eliminate** them in consultation with the domain experts to assure that all terms (words) have exactly one meaning, and that all relevant meanings are covered by exactly one term.
4. This results in a **list of final phrases** which is used as input for the model. Form the list of phrases for the initial model via these criteria: all extracted phrases that are marked as relevant and not discarded, all newly added and rewritten phrases, and replacement of the possible homonyms and synonyms with the chosen term.

The KISS method for Object Orientation [3], which is the starting point for the grammatical analysis in MuDForM, provides a more detailed description of this phase in the modeling process.

5.1.3 | Text-to-model transformation

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

1. **Identify candidates**: Determine the terms in the phrases, *i.e.*, nouns, verbs, adjectives, and adverbs, that are a potential specification element for the model engineering step.
2. **Classify candidates**: Select which type of element each identified term is. The possible types are: domain class, domain activity, context class, function, attribute, domain, feature, context, operation, condition, function event, function step, activity operation, class relation, or specialization.
3. **Identify specification spaces**: Identify contexts, domains, and features. Choose spaces for specification elements that are coherent. Each specification space should have an owner who is responsible for its content.
4. **Create initial specification spaces view**: create a view (*e.g.*, a diagram) with all the specification spaces. Create relations (dependencies or compositions) between spaces if they are expected, or already known, complying with to the rules specified in Section 4.2.
5. **Declare and allocate elements**: place each specification element in the most logical specification space. An element can be reallocated during model engineering.
6. **Create initial models**: create a first version of the models in the specification spaces from the list of final phrases. For a feature model, this means creating the initial feature structure and initial function signatures. (The other specification spaces are not the topic of this paper.) In the case that there is no input text, and hence no grammatical analysis, this is the starting point of the modeling process.

5.2 | Engineer Feature

During model engineering, the initial models are iteratively transformed into engineered domain models, context models, and feature models, as depicted in Figure 2b. The two main modeling principles are 1) keeping the views consistent, and 2) acquire information from experts or documents to achieve a complete specification. This section zooms in on the step **Engineer feature**. Each feature engineering step corresponds with a specific viewpoint. The rest of this section explains those steps. A feature is engineered by working in parallel on the **feature structure, function lifecycles, and function signatures**, as depicted in Figure 6.

5.2.1 | Specify feature structure

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

5.2.2 | Specify function lifecycle

During this step, the control flow of each function is specified. This means describing the order in which the function steps must be executed. The function steps refer to sub-behaviors of the function (see Figure 4). All the sub-behaviors of the functions must occur at least once as a function step. The sub-behaviors are references to domain activities, operations, or functions, which must be declared in one of specification spaces that

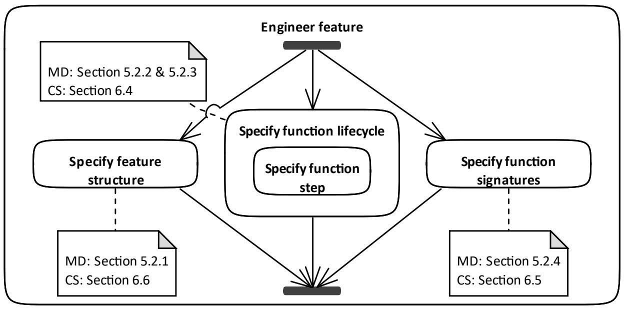


Figure 6 The steps of Engineer feature (UML activity diagram, MD: method definition, CS: case study)

the containing feature depends on. MuDForM distinguishes different types of ordering: sequences, selections, concurrency, or iterations. There are typically two ways to reason about the lifecycle. The first way is to start with the main input attributes of the function and decide which activities should be performed on them going forwards in time. The second way is to start with the end of the function in mind, which is a postcondition or a domain activity that has to be performed, and then reason backwards about the order of the steps.

5.2.3 | Specify function step

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

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

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

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

5.2.4 | Specify function signatures

A function signature describes the interface of each function in terms of attributes and events, and frames the behavior of the function. Attributes are typed by a (domain) class, and events are typed by a behavioral element, as depicted in Figure 4.

All function signatures can be put in a single diagram (see for example Figure 17), or a separate diagram is created for important and complex functions. Each attribute can be an input, output, or local attribute. Local attributes, which are used to pass on data between function steps, could be omitted from the signatures, because they are not visible outside the function. But then, a different view would have be created for the declaration of the local attributes. So normally, we put them in the same view.

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

6 | A CASE STUDY: MODELING THE PROCESSES OF ISO26262

This section presents the results from a case study in which we applied MuDForM to model the ISO26262 standard for functional safety together with automotive engineers from research and innovation institute TNO[3]. The ISO26262 standard was chosen as our case for the following reasons:

- It is mature and comes with an explicit glossary of definitions, which are a good starting point for analysis.
- It is clearly structured according to a set of processes, which are described in several documents (called parts).
- It is large and covers many aspects, which makes it relevant as a case for validating MuDForM.
- TNO Automotive has a need for an unambiguous, comprehensible specification of the ISO26262, which will be explained in Section 6.1.
- Functional safety is an example of a quality domain, which is specifically the target of MuDForM, as stated in Objective O2 in Section 2.1.

The goal of the case study is to evaluate the MuDForM support for specifying features in terms of domain models. This means to use the domain activities and domain classes of the domain model (published by Khabbaz Saberi[18]), as the types of function steps and function attributes respectively. The focus of the case study is not to validate how suited the resulting model is for a specific application.

Section 6.1 introduces the case. Section 6.2 gives an overview of the part of the case study that is the focus in this paper, and explains its execution. Sections 6.3 through 6.6 present the resulting model and elaborate on the modeling decisions by explaining how the guidelines from Appendix A are applied. The presented diagrams use a notation that is compliant with the UML metamodel[19], and are made with the tool Enterprise Architect[32]. We have added explanations for non-standard uses of the UML notation in the rest of this section.

6.1 | Introduction to the Case

TNO Automotive is a research and innovation organization that develops new concepts and new approaches for the development of automotive systems for industry partners. The ISO26262 standard for functional safety in automotive systems prescribes the processes that must be executed, and the work products that must be produced to prove the absence of unreasonable risk in safety-critical systems in the automotive domain. The specification of ISO26262 consists of 10 separate documents, called parts, which add up to 470 pages. Each part consists of several clauses. The case study did not involve the entire standard, as it would require involvement of many experts over a wide range of the automotive supply chain. Section 6.3.1 specifies which part of the standard's text is selected for the case study.

In the automotive sector, there is an increase in the complexity of the systems, in the communication between these systems, and in the amount of safety-critical functionalities. Accordingly, the work needed to achieve functional safety and its certification, is becoming increasingly time consuming and prone to human error. TNO and some of their customers, based on their own experience, expressed the need for a more predictable and uniform process. They find that a more objective certification process will help reducing the risk of human errors during the system development process, decreasin the certification costs, and increasing the safety of automotive systems[18]. Moreover, they find that the use of a controlled language can help to achieve this[34].

In the development of automotive systems, people from different engineering disciplines and several companies cooperate very intensively. As explained by Khabbaz Saberi[18], these people often have different interpretations of the standard's text, because the terminology and phrasing are not always consistent or completely unambiguous, and there are assumptions in the standard about the meaning of terms that are not an intrinsic part of the standard, *e.g.*, the terms pertaining system design like system, element, and function. A MuDForM model that covers the domain concepts and the work processes of the standard, could be the basis for a shared understanding.

TNO has a research program in integrated vehicle safety[35], and offers services to automotive suppliers for acquiring the required ISO26262 certification for their products. Therefore, TNO embraces that the people involved in functional safety analysis and system design gain a **consistent and unambiguous understanding of the activities and artifacts** prescribed by the ISO26262 standard. Parts of the domain model and the process specifications resulting from this case study are explained in depth by Khabbaz Saberi[18]. That manuscript explains the reasons why TNO needs a domain model of the ISO26262 standard.

6.2 | Case study overview and execution

The case study was executed as a collaboration between a MuDForM researcher, several TNO Automotive engineers, and an ISO26262 committee member supporting the unraveling of unclarities in the standard's text. The case study was performed between January 2018 and January 2020.

[3]https://www.tno.nl/en/

The case study is following the MuDForM method flow depicted in Figure 5 (cf. page 11), and the feature engineering flow depicted in Figure 6 (cf. page 13). During the modeling process, the most important decisions were recorded, and some of them are used in the explanation of the modeling results throughout this section. The complete model is not publicly available due to intellectual property rights. This paper shows examples of the resulting model to illustrate how the method is applied.

A challenge for this case study is how to model the explicitly stated requirements from the text in the standard. Section 6.3.2 and 6.4 showcase how such a requirement is treated by MuDForM, and how it is captured in the model.

Section 6.3 presents the phase from modeling initiation to the initial model as described in Section 5.1. It is limited to feature modeling as much as possible. We give some examples of how the grammatical analysis looks like and refer to the guidelines for some of the analysis decisions.

After that, we present the step Engineer feature by following the steps of Figure 6. Section 6.4 presents the modeling of function lifecycles for some of the ISO26262 processes, including the details of specifying function steps inside a function. This section concludes with the final function signatures and the final feature structure in Sections 6.5 and 6.6, respectively.

6.3 | From Modeling Initiation to Initial Model

This section presents the steps and the results from the case initiation to the first version of the model: Scoping, Grammatical analysis, and Transformation from text to model.

6.3.1 | Scoping

The three steps of Scoping lead to the following results:

1. The **purpose** of the total model (including the part that is described by Khabbaz Saberi [18]) is to have an unambiguous specification of the artifacts and processes that the ISO26262 standard prescribes. Following the guideline **Different specification spaces have a different purpose**, we distinguish two specific purposes for the specification of the processes: provide safety engineers with work instructions, and provide the requirements for a tool that supports the ISO26262 processes. These two purposes are both derived from the guideline **Common feature model purposes**.

2. The **demarcation of the area** concerns the clauses for item definition and hazard analysis. Examples of concepts that are in scope are: `item`, `hazard`, `hazardous event`, and `malfunction`. System design concepts like `system`, `element`, and `function`, are out of scope, but might be needed as reference, and thus captured in a context model.

3. For this version of the model, we **select as input text** Part 3 of the ISO26262 (named the Conceptual phase) excluding the Functional Safety Concept clause. The text of Parts 1 (Vocabulary), 8 (Supporting processes), and 10 (Guidelines to the ISO 26262) to which Part 3 refers, will also be considered.

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

6.3.2 | Grammatical analysis

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

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

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

If similar safety goals are combined into a single one, in accordance with 6.4.4.1, the highest ASIL[4] shall be assigned to the combined safety goal.

[4] Automotive Safety Integrity Level

Table 1 shows phrases that we extracted from the input sentences and that are relevant for Feature modeling. The first column contains the input sentence. The second is the phrase type, as mentioned in the **Extract phrases** step of Section 5.1.2, followed by the extracted phrase. The last column explains the made analysis decisions. The table is the result of all four steps of grammatical analysis, *i.e.*, **Extract phrases, Determine relevance, Eliminate homonyms and synonyms**, and **List final phrases.**

Table 1 Selection of the grammatical analysis

Input sentence	Extracted phrase and Phrase type	Decisions
This definition serves to provide sufficient information about the item to the persons who conduct the subsequent sub-phases: "Hazard analysis and risk assessment" and "Functional safety concept"	*State structure phrase*: Item definition is a phase	"This definition" refers to "Item definition" (confirmed by the domain expert). A phase is considered to be a part the total safety lifecycle. It is not a safety concept itself, but a concept to organize the process for functional safety.
	State structure phrase: Hazard analysis and risk assessment (HARA) is a phase	The same reasoning as for Item Definition.
	State structure phrase: Functional safety concept is a phase	Functional safety concept is out of scope as stated in section 4.2. "to conduct" is out of scope, because it is about the how the document is structured.
	Interaction structure phrase: Person conducts phase	Apparently, different phases can be handled by different people. The verb "to conduct" is about the process definition domain, which is considered irrelevant.
	Interaction structure phrase: HARA follows Item definition	This phrase says something about the order of behavior. The verb "to follow" is not considered as a relevant specification element itself for the same reason as "to conduct" is not.
If similar safety goals are combined into a single one, in accordance with 6.4.4.1, the highest ASIL shall be assigned to the combined safety goal.	*Conditional phrase*: If safety goals are combined into a single safety goal, then the ASIL of the single safety goal shall be equal to the highest ASIL of the combined safety goals	There are several decisions involved: • To remove possible ambiguity the word 'then' is added after the first comma in accordance with guideline Standardize logical constructs. • The part "in accordance with 6.4.4.1" is removed according to guideline Ignore phrases about the document itself. • The word similar is considered irrelevant for the meaning. The domain expert stated that determining similarity between goals is the responsibility of the safety analyst. The requirement is not influenced by goals being more or less similar. • More phrases were extracted from the input sentence, like "To combine goals into goal". These are not analyzed and discussed here, because they were phrases for the domain model.

6.3.3 | Text-to-model transformation

This section discusses the creation of the initial models from the results of the grammatical analysis. Table 2 presents a subset of the candidate elements and their classification. It is the result of the steps Identify candidates and Classify candidates described in Section 5.1.3.

Table 2 Classification of candidates

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

To understand the relation of the `Safety Lifecycle` feature to its context, Figure 7 presents the specification spaces (represented as UML packages) and their relations. The `Safety Lifecycle` feature depends on the `Design Specification` context, which defines the concepts that are the input of a safety analysis, and depends on the `Functional Safety` domain, which defines all the concepts from the standard that are needed to define the feature. The `Functional Safety` domain has two sub domains: the `Item Definition` domain, and the `Hazard Analysis and Risk Assessment` domain. This view is the result of the steps **Identify specification spaces** and **Create initial specification spaces view**. In the step **Declare and allocate elements**, the specification spaces are populated with the identified candidates and with the phrases that involve those candidates. There are two views related to feature modeling, which are created in the step **Create initial models**: 1) the initial feature structure, and 2) the initial function signatures of the feature `Safety Lifecycle`.

From the output of the **Grammatical analysis** step, the initial feature structure of the feature `Safety Lifecycle` are created (see Figure 8). The Feature `Safety Lifecycle` has two sub functions: 1) `Item Definition`, and 2) `HARA`, which correspond to clauses in the ISO26262 text.

As mentioned in the example sentences of Section 6.3.2, HARA consists of sub activities, which can be performed by a different person. That is why we applied the guideline **Define a function for coherent behavior that will be assigned to one actor**. So, `HARA` has four sub functions: 1) `Hazard Analysis`, 2) `Hazardous Event Identification`, 3) `Risk Assessment`, and 4) `Safety Goal Determination`. Those sub functions correspond to sections in the HARA clause of the ISO26262 text. The relations have been described with a UML composition relation, because the instances of the composed activities are fully executed within the instances of the composing activities, *e.g.*, `Risk Assessment` happens within `HARA`, and `Item Definition` happens within `Safety Lifecycle`.

Figure 9 presents the initial function signatures. For the feature `Safety Lifecycle`, which is a function itself, we have applied the guideline **Define attributes for sets of strong objects**, which has led to the set attributes `all systems`, `all design objects`, and `all items`. For `HARA`, we

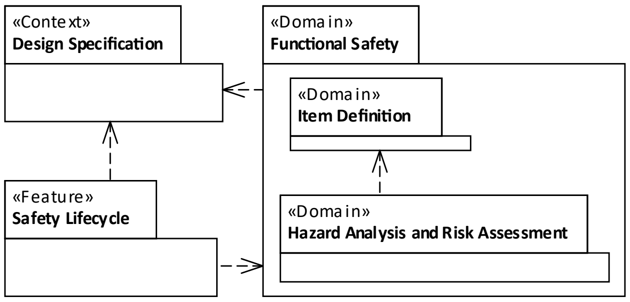


Figure 7 Specification spaces of the ISO26262 (UML package diagram)

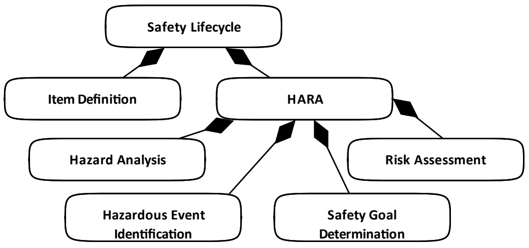


Figure 8 Initial feature structure of Safety Lifecycle (in UML notation)

have applied the guideline **Define function wide attributes for central objects**, resulting in the function wide attribute `the Item` for `HARA` and for `Item Definition`. The four sub functions of `HARA` are in the diagram too, because they are identified in the feature structure. However, there is no information to identify their attributes yet. The UML aggregation relation is used to declare the function attributes. The role names on the side of the classes indicate the names of the attributes, *e.g.*, `Item Definition.all design objects` has the type `Design Object` (from the context model `Design Specification`).

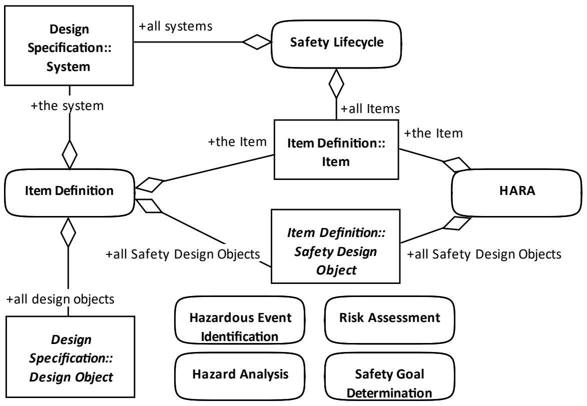

Design Specification:: System
+all systems
Safety Lifecycle
+the system
+all Items
+the Item
Item Definition:: Item
+the Item
Item Definition
HARA
+all Safety Design Objects
Item Definition:: Safety Design Object
+all Safety Design Objects
+all design objects
Design Specification:: Design Object
Hazardous Event Identification
Risk Assessment
Hazard Analysis
Safety Goal Determination

Figure 9 Initial function signatures (in UML notation)

6.4 | Specify Function Lifecycles

This section describes the creation of a subset of the function lifecycles to demonstrate the steps and guidelines explained in Section 5.2.2.

Safety Lifecycle -

The function lifecycles of the function `Safety Lifecycle` contains two steps: 1) `Item Definition` and 2) `HARA`. These steps can be executed iteratively in any order (see Figure 10) for as long the `Safety Lifecycle` function is active. We have identified two function attributes: `all Items` and `all Systems` (as already modeled in Figure 9) . These are the result of the guideline **Introduce feature attributes for sets of existing objects**, and the fact that `Item Definition` starts with a set of `Systems` that can be chosen for `Safety Analysis`, and `HARA` starts with a set of `Items` to choose from. UML control flows are used to model the order of the steps, and are represented with thick arrows. UML object flows are used to specify that a function attribute participates in a function step, and are represented with a thin arrow. The direction of the arrow indicates if the attribute is input or output for the step. The stereotype «set» indicates that the function attribute does not contain one object of the type System, but can contain multiple objects of the type System.

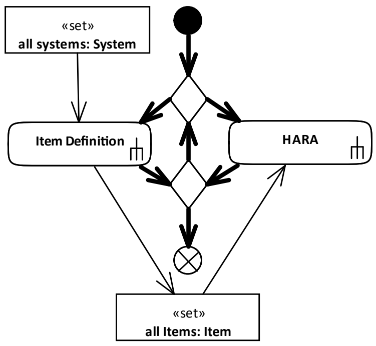


Figure 10 Function lifecycle of the function Safety lifecycle (UML activity diagram)

Item definition -

Figure 11 shows the function lifecycle of `Item Definition`. The diagram is defined by following the guideline **Go with the flow**. The guideline **Check the domain models for unused activities** is used to check whether all the referred activities from the `Item Definition` domain are defined. We will not mention these two guidelines anymore, because they are used in the creation of every function lifecycle. After specifying the control flow of the lifecycle, we specify the details of each function step.

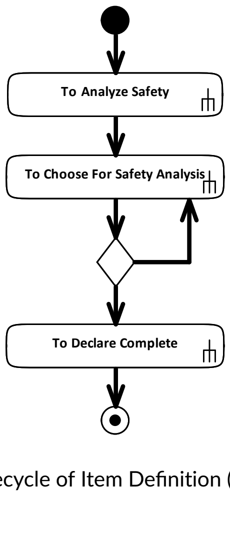


Figure 11 Function lifecycle of Item Definition (UML activity diagram)

Figure 12 shows which function attributes are participating in each step, as described in Section 5.2.3. The feature attribute `all systems` and the function attribute `all design objects` are participating in the steps `To Analyze Safety` and `To Choose for Safety Analysis` respectively. All three function steps (represented as UML actions) in the lifecycle are invocations of domain activities (defined in the domain `Item Definition`).

We gave the actions the same name as the corresponding domain activity. The attribute `the Item` is connected to all three steps. The attribute `safety design objects of the Item` is connected to the step `To Choose for Safety Analysis`. the statement between the '[..]' indicates the precondition that must hold for the instances in the attribute `all design objects`. The flow ends with the step `To Declare Complete` when the analyst determines that all needed `Design` objects are chosen for safety analysis. After this, the `Item` is available for `HARA`, and the following steps of the `Safety Lifecycle`.

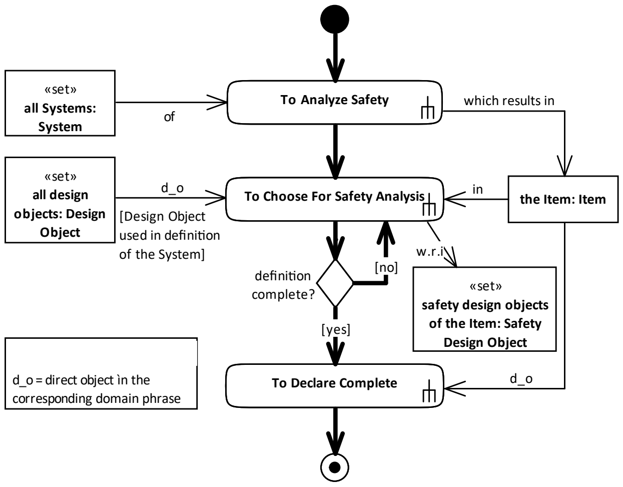

«set»
all Systems: System
of
To Analyze Safety
which results in
«set»
all design objects: Design Object
d_o
[Design Object used in definition of the System]
To Choose For Safety Analysis
in
the Item: Item
w.r.i
definition complete?
[no]
[yes]
«set»
safety design objects of the Item: Safety Design Object
d_o = direct object in the corresponding domain phrase
To Declare Complete
d_o

Figure 12 Item Definition with specified function steps (UML activity diagram)

HARA -

Figure 13 presents how the sub functions of `HARA` are ordered. The function lifecycle contains four steps, one for each sub function of `HARA`. They can be executed iteratively in any order.

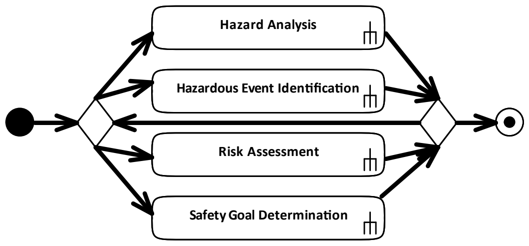


Figure 13 Function lifecycle of HARA (UML activity diagram)

The next step is to apply **Specify function step** to `HARA`. Figure 14 shows which function attributes of `HARA` are participating in each function step. To avoid too many crossing lines, which would hinder the readability of the diagram, the control flow arrows, as depicted in Figure 13, are omitted `HARA` is centered around the function attribute `the Item`. Because all sub functions are contained in the definition of `HARA`, as modeled in Figure 8, they can access the `the Item` and it does not have to be passed onto the sub functions via parameters. All the attributes in `HARA` have as a type a class from the domain model, *e.g.*, `:Item intended Function` has the type `Item Intended Function`. Each of those classes is a subclass of the domain class `Safety Design Object`. All the attributes, *i.e.*, all the involved `Safety Design Objects` must be part of `the Item`, which is depicted in Figure 17. That a `Safety Design Object` is part of an `Item` is modeled in the domain model, of which the corresponding fragment is presented in Figure 18.

For HARA, the lifecycle view has been split into two separate views, because otherwise there would too many control flow arrows and object flow arrows crossing each other, which would make the view messy. There is 1) a view with just the order of the steps and without any function

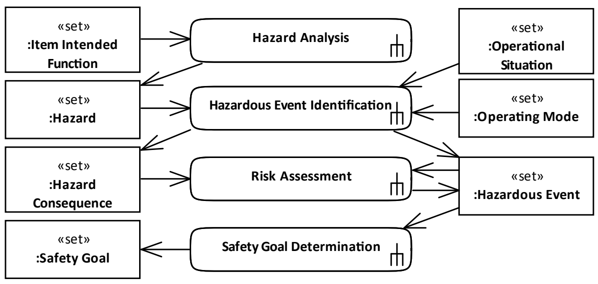

«set» :Item Intended Function
«set» :Hazard
«set» :Hazard Consequence
«set» :Safety Goal
Hazard Analysis
Hazardous Event Identification
Risk Assessment
Safety Goal Determination
«set» :Operational Situation
«set» :Operating Mode
«set» :Hazardous Event

Figure 14 HARA with function attributes connected to function steps (UML notation)

attributes related to them (Figure 13), and 2) a view, which specifies which function attributes are participating in which function step (Figure 14). Another option is to use a textual notation for the function steps, like with a regular programming language, where a function call contains the link between variables and the actual parameters of the function call.

Bookkeeping -

During the modeling of `Hazard Analysis`, `Hazardous Event Identification`, and `Safety Goal Determination`, we detected that they all required the usage of the domain activities `To Reject` and `To Combine` from the `Functional Safety` domain. These are activities to manage `Safety Analysis Objects`. Following the guideline **Define a function for recurring behavior in multiple functions**, we identified the function `Bookkeeping`, which is specified in Figure 15.

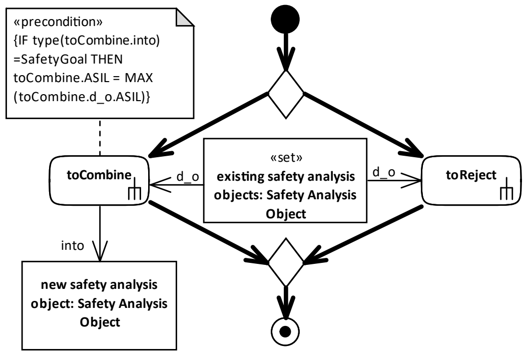


Figure 15 Function lifecycle of Bookkeeping (UML activity diagram)

Phrases that were classified as a condition need to be captured in the model as a precondition, postcondition, or invariant of a function, domain activity, or function step. The function lifecycle `Bookkeeping` contains an example of such a condition. To understand the condition, the elements from the domain model that are used in the specification of the condition, need to be clear. To understand the formal specification of the condition, we have shown the relevant part of the domain model in Figure 16. The association relations between the activity `To Combine` and `Safety Analysis Object` means that instances of that class participate in an instance of the activity. The arrow on the end of the `into`-association indicates that the activity instantiates an object of the domain class `Safety Analysis Object`. So, `To Combine` requires two or more `Safety Analysis Objects` along the `d_o` association and gives a new `Safety Analysis Object` along the `into`-association. The aggregation relation between `To Combine` and `ASIL Ranking` indicates that `To Combine` has a parameter with the name `ASIL` and the type `ASIL Ranking`.

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

- Domain class: `Safety Goal`.
- Domain activity: `To Combine`.
- Activity role: `To Combine into`.

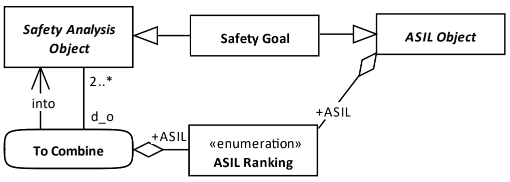


Figure 16 Fragment of domain model around to Combine (UML notation)

- Activity role: `To Combine d_o` (direct object) derived from "safety goals are combined" in the phrase.
- Context class: `ASIL`.
- Attribute: `Safety Goal.ASIL` with type `ASIL`.
- Operation: `highest` (defined on `ASIL` values). There must be a ranking of ASIL values to be able to speak of the highest ASIL. This is covered in the definition of the class `ASIL`.
- Activity attribute: `To Combine.ASIL` with type `ASIL`.

Finally, the condition has to be added to the model. This is done in two steps: 1) determine the moment in the lifecycle that the condition must hold, and 2) formalize the predicate. In this case, the position is the `To Combine` step in `Bookkeeping` (see Figure 15). The predicate is:

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

The IF clause about "type(...) = `SafetyGoal`" is added, because the domain activity `To Combine` is defined on the abstract domain class `Safety Design Objects` (see Figure 16) and not just on the domain class `Safety Goal`. The requirement about taking over the highest ASIL ranking is only valid for `Safety Goals` and not for other types of `Safety Design Objects`.

6.5 | Specify Function Signatures

Figure 17 presents the final signatures of a subset of the functions. The major changes to the initial function signatures of Figure 9 are the addition of some constraints. The invariant connected to `Item Definition` states that only the `Design Objects` that are used in the definition of `the system` may be used in the definition of `the Item`. Similarly, in `HARA`, only the `Safety Design Objects` that are part of `the Item` may be used in the sub functions of `HARA`. The semantics for these invariants come from the domain model of `Item Definition`, which is depicted in Figure 18. Another constraint is the precondition that only `Items` that have been declared complete in the `Item Definition` may be used in `HARA`.

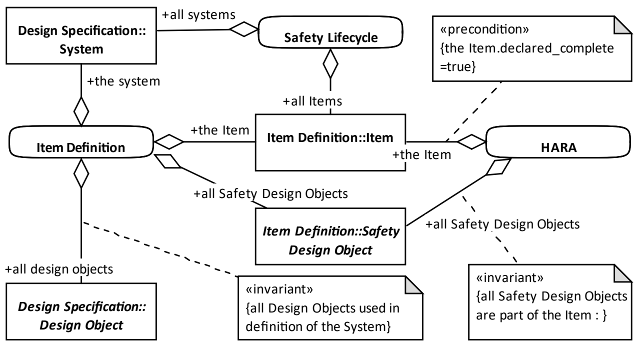

Design Specification:: System
+all systems
Safety Lifecycle
«precondition» {the Item.declared_complete =true}
+the system
+all Items
Item Definition
+the Item
Item Definition::Item
+the Item
HARA
+all Safety Design Objects
Item Definition::Safety Design Object
+all Safety Design Objects
+all design objects
Design Specification:: Design Object
«invariant» {all Design Objects used in definition of the System}
«invariant» {all Safety Design Objects are part of the Item : }

Figure 17 Final function signatures (UML notation)

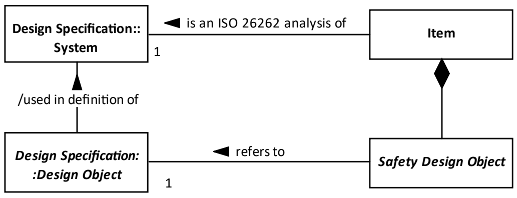

Design Specification:: System
◄ is an ISO 26262 analysis of
1
Item
/used in definition of
Design Specification: :Design Object
◄ refers to
1
Safety Design Object

Figure 18 Fragment of Item Definition domain (UML class diagram)

6.6 | Specify Feature Structure

At the end of the feature engineering phase, we revisit how the functions in the feature model relate to each other and to the context (see Figure 19). Most of the functions invoke activities from the domains that the `Safety Lifecycle` feature is dependent on, *e.g.*, `Bookkeeping` invokes `To Combine` and `Item Definition` invokes `To Analyze Safety`, which is expressed via an aggregation association. This means that the function lifecycle of those functions may contain steps that are instances of such a domain activity. The function `Safety Lifecycle`, which is the root of the structure, as well as the function `HARA`, only use other functions from the feature model.

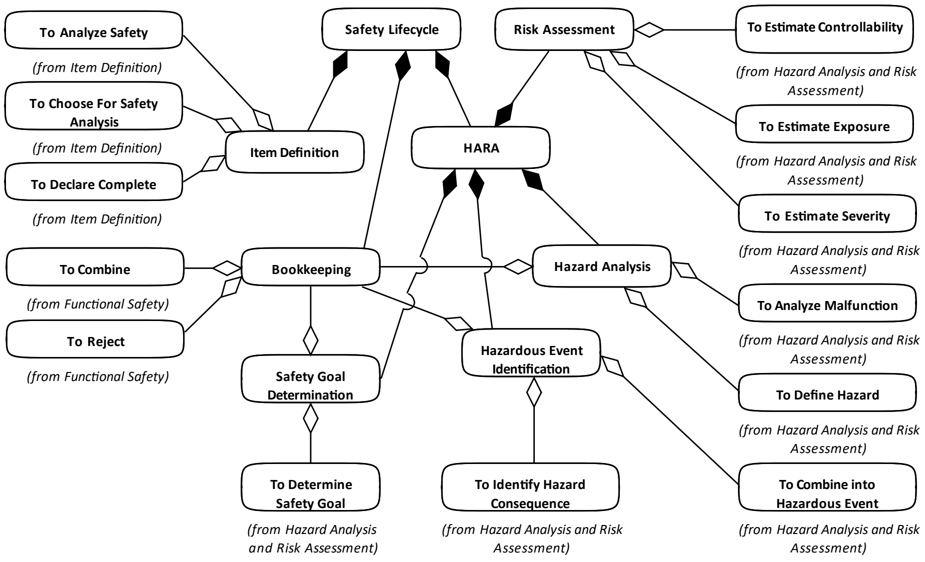

To Analyze Safety
(from Item Definition)
To Choose For Safety Analysis
(from Item Definition)
To Declare Complete
(from Item Definition)
Safety Lifecycle
Risk Assessment
To Estimate Controllability
(from Hazard Analysis and Risk Assessment)
Item Definition
HARA
To Estimate Exposure
(from Hazard Analysis and Risk Assessment)
To Estimate Severity
(from Hazard Analysis and Risk Assessment)
To Combine
(from Functional Safety)
Bookkeeping
Hazard Analysis
To Analyze Malfunction
(from Hazard Analysis and Risk Assessment)
To Reject
(from Functional Safety)
Safety Goal Determination
Hazardous Event Identification
To Define Hazard
(from Hazard Analysis and Risk Assessment)
To Determine Safety Goal
(from Hazard Analysis and Risk Assessment)
To Identify Hazard Consequence
(from Hazard Analysis and Risk Assessment)
To Combine into Hazardous Event
(from Hazard Analysis and Risk Assessment)

Figure 19 Final feature structure (UML notation)

7 | DISCUSSION

In this section, we reflect on the research questions and discuss our findings emerging from the case study. Section 7.1 discusses the support that MuDForM provides for making domain-based specifications. Section 7.2 reflects on how MuDForM helps bridge the gap between feature models (as meant in FODA[17]) and domain models. Section 7.3 discusses how feature modeling fits into the rest of MuDForM. The usability of UML for MuDForM is discussed in Section 7.4. Finally, Section 7.5 discusses how the case study helped to realize the objectives of TNO.

7.1 | Support for domain-based Specifications

This section discusses the methodical support for specifying feature models in terms of domain models (RQ2 from Section 1.2). Objective O5 of MuDForM is to support the separation of what can happen from what shall happen, *i.e.*, distinguish descriptive domain specifications from prescriptive domain-based specifications. Deckers and Lago[9] concluded that the existing domain-related literature provides little support for the methodical use of a domain specification (DM or DSL). This paper shows how MuDForM provides such support.

In our case study, we modeled clauses of part 3 of the ISO26262 as functions in the feature "Safety Lifecycle". We followed the MuDForM method steps and applied the guidelines. The results are 1) a feature structure, 2) a set of function lifecycles, and 3) a set of function signatures. The domain model elements have been applied in the specification of functions, *i.e.*, the domain activities are invoked via function steps in function lifecycles, and the domain classes are used as types of function attributes. The domain model elements are also used in the formulation of invariants and preconditions in the function signatures and function lifecycles. Moreover, the concepts of the System Design context model are used in the specification of the Item definition function. This function is the place where concepts from outside the scope of Functional Safety, *i.e.*, System and Design Object, are used to define the main concepts for the Safety lifecycle, *i.e.*, the Item and its Safety Design Objects.

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

The clearest example of ambiguity reduction that the modeling resulted in, is the identification, specification, and integration of the Bookkeeping function, which is not explicitly present in the standard's text, and the formalization of the included constraints (see Section 6.4). Moreover, if some readers find the diagrams less comprehensible than the text of the standard, then it is possible to adjust the standard's text based on the inconsistencies and gaps that were detected and solved during the modeling process.

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

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

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

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

7.2 | Bridging the Gap between Feature Trees and Domain Models

We argue that MuDForM feature modeling can be applied to bridge the gap between approaches that focus on modeling feature structures (top-down), like FODA[17] and ADOM[36], and approaches that consider domain models as a language to define other specifications (bottom-up), like the security domain model by Firesmith[5], and the many examples from van Deursen *et al.*[10]. The MuDForM part presented in this paper can unite these perspectives. The functional decomposition in the feature structure viewpoint of MuDForM is similar to the feature model in FODA, and by specifying the functions in the feature in terms of the domain model, the gap is bridged. Via the function attributes and the function lifecycles, the two specification areas are connected. The advantage for the top-down approaches is a that is enables their models to be the entry point for system development in a systematic way. Namely, the addition of function attributes and function lifecycles turns the feature structure into to a system behavior specification. The advantages for the bottom-up approaches is that the path from their models (and languages) to a system specification becomes systematic.

In addition, MuDForM feature modeling and domain modeling can be used to formalize the process models from approaches centered around process modeling languages like BPMN[37,38]. The MuDForM steps and guidelines help to organize the modeling activities. Furthermore, the use of an explicit domain model, with both domain classes and domain activities, facilitates the formalization of the process flows and process artifacts. As a result, the process models are fully expressed in well-defined terms, like in the case study of Section 6, which improves their usability for process engineering and implementation.

The steps and guidelines for feature modeling could support modeling with the Object Process Methodology (OPM)[39,40]. OPM integrates behavior concepts (processes) and state concepts (objects). It also allows composition for both types of concepts and has viewpoints similar to the feature signature and feature structure viewpoints of MuDForM. OPM has the Object Process Diagram, which is similar to the interaction view of a MuDForM domain model[18], and to the function signature viewpoint introduced in Section 5.2.4. To our knowledge, OPM does not have a viewpoint to model the internal behavioral structure of a process, like the function lifecycle of MuDForM. Nor does OPM explicitly distinguish contexts, domains, and features. Modelers using OPM could benefit from the steps and guidelines of MuDForM, and vice versa. As such, combining MuDForM OPM could increase the application scope of both approaches.

It is the concept of domain activities, and the distinction between prescriptive feature models and descriptive domain models, that enable the unambiguous specification of the process flows. Due to making feature specifications and process models domain-based, they become unambiguous and fully integrated with data (object states). In combination with explicit context models, it follows that there are no undefined terms in a feature specification. Furthermore, the domain activities and their relation with domain classes enable an easy verbalization of a model, which improves its conveyance to people who are more text oriented. For example in Figure 16, one can read the phrase "to combine several safety analysis objects into another safety analysis object". Such a verbalization is very helpful in validating the model with domain experts, who might not easily understand a graphical notation like UML.

The use of domain activities and domain classes could be an addition to BPMN related methods[37,38]. Namely, those methods do not have the literals in their models normalized, *i.e.*, they model process steps, but the steps have no explicit type with well-defined properties. For example, where in the example of Section 6.4, "to Combine" is the type of the step Bookkeeping.toCombine, in pure process-oriented methods, one could not build on the predefined feature-independent definition of something like the domain activity "to Combine". We cannot, however, fully make this claim, as we did not perform a complete literature review on the use of types and references in process modeling approaches. We plan to carry out this review after having gathered more guidelines for function and process modeling from the existing literature.

Although not demonstrated in section 6, not only the processes, but also the process artifacts can be defined completely in terms of the domain model. Each artifact can be specified as a composition of domain model elements, similar to the use of domain model elements in the constraint example of Figure 15 in Section 6.4. For example, a HARA document can be seen as a query on a repository with a database scheme that is based on the domain model. Also here, having explicit domain activities plays an important role, because they facilitate the logging of all elementary changes during a HARA execution.

7.3 | Feature modeling is part of MuDForM

This section discusses how the support for feature modeling fits into MuDForM (RQ1 from Section 3). MuDForM is based on the KISS method for Object Orientation[3], which already has support for function modeling. Differently from MuDForM, however, the KISS method does not provide support for grammatical analysis related to functions, nor does it cover the notion of (MuDForM) feature, or provide detailed steps, guidelines, and viewpoints for feature modeling.

Section 4 presents the foundation of MuDForM and how the steps and guidelines for feature modeling fit into the overall steps and guidelines of MuDForM. The partial metamodel of Figure 4 addresses minimally how the modeling concepts fit.

We observed the following regarding the method flow: already in Scoping, feature modeling is addressed, because the purpose of a feature model mostly differs from the purpose of a domain model. During the grammatical analysis, MuDForM offers guidelines for the identification of functions, function attributes, function steps, and constraints. In the transformation from text to model, there are guidelines for the identification of typical function attributes, *i.e.*, for the central objects in a feature, and for the context objects that the feature uses as input. The transformation from text to model introduces the creation of two views: 1) the feature structure, and 2) the function signatures. During Model engineering, feature engineering is a step, which is typically organized separately. Though, feature engineering might lead to changes in the domain models and context models. Namely, if a term in a feature model is not defined, then it is not the correct term and thus may not be used, or it must be integrated in a domain model or context model. This way, feature modeling validates the domain model and context model. We did not detect a point where feature modeling does not fit or is inconsistent with the rest of the MuDForM definition.

With MuDForM, a domain object, *i.e.*, instance of a domain class, can only change state through participation in a domain action, *i.e.*, instance of a domain activity. The effect is that functions do not need to take care of preserving correct object states. But, they need to take care that the objects that participate in a function step comply with the preconditions of the step and the type of the step, which might be a domain activity. The general idea is that the what-must-happen specification in the feature model may not go outside the boundaries set by the what-can-happen specification in the domain model. In other words, the domain model forms a design space for the feature model.

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

- **On the metamodel.** The metamodel has modeling concepts that are specific for feature modeling, which must be related to the concepts for domain modeling, and possibly to other modeling concepts as well, *e.g.*, analogous to the context modeling concept in MuDForM. This is not just a matter of relating concepts to each other, but also the semantics must be aligned. For example, most domain modeling methods do not have autonomous modeling concepts for specifying behavior, like the domain activity concept in MuDForM. They just model classes and attributes, and relations between classes, and often capture behavior in generic data-oriented operations like create, update, and delete. The feature modeling metamodel in this paper uses behavioral elements, *i.e.*, operations, domain activities, and functions as elementary modeling concepts (see Section 4.3). They are used as the types of the steps in a function. If the domain model only has classes, then the function steps will be a combination of create, update, and delete operations. One can solve this issue, by modeling low-level functions that serve as a surrogate for the domain activities.

- **On the notation and viewpoints.** The case study uses UML for the notation of MuDForM concepts. But MuDForM itself does not prescribe a specific notation. When feature modeling is integrated with another domain modeling method, it is possible to choose a notation that is close to the existing notation of that domain modeling method. For example, a text based notation can be chosen for the function flows, which resembles the notation of any imperative programming language like Java or Python. Of course, a notation must be defined for all the viewpoints described in Section 5.2.

- **On the method steps.** The four main steps of the MuDForM method flow (Figure 2a cf. page 8) can be generalized into: Scoping, Discovery and Elicitation (for capturing specific knowledge from a knowledge source), Switch to modeling, and Model engineering. Scoping, and Discovery and Elicitation do not require specific steps for feature modeling. However, the scope of a feature model is inherently different from the scope of a domain model. So, the knowledge that is captured from a knowledge source is different for a feature model then for a domain model. The steps Switch to modeling and Model engineering will have detailed steps that are specific for feature modeling, because feature modeling has different viewpoints then a domain model, and the modeling steps typically correspond with viewpoints. We think the steps described in Figure 6 (cf. page 13) can be used unchanged in integrations with other methods.

To put the method part of this paper in perspective of the whole method: the MuDForM metamodel contains 61 classes, the total method flow has 33 steps and sub steps, there are 12 different viewpoints, and there are currently 125 guidelines. At this moment, the method's ingredients differ in maturity and completeness. Namely, the metamodel is quite stable, and the method steps change sometimes, but guidelines are still frequently discovered, discussed, and specified more concisely. The latest version of the complete MuDForM definition is available via the MuDForM Github repository[6].

7.4 | Reflection on using UML for MuDForM

In the case study, we used UML[19] and Enterprise Architect[32] as the modeling tool. We used the standard notation and semantics as much as possible. Though, for some modeling concepts and viewpoints, we *misused* a UML concept, by giving it a MuDForM meaning. These are our findings:

- There is **no UML diagram to model the allocation** of function attributes to (actual parameters of) function steps. We used the UML concept Object Flow, but this does not cover the MuDForM semantics. Namely, a function attribute, which is a reference to an object, participates in a function step; it does not flow in or out the function step. Moreover, there is no syntax to state explicitly which parameter of the function step a function attribute is allocated to.

- **Activity diagram coordinators**, *i.e.*, fork-join and decision-merge **are not suited** for the MuDForM metamodel, which is based on process algebra. Coordinators for selection and concurrency are already offered in other methods[41,3], and they can be used. For example, the process algebra counterpart of Figure 10 is "(Item Definition | HARA)*". So, we perceive UML activity diagrams to be cumbersome for representing processes, because they are derived from the concept of state transition diagrams, which were already present in earlier UML versions. The process algebra notation does not require arrows for the loop, nor an explicit "merge" symbol, which leads to diagrams with fewer nodes and edges.

We find UML usable for a large part of the feature modeling view, but not so much for the aspects mentioned above. This is one of the reasons why we are currently collaborating with another company to build a MuDForM modeling tool in MPS (Meta Programming System)[42].

7.5 | Reflection on Case-specific Objectives

The objective of TNO, which is mentioned in Section 6.1, is realized via several MuDForM characteristics:

- The process models, *i.e.*, steps and object flows, are unambiguous now, because they are completely defined via concepts in the MuDForM metamodel.
- The processes are completely defined in terms of the domain model and the context models:
  - The process steps are expressed as invocations of domain activities, or invocations of other functions, and the modeling process has eliminated homonyms and synonyms, which makes the process steps unambiguous.
  - The objects (data) that are handled by the processes are expressed as function attributes, which are instances of domain classes or context classes.
  - The requirements in the standard are expressed as constraints, and defined in terms of the context classes, domain classes, and their attributes. They are attached to the model as a function invariant, function precondition, guard, or pre- or postcondition of a function step.

TNO also used the model of the case study to build a tool for compliance testing. This tool gives two benefits. First, The domain classes, and their attributes and relations, are used for the data structure and user interface terminology of the tool. The artifacts prescribed by the standard are corresponding with the function attributes defined in the function signatures (see Section 6.5). This encoding of the model in the tool helps unify the terminology for the safety engineers of TNO and its customers. The second benefit is that the tool provides a formalization of requirements of the standard, *e.g.*, the constraint in Section 6.4 (cf. page 21). They are implemented in the tool as validity checks[43] on the imported functional safety artifacts, and allow the tool to automatically create a (partial) ISO26262 compliance statement. This has increased the objectivity and uniformity of the certification process.

8 | THREATS TO VALIDITY

As highlighted by Petersen *et al.*[26], action research yields three main validity concerns: 1) context dependency (which is intrinsic to action research and hence*"can never be really mitigated but it can be reduced [...] when lessons learned may be transferred to similar contexts"*), 2) bias of the researcher (which can be mitigated by involving multiple researchers and/or feedback from external experts), and 3) the time factor, *i.e.*, learning and changes in the context (which can be mitigated, again, by involving multiple researchers and being aware of major changes).

Petersen *et al.* map the above threats to external validity and internal validity. Accordingly, we follow the definition of Wohlin *et al.*[44] to report the related potential validity threats that we identified, and the measures we adopted to mitigate them.

External Validity concerns the generalizability of the study. We identify two main limitations to external validity, regarding the set of guidelines we built, and the general applicability of MuDForM.

We make no claims to the general applicability or completeness of the guidelines. To mitigate this threat, we share them to be reusable and extensible so that they mature for general applicability. In fact, the guidelines are continuously adapted and extended. During modeling activities, we write down reasons for decisions when we notice them. We then let them review by others, in particular by people involved in the modeling process, and possibly formulate it as a guideline. The guidelines are linked to a modeling step, which makes it possible to only explain the guidelines to involved people at the beginning and during a specific modeling step. This ensures dosed learning of the guidelines.

The guidelines are written in terms of metamodel elements and, if necessary, complemented by commonly-used terminology from outside the metamodel. As such, we cannot ensure general-understandability. As a mitigation, however, we use common terminology that 'anybody in the field' should be able to understand. To verify this, the method steps have been applied in this and in other studies, and the guidelines are reviewed by several practitioners.

Concerning the general applicability of MuDForM, the method is designed to be domain independent (cf. objective O2 in Section 2.1). A possible limitation is that the method takes a textual description as input. As a mitigation of this potential threat, the step of extracting phrases can also be applied in interviews with domain experts, to acquire a set of usable input sentences. About domain experts, in this specific study we have involved a member of the ISO26262 standard committee to provide feedback. We are therefore confident that the modeled standard, output of the application of our method, is generally applicable to ensure standard conformity.

Finally, the study might not cover all the method parts, and thus some method parts might potentially be incorrect. More experimentation will be needed to increase method maturity. As a general mitigation, the steps have been executed in other cases too[45,18,7], and the guidelines have been reviewed by practitioners.

Internal Validity concerns the causality between the study and its outcomes[44]. We see a limitation regarding the potential bias that the direct involvement of the method developers might have introduced. For example, with respect to our observation that the used UML notation is unambiguous (Section 7.1), or that all process artifacts of the modeled ISO26262 processes can be defined completely in terms of the domain model (Section 7.2). To mitigate this threat, we regularly involved other stakeholders to provide feedback, namely ISO26262 standard experts, experts in the automotive domain, ISO26262 users (*e.g.*, for feature engineering), and functional safety tool developers.

9 | RELATED WORK

Our SLR[9] did not uncover any work on methodical support for applying a created domain specification. However, the SLR started the literature search with a search string that included the term “domain". So, it might have missed works that are not centered around the domain concept. To make sure we would not miss relevant work, we performed a search on Google Scholar on the following terms in the publication title: ((function) OR (process) OR (feature) OR (use case)). This leads to more than 100.000 hits. Adding “domain" as a mandatory term would not be helpful, as this is already covered by our SLR. For feasibility, we instead checked the top publications ordered by relevance. This resulted in a number of works related to ours, even though none of them covered methodical support for creating domain-based specifications. The following reports on them.

On process modeling. The Object Process Methodology (OPM)[39,40] also has concepts for modeling state (objects) and change (processes), and has a viewpoint to specify how those are related (object process diagram). It also allows composition for both types of concepts and has some viewpoints similar to viewpoints of MuDForM. OPM distinguishes a descriptive aspect and a prescriptive aspect in a domain, but does not provide methodical support for the specification of the prescriptive models in terms of descriptive models. (Section 7.2 elaborates on the relation between OPM and MuDForM.)

On feature modeling. Lee *et al.*[25] give guidelines for identifying features and state that a domain dictionary is required to assure uniformity across feature names. But they do not give steps and guidelines on how to use such a dictionary.

On use case modeling. Samarasinghe and Somé[46] discuss the extraction of domain models from use cases. But we could not find a paper that shows how to write use cases in terms of a domain model. A paper from Śmiałek *et al.*[47] presents an approach and metamodel to standardize sentences in use case scenarios. However, they do not show a method to do so, *i.e.*, no steps or guidelines are presented. Furthermore, they do not use a domain model, but just a vocabulary.

On functional modeling. We did not find any relevant papers that use a domain model as the vocabulary in functional specifications, except for the aforementioned KISS method[3]. But this book and other publications that refer to the KISS method, e.g.,[48,49] do not provide a metamodel, detailed steps, or guidelines.

10 | CONCLUSION AND FUTURE WORK

This paper describes the MuDForM methodical support for using a domain model as a language to define other models in general, and feature models in particular; and reports on an industrial case study in the automotive domain.

We were interested in studying the following research questions: (**RQ1**)How should methodical support for making feature specifications be integrated in a method that supports the creation of domain models?, and (**RQ2**) What methodical support can be given for the specification of a feature, such that it is defined in terms of domain models? We conclude that the current way in which MuDForM is defined, is applicable to define feature modeling too. Namely, we have defined modeling concepts, method steps, guidelines, and viewpoints, and integrated those method ingredients with the rest of the MuDForM definition. Furthermore, we conclude that it is possible to provide guidance for the creation of domain-based specifications. In doing so, we observe that the defined metamodel, and method steps are quite mature, as we did not detect relevant knowledge from the ISO26262 that we could not capture. However, more case studies would help perform a quantitative evaluation of MuDForM, to get clarity on which method ingredients are more or less mature, and which modeling decisions require better guidance. Furthermore, we would also like to have a more elaborate comparison on making domain-based specifications with and without MuDForM, which (of course) would require the involvement of modelers that have no previous knowledge of MuDForM.

The results from our study fill an important gap in the state of the art, which to the best of our knowledge lacks in providing methodical support in the first place. It lays the foundation for our future work on building a significant, validated and reusable set of guidelines for which we plan the following:

- Building a community that actively validates, identifies, and manages guidelines.
- Searching for guidelines in existing literature, and extending the list of useful publications[3,50,51,52,53,54], which were found by our literature review[9].
- Conducting a literature review to find, and analyze guidelines from process modeling approaches. We will also search for more literature on OPM, besides the already mentioned publications[39,40].

Regarding the method engineering, in our future work we plan to define criteria for deciding whether something should be a method step or a guideline. This could lead to a refactoring of MuDForM as mentioned in Section 7.1. Moreover, we are working on a paper that covers the generic principles, design patterns, design criteria with which the method is constructed.

To facilitate industrial adoption, we plan to create a MuDForM handbook for practitioners, and manage its evolution via an open platform. We are currently investigating the requirements and possibilities for a modeling tool that supports MuDForM, in order to replace the UML modeling tool that we currently use, *i.e.*, Enterprise Architect[32].

Finally, we want to mention that we, MuDForM researchers and employees of another company, are currently working on combining MuDForM and Behavior Driven Development with the Gherkin language[55,56], to create a language for model-based test specification, which involves combining the MuDForM metamodel with the Gherkin metamodel. This is an example of how to integrate MuDForM with another language. But in this case, the targeted specifications are not only based on MuDForM domain models, but also on MuDForM feature models.

ACKNOWLEDGMENTS

We like to thank Arash Khabbaz Saberi, Frank Benders, Dennis van den Brand, Jennek Geels, Arjan van Krimpen, Wan Fokkink, and Ivano Malavolta for their feedback and good discussions.

References

1. Shlaer S, Mellor SJ. An Object-Oriented Approach to Domain Analysis. *SIGSOFT Softw. Eng. Notes* 1989; 14(5): 66–77. doi: 10.1145/71633.71639
2. Simos MA. Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle. In: ; Aug 1995: 196–205.
3. Kristen G. *Object Orientation, The KISS Method, From Information Architecture to Information System*. Addison Wesley . 1994.
4. Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of software*. Addison-Wesley . 2004.
5. Firesmith D. Specifying reusable security requirements. *Journal of Object Technology* 2004; 3: 61–75.
6. Deckers R. MuDForM Method definition. tech. rep., Atom Free IT, online at https://github.com/robertdeckers/MuDForM; 2022.
7. Deckers R, Lago P. Methodical conversion of text to models: MuDForM Definition and Case Study. In: ; 2022: 113–127.
8. ISO I. 26262: Road vehicles-Functional safety. *International Standard ISO/FDIS* 2018; 26262-2.
9. Deckers R, Lago P. Systematic Literature Review of Domain-oriented Specification Techniques. *Journal of Systems and Software* 2022: 1–23. doi: 10.1016/j.jss.2022.111415
10. Deursen Av, Klint P, Visser J. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices* 2000; 35: 26–36.
11. Barišic A, Amaral V, Goulao M, Barroca B. Quality in use of DSLs: Current evaluation methods. *Proceedings of the 3rd INForum-Simpósio de Informática (INForum2011)* 2011.
12. Barišic A, Amaral V, Goulão M, Barroca B. Evaluating the usability of domain-specific languages. In: IGI Global. 2014 (pp. 2120–2141).
13. Barišić A, Amaral V, Goulão M. Usability driven DSL development with USE-ME. *Computer Languages, Systems & Structures* 2018; 51: 118–157.
14. Gabriel P, Goulão M, Amaral V. Do Software Languages Engineers Evaluate their Languages?. *arXiv preprint arXiv:1109.6794* 2011.

15. Gray J, Fisher K, Consel C, Karsai G, Mernik M, Tolvanen JP. DSLs: The Good, the Bad, and the Ugly. In: OOPSLA Companion '08. Association for Computing Machinery; 2008; New York, NY, USA: 791–794

16. Völter M. Best practices for DSLs and model-driven development. *Journal of Object Technology* 2009; 8(6): 79–102.

17. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. Feature-oriented domain analysis (FODA) feasibility study. tech. rep., Software Engineering Institute, Carnegie Mellon University; 1990.

18. Khabbaz Saberi A. *Functional Safety: A New Architectural Perspective: Model-Based Safety Engineering for Automated Driving Systems*. PhD thesis. Eindhoven University of Technology, 2020.

19. OMG . Unified Modeling Language Version 2.5.1. tech. rep., OMG; 2017.

20. Kronlöf K. *Method integration, concepts and case studies*. John Wiley and Sons . 1993.

21. Abouzahra A, Bézivin J, Didonet M, Fabro D, Jouault F. A Practical Approach to Bridging Domain Specific Languages with UML profiles. In: . 5. ; 2005.

22. Fowler M. *Domain specific languages*. Addison-Wesley Professional . 2010.

23. Kelly S, Tolvanen JP. *Domain-Specific Modeling*. IEEE Computer Society . 2008.

24. Simos M, Creps R, Klingler C, Lavine L. Software Technology for Adaptable Reliable Systems (STARS). Organization Domain Modeling (ODM) Guidebook, Version 1.0.. tech. rep., UNISYS DEFENSE SYSTEMS RESTON VA; 1995.

25. Lee K, Kang KC, Lee J. Concepts and guidelines of feature modeling for product line software engineering. In: Springer. ; 2002: 62–77.

26. Petersen K, Gencel C, Asghari N, Baca D, Betz S. Action research as a model for industry-academia collaboration in the software engineering context. In: ; 2014: 55–62.

27. Voelter M. *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*. Createspace Independent Publishing Platform . 2013.

28. Mannaerts H, Verelst J. *Normalized systems*. Koppa BvBa . 2009.

29. Steinberg D, Budinsky F, Merks E, Paternostro M. *EMF: eclipse modeling framework*. Pearson Education . 2008.

30. OMG . Meta Object Facility 2.5.1. tech. rep., OMG; 2016.

31. MetaCase . MetaEdit+ Workbench User's Guide version 4.5. tech. rep., OMG; 2022.

32. Sparx Systems . Enterprise Architect version 15.2. https://sparxsystems.com/products/ea/; 2021. Accessed: 2021-08-19.

33. Fokkink W. *Introduction to process algebra*. springer science & Business Media . 2013.

34. Luo Y, Brand v. dM, Kiburse A. Safety Case Development with SBVR-Based Controlled Language. In: Desfray P, Filipe J, Hammoudi S, Pires LF. , eds. *Model-Driven Engineering and Software Development*Springer International Publishing; 2015; Cham: 3–17.

35. Automotive T. RESEARCH ON INTEGRATED VEHICLE SAFETY. https://www.tno.nl/en/focus-areas/traffic-transport/expertise-groups/research-on-integrated-vehicle-safety/; 2021. Accessed: 23-8-2021.

36. Reinhartz-Berger I, Sturm A. Behavioral Domain Analysis — The Application-Based Domain Modeling Approach. In: Springer-Verlag; 2004: 410–424.

37. Aagesen G, Krogstie J. Analysis and design of business processes using BPMN. In: Springer. 2010 (pp. 213–235).

38. Decker G, Dijkman R, Dumas M, García-Bañuelos L. The business process modeling notation. In: Springer. 2010 (pp. 347–368).

39. Dori D. Object-process analysis: maintaining the balance between system structure and behaviour. *Journal of Logic and Computation* 1995; 5(2): 227–249.

40. Dori D, Goodman M. From object-process analysis to object-process design. *Annals of Software Engineering* 1996; 2(1): 25–50.

41. Wieringa R. Object-oriented analysis, structured analysis, and Jackson System Development. In: Citeseer. ; 1991: 1–21.

42. JetBrains . MPS, Meta Programming System. https://www.jetbrains.com/mps/; 2021. Accessed: 2021-08-19.

43. Brand DVD. Formalization of the ISO 26262 standard. Master's thesis. Eindhoven University of Technology. 2018.

44. Wohlin C, Runeson P, Höst M, Ohlsson M, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Springer Science & Business Media . 2012

45. Moghaddam FA, Deckers R, Procaccianti G, Grosso P, Lago P. A domain model for self-adaptive software systems. In: ; 2017: 16–22.

46. Samarasinghe N, Somé SS. Generating a Domain Model from a Use Case Model.. *IASSE* 2005; 278.

47. Śmiałek M, Bojarski J, Nowakowski W, Ambroziewicz A, Straszak T. Complementary use case scenario representations based on domain vocabularies. In: Springer. ; 2007: 544–558.

48. Hoppenbrouwers S, Bleeker A, Proper H. Modeling linguistically complex business domains. tech. rep., Nijmegen Institute for Information and Computing Sciences, University of Nijmegen; 2004.

49. Proper HA, Bleeker AI, Hoppenbrouwers SJBA. Object–Role Modelling as a Domain Modelling Approach. In: ; 2004: 317–328.

50. Sagar VBV, Abirami S. Conceptual modeling of natural language functional requirements. *Journal of System and Software* 2014; 88.

51. Abirami S, Shankari G, Akshaya S, Sithika M. Conceptual Modeling of Non-Functional Requirements from Natural Language Text. In: Jain LC, Behera HS, Mandal JK, Mohapatra DP., eds. *Computational Intelligence in Data Mining - Volume 3*Springer India; 2015; New Delhi: 1–11.

52. Vos v. dB, Hoppenbrouwers. J, Hoppenbrouwers S. NL Structures and Conceptual Modelling: the KISS case. In: ; 1996: 197.

53. Ibrahim M, Ahmad R. Class diagram extraction from textual requirements using natural language processing techniques. In: IEEE; 2010: 200–204.

54. Elbendak M, Vickers P, Rossiter N. Parsed use case descriptions as a basis for object-oriented class model generation. *Journal of Systems and Software* 2011; 87: 1209–1223.

55. Smart J. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster . 2014.

56. Wynne M, Hellesoy A, Tooke S. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf . 2017.

APPENDIX

A GUIDELINES PERTAINING TO FEATURE MODELING

The guidelines below pertain to feature modeling and are a subset of the complete MuDForM definition[6].

Step	**Name**	**Description**
Define purpose	Common feature model purposes	When in doubt about the purpose, check if these common purposes for feature models are applicable: • Provide the terminology for specifying other features. • Provide the terminology for specifying requirements for a system that implements the feature, like a software application, work process, or hardware. • Form the starting point for deriving specifications in another domain, typically a software domain. In other words, generate code (or models) for a specific target platform. In this case the model would typically serve as the source model for transformation rules. • Provide terminology for specifying tests to verify if a system works according to the feature model. • Provide actors with work instructions. Actors can also be (software) systems, in that case the feature model can be seen as a functional system specification.
Define purpose	Different specification spaces have a different purpose	Separate the purpose of a domain model from the purpose of a feature model. A domain model has typically a wider applicability than a feature model.
Select input text	Start with the foundation and the core concepts	When a text is too large to take in at once, then the selection can be narrowed (initially) by selecting the parts of the text that are needed for understanding other parts. This might require knowledge of the text, or at least some initial analysis to see the dependencies between parts (chapters, sections, paragraphs) of the text.
Select input text	Start small	Limit to 50 sentences for a first iteration. This helps to quickly get an initial model. After the transformation from text to an initial model, one can choose to start the model engineering, or to first add more sentences.
Extract phrases	Check if the subject is also an object in other phrases	Check if the subject of an interaction phrase occurs as object in other phrases. In this case, the phrase often means "The actual subject/actor observes that". One can maintain the original phrase structure, but the subject will not become a candidate actor, but most likely a candidate domain class. Example: In a toll registration system "The vehicle passes the toll booth" could be rewritten as "The system observes the vehicle passing the toll booth". But the original phrase is more natural and can be kept. And the subject "vehicle" will most likely occur as an object in other phrases, causing it to be a candidate domain class.
Determine relevance	Ignore phrases about the document itself	Ignore phrases that are about the document itself, like an explanation of the document structure, or sentences that "glue" paragraphs together. For example, an extracted phrase like "TO explain <some topic> in chapter", or "TO summarize document in summary" can be ignored. (Unless the domain is writing reports of course).

Step	Name	Description
Eliminate homonyms and synonyms	Replace generic verbs with a domain specific term	Be aware of generic verbs. These are often data-oriented verbs or verbs that are easily applicable to a neighboring domain. Examples of data-oriented verbs are: • Create, identify, enter, define, describe, register, select, add • Update, adapt, change, modify • Delete, terminate, erase, remove, end These verbs are typical for administrative and conceptual objects. Preferably use a more semantical term from the actual domain. For example, “change address of a person” is actually “person moves to a new address” or “enter an order” becomes “place an order” or simply “to order”, or “change the color of the wall in blue” is really “paint the wall blue”. The verb term may be overloaded, when there is no good alternative available according to the domain experts. For example, to describe a person could be seen the same as the same activity as to describe a dog. But in case you consider it to be different, you can postfix the general verb with the direct object, resulting in “to describe person” and “to describe dog”.
Eliminate homonyms and synonyms	Standardize logical constructs	Logical constructs are not always formatted uniformly in the input text. Sometimes punctuation is used to construct sentences containing the semantics “if-then-else“, “for all”, “implies that”, and “or”. By adding a clarifying keyword like “then” or parentheses it becomes clear which interpretation is meant. It also holds for operations like “is equal to” or “has the same value as”. Use a uniform syntax to express conditions and operations in corresponding phrases. For example, replace “when it rains, take an umbrella” with “if it rains, then take an umbrella”.
Classify candidates	Identify features and functions from text headers	Text headers often indicate the name of a function or feature, because texts are typically written as a coherent chronological series of events.
Classify candidates	Identify functions from use case interactions	If use cases, user stories, system (interaction) scenarios, are used as input source, then the steps that describe system behavior are often calls of system functions.
Classify candidates	Definite articles indicate a function attribute	A definite article, *i.e.*, “the”, might point to a role an object plays in a function. Classify it as a function attribute with a (domain) class as a type.
Declare and allocate elements	Auxiliary verbs indicate the specification space type	Auxiliary verbs can be an indication if a phrase belongs to a feature or to a domain. Verbs like “will, can, be able” indicate that it is content for a domain model. Verbs like “must, shall, should, ought to” indicate that it is content for a feature model.
Identify specification spaces	Begin with one context, one domain, and one feature	If there are no existing specifications spaces and there are no obvious boundaries, then start with one context, one domain, and one feature.
Identify specification spaces	Separate contexts for domain definition from contexts for feature interaction	Separate concepts for defining domain class attributes, domain activity attributes, and operations in activity models, from concepts that are needed to specify the interaction of features with their environment. Examples of the latter are external actors and operations that are the type of function events.
Create initial model, Specify function signatures	Introduce feature attributes for sets of existing objects	A feature is often activated in an environment of sets of (independent) objects. Such sets of objects often serve as the pool of objects to select from for participation in function steps. Introduce set attributes for those objects in the feature or in the top-level functions.

Step	Name	Description
Create initial model, Specify function signatures	Introduce feature wide attributes for the central objects	Features, and sometimes top-level functions in the feature, often center around one or more central objects, which are referred to via a function attribute. Such an attribute can be global in the feature (or the top-level function) to prevent that other functions must define it separately as a function attribute.
Specify feature structure	Start specifying functions for domain classes that are not a part of a composition or aggregation	If there are not already functions defined on a domain, then at least functions are needed to create and manipulate the objects that are not dependent on other objects; the so called strong objects. Namely those objects are needed to instantiate the weaker objects that dependent on them.
Specify feature structure	Define a separate function for function steps sequences that occur more than once	Like normal functional decomposition used in programming or a function-oriented modeling method, a functional decomposition is handy when several higher-level functions contain the same sub-behavior. This common sub-behavior may be captured in a separate function.
Specify feature structure	Define a function for coherent behavior that will be assigned to one actor	Define functions for units of behavior, often called tasks, that will be assigned to and performed by one actor.
Specify feature structure	Check if atomic object manipulations are domain activities	During feature specification, one may find functions that manipulate a single object. Take into consideration if such a function should be defined as a domain activity. If so, it should be allocated to the domain model. It might be a domain activity if it is about what can happen and not about what must happen or how it happens, and if it is independent from the feature, and any actor that performs the function.
Specify feature structure	Find functions from system specifications	System functions can be found from several perspectives: • System use cases indicate a system function, and steps in the use case scenario indicate lower-level functions. • Sub-systems in a system architecture often indicate a high-level function. • A decomposition of the system requirements may indicate functions and sub functions. • Chapters or aspects in a requirements document indicate features or high-level functions.
Specify feature structure	Check the domain models for unused activities	Go through the domain activities of the relevant domain models and check if they should be used in the feature. It is not that all activities must be used, because a feature might only cover a domain partially. But if activities are not used at all, they might be forgotten in the feature model, or might not be a domain activity at all.
Specify function lifecycle	Go with the flow	Begin with the major function steps: • The activities and functions that must be executed in the function. • Their temporal ordering: sequence, selection, parallel, iteration. Initially skip: • Constraints of steps (enter criteria and exit criteria) • Decision logic of coordinators (guards of selections, forks, iterators) • Decision logic of step participants, *i.e.*, constraints on the attributes that are allocated to step participants.

Step	**Name**	**Description**
Specify function lifecycle	Check if all function sub-behaviors are occurring as a step in a function lifecycle	All the function sub-behaviors should occur at least once as a step in the function lifecycle. It means that the type of the function step is the same as the type of the function sub-behavior.
Specify function lifecycle	Check for temporal words in the input text	Temporal words in the input text are a hint about the passage of time or the position of an event in time, usually indicated with a transitional preposition (*e.g.*, after, before, during, until). Other temporal words can also be a hint, *e.g.*, now, eventually, suddenly, initially.
Specify function step	Check the used activities with the domain model	For each domain activity that is used (invoked) in a function step, immediately cross-check the domain activity with the domain model. Is the domain activity also present in the interaction view? Does it have the same objects associated with it (using the same prepositions)? Postpone other cross checks with the domain model until the function lifecycle is completed. By doing the domain model check immediately, the domain model is validated as well, and it is assured that all the domain activities usages conform to the domain model.
Specify function step	Check the consistency between constraints that involve the same domain class	Find all the constraints that are relevant for a function step and that involve the same domain class. These are not only the constraints directly connected to that step, but also the ones that are specified at a higher aggregation level, e.g., as an invariant of a container function. Then verify if they are free of contradictions.
Specify function step	Default allocation	Often one domain class will only occur just once as the type of an attribute in a function. That means that such an attribute is probably the attribute that will be allocated to all function step participants that have that domain class as type. In practice, this guideline can imply that those step participants do not need to be allocated manually.