Methodical conversion of text to models
MuDForM Definition and Case Study

Robert Deckers[1] and Patricia Lago[2]

[1] Vrije Universiteit Amsterdam & Atom Free IT, robert.deckers@AtomFreeIT.com
[2] Vrije Universiteit Amsterdam, p.lago@vu.nl

Abstract. To enable the people involved in a software development process to communicate and reason close to their area of knowledge, we are investigating a method to formalize and integrate knowledge into domain models and into specifications in terms of those domain models. For this purpose, we have previously defined a list of method objectives, and an initial version of the method –called MuDForM. This paper reports on the method part that covers the creation of an initial model from textual documents via systematic grammatical analysis, which is especially helpful in the transition from a text-based- to a model-driven development process. We performed a case study in the printing domain to validate the method. We found that the presented analysis concepts, method steps, and guidelines help to systematically convert a textual specification into an unambiguous model.

Keywords: Method engineering · Natural language processing · Domain modeling · Model-based engineering · Software Engineering

1 Introduction

This work introduces an integral modeling method, called Multi-Domain Formalization Method (MuDForM), which provides support for the creation of *domain models* (DM), and for the creation of models that are defined in terms of a domain model, called *domain-based models* (see Fig. 1). All together, these are called *domain-oriented models*. MuDForM provides analysis and modeling concepts, steps, and guidelines to conduct a modeling process, which starts with a knowledge source, like a *(domain) text* or *(domain) expert*. This paper explains the part of MuDForM that transforms a textual specification into an initial *MuDForM model*, and demonstrates it in a case study.

The rest of this section describes the problem we aim to address, our contribution, and target audience. Section 2 explains in more detail what we aim to achieve with MuDForM. Section 3 explains the research methodology. Section 4 gives an overview of MuDForM, which explains how the support for grammatical analysis (GA) and the text-to-model transformation, respectively defined in Sections 5 and 6, are integrated in MuDForM. Section 7 reports on a case study in which we applied MuDForM to formalize system behavior descriptions.

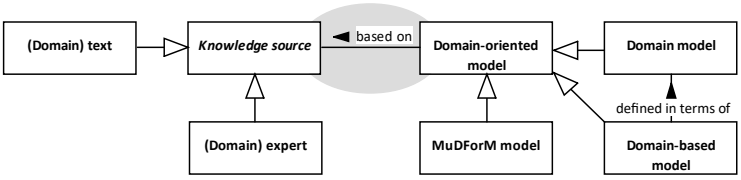


Fig. 1: Context of MuDForM models (UML class diagram)

Section 8 reflects on MuDForM's support for grammatical analysis. Section 9 discusses related work, and Section 10 concludes the paper and presents suggestions for future work.

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

Kosar *et al.* [10] present a systematic mapping study on DSLs. They conclude that (domain) analysis is mostly done in an informal and incomplete way. Among the reasons for this weakness, they mention that domain analysis is too complex and outside software engineers' competencies. Czech *et al.* [3] gather 130 best practices from 19 studies on domain-specific modeling (DSM). They group the best practices in different classes: domain model, language design and concepts, generators, DSL-tooling, meta-model tooling, and practices that concern an entire DSM-solution. Only 3 best practices are about the domain model, and those are actually not about modeling itself, but about the context of a domain model. We observe that they did not find and distill any best practices for extracting domain models from text.

Deckers and Lago observe in a systematic literature review (SLR) [6] that most approaches for domain-oriented specifications do not offer full methodical support for extracting models from natural language texts. Some offer a meta-model, some others process steps, or guidelines. But none integrates them all.

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

Contribution and Audience. This paper has two main contributions. First, it presents methodical support for the analysis of domain texts to extract model elements and model fragments. Practitioners may use the support to bootstrap their modeling activity. Method developers may use the description of the sup-

port as an example of how to extend a modeling method with a part for bootstrapping a model from an input text.

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

2 Background: MuDForM Development

To understand the reason behind the work that is reported here, we explain what we aim to achieve with MuDForM.

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

We have presented the objectives MuDForM in [6]. The objective relevant for this paper is as follows: Almost all people, including domain experts, use natural language to convey their knowledge and decisions. It is used in many documents that are relevant in a system development process. A specification method should **support the transformation of knowledge described in natural language into unambiguous models**. The purpose of this support is to minimize loss of semantics and increase mutual understanding in the communication between modelers and domain experts.

3 Research Methodology

This section describes the research methodology we have applied to gather the results presented in this paper. Based on the problem statement and the above explanation of the MuDForM vision, we define the following research questions:

(RQ1) What methodical support can be given for the conversion of text into ingredients of an initial domain-oriented model? The answer is given in Sections 5 and 6, in terms of GA concepts, method steps, and guidelines, and validated though the case study from Section 7.

(RQ2) How should methodical support for extracting knowledge from text be integrated in a method that aims to produce domain-oriented models? The answer is given in Section 4, in terms of how modeling concepts and method steps fit with MuDForM's other modeling concepts and method 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 Peterson *et al.* [13,14], which inspired us to organize our study along the phases described below.

Diagnosis. Based on our experience with modeling, architecture, and model driven development in the past 25 years, we have defined a vision on software development and related method objectives (see Section 2). In parallel, we started to define the method, which led to initial versions of metamodel, method flow, and guidelines. We have started to record and generalize our experiences, and work them out in detail since we started the MuDForM research program in 2015. The method definition is available in [5].

Action planning. We have performed a SLR [6], which was derived from the same method objectives. From the SLR and the initial method definition, we identified topics needing further research, and the parts of MuDForM that needed further development. One of them is the methodical support for extracting models from natural language texts, *i.e.,* the topic of this paper. 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.

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

Case study. We defined the case-specific objectives together with the industry partner, and agreed on the timeline, and availability of people and documentation. We performed the case study and presented and explained the recorded model to the industry partner. They used the final model as the terminology in Gherkin test scenarios (*e.g.,* [17]), in order to make those scenarios unambiguous, and provided feedback. In Sections 8 and 9, we reflect on the case study from the perspective of the research questions and related work.

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

4 MuDForM Overview

This section presents an overview of MuDForM, which forms the framework for the method parts described in Sections 5 and 6.

MuDForM is defined according to the guidelines of Kronlöf [12], which has resulted in a method definition with the following ingredients: (i) a metamodel containing classes, activities, attributes, associations, specializations, and constraints, which define the modeling concepts and their relations, and (ii) a method flow containing steps, guidelines, and viewpoints, which guide the modeling process.

Section 4.1 explains the overall MuDForM modeling process; Section 4.2 the high level structure of a MuDForM model; and Section 4.3 the modeling concepts that form the link between the GA and the model engineering phase.

4.1 MuDForM Modeling Process

Figure 2a shows the steps of the MuDForM modeling process:

1. **Scoping**: the scope of the targeted model is specified by defining its purpose, its boundaries, and the input text that is selected from the knowledge source. The knowledge source is often an existing document, or a document that is created from interviews with (domain) experts.
2. **Grammatical analysis**: the input text is analyzed and transformed into a set of phrases with terms that are candidate elements for the model. The goal of this step is to maximize the knowledge elicitation from the source, and to make the resulting model traceable back to the input. This step is explained in detail in Section 5.
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 model fragments, which are allocated to one of the specification spaces. This transformation is the transition from working with text to working with models, and is explained in Section 6.
4. **Model engineering**: the model is completed and inconsistencies are solved. 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. The complete MuDForM definition [5] contains more detail about the sub steps of model engineering.

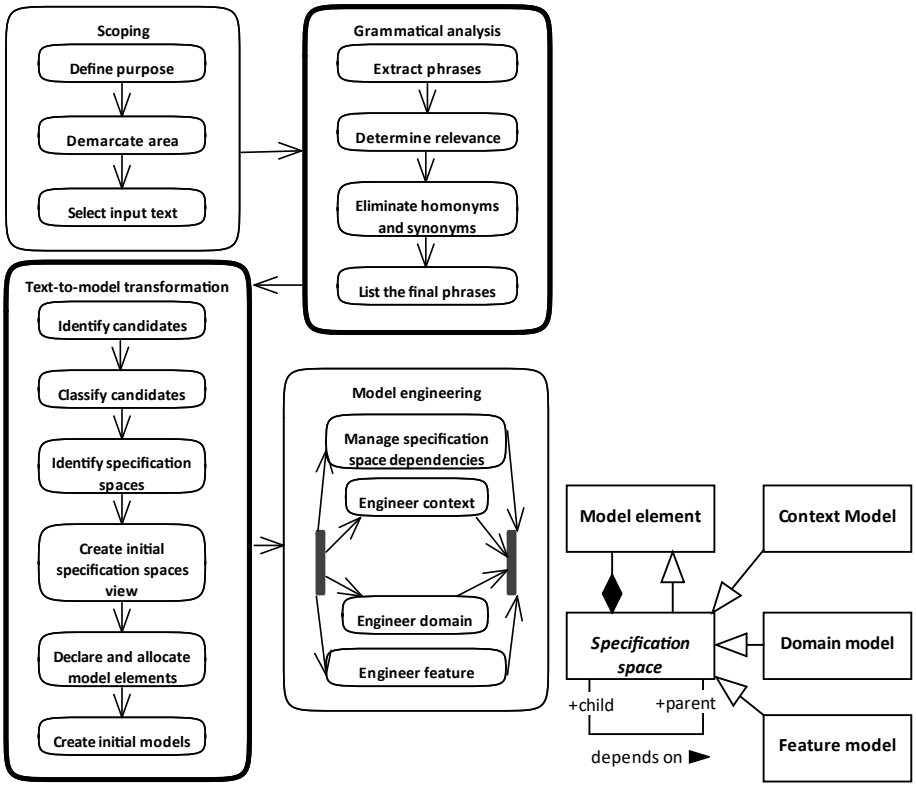
4.2 MuDForM Model Structure

The top-level structure of a MuDForM model consists of related specification spaces, as depicted by the MuDForM metamodel fragment in Fig. 2b. MuDForM uses specification spaces (similar to UML packages) as containers for model elements.

In our notion of domain, a domain model describes what can happen and what can exist in a domain. A feature model prescribes what shall happen and what shall exist, and is expressed in terms of domain model elements. Context models capture assumptions and knowledge about elements that are needed to specify domains and features, but that exist outside those domains and features. By defining the dependencies between the different specification spaces, domains and features have no implicit semantics.

4.3 MuDForM Model Elements

MuDForM offers different types of model elements. The type of specification space, *i.e.,* domain, feature, or context, determines which types of model elements are allowed, and what is their semantics. The three different specification spaces all have concepts to specify state, behavior, and concepts to specify the relation between state and behavior. Moreover, almost all model elements can have **attributes** and **specializations**, and can have **constraints** attached to them. The following elements are specific for engineering the domain model, and are thus possible output of the GA and text-to-model transformation:



(a) MuDForM method flow (b) Specification spaces

Fig. 2: MuDForM overview.

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

We have limited the explanation above to the domain model, because feature models and context models are absent in the description of the case study in Section 7. Though, they are explained in the complete MuDForM metamodel [5].

The overview of MuDForM from this section forms the context for the definition of GA and text-to-model transformation, which are explained in the next two sections.

5 Grammatical Analysis

This section describes the grammatical analysis step as introduced in Section 4.1. This paper only describes the method steps and GA concepts. The full method description, including the metamodel and guidelines can be found in the MuDForM method definition [5]. The sub steps of grammatical analysis are:

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

   Table 1 in Section 7.3 shows examples of these phrase types. 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 analysis decisions.
2. **Determine the relevance** of each extracted phrase from the perspective of the defined scope. Discard phrases that do not fit the scope definition. Also check if phrases are still valid in case legacy text is analyzed.
3. All phrases are checked for **homonyms and synonyms**. These are then **eliminated** in consultation with the domain experts to assure that all terms have exactly one meaning, and that all relevant meanings are covered by exactly one term.
4. This results in a **list of final phrases** which is used as input for the model. The list contains all extracted phrases that are marked as relevant and not discarded, and newly added phrases, in which the identified homonyms and synonyms are replaced with the chosen term.

During the analysis, issues can be raised for an analysis item, *i.e.,* a phrase or term. Guidelines can be used in the decisions made to solve an issue. Fig. 3 presents the metamodel for GA and the text-to model transformation.

6 Text-to-model Transformation

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

1. **Identify candidates**: Determine which terms, *i.e.,* nouns, verbs, adjectives, and adverbs, are a potential model element.

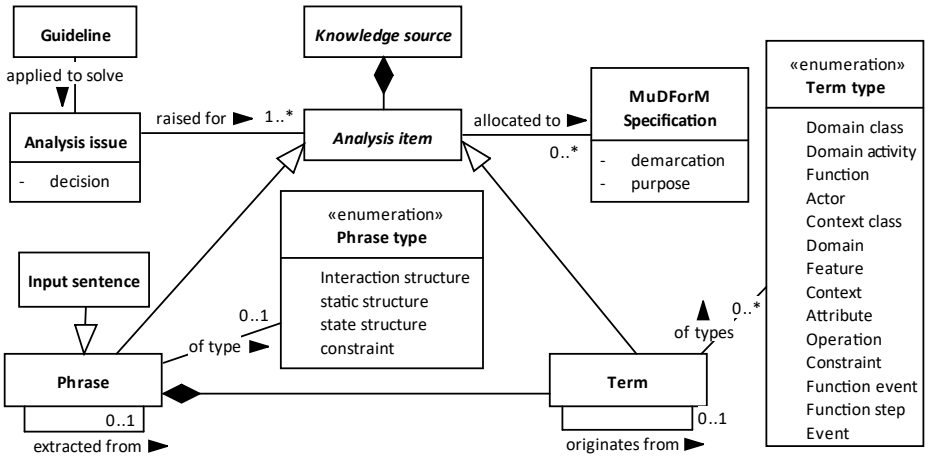


Fig. 3: MuDForM concepts for grammatical analysis (UML class diagram).

2. **Classify candidates**: Select the type of each identified term. The metaclass Term type in Fig. 3 gives the possible types, which are partially explained in Section 4.3.
3. **Identify specification spaces**: Identify contexts, domains, and features. Each specification space should have an owner who is responsible for its content.
4. **Create initial specification spaces view**: create a view with all the specification spaces. Create dependencies and compositions between spaces if they are expected, or already known.
5. **Declare and allocate elements**: create a model element for each candidate term and put it in a specification space. The model engineering phase will reallocate an element if it was initially allocated incorrectly.
6. **Create initial models**: create a first version of the models from the list of final phrases. All interaction phrases become a relation between a behavioral element (activity, operation, function) and a class. All static structure phrases become an attribute of the subject, and the attribute type corresponds with the object of the phrase. All state structure phrases become a generalization relation between the subject and the nominal part of the phrase. For the constraint phrases it depends; they can become invariants, preconditions, postconditions, or a temporal ordering in the lifecycle of a domain class or function.

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

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

7.1 Introduction to the Case

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

The goal of the case study is to evaluate MuDForM support for transforming textual specifications into initial models. The rest of this section focuses on the phase from text to initial domain model. Deckers and Lago present more GA examples, including some for feature modeling, in [7].

7.2 Case Study Overview and Execution

The case study was executed as a collaboration between a MuDForM researcher, a modeling expert from the customer to guard the fit for purpose of the model, and several domain experts supporting the unraveling of unclarities in the SBD text. The case study was performed between January and April 2022.

During the modeling process, the most important decisions were recorded, and some of them are used in the explanation of the results. We show examples of the resulting model to illustrate how the method is applied. The complete model is not publicly available due to intellectual property rights. But we have made a more elaborate excerpt of the case available via [4]. The next two sections discuss the execution of the steps Grammatical analysis and Text-to-model transformation as explained in Sections 5 and 6.

7.3 Grammatical Analysis of the History SBD

The GA starts with **Extracting phrases** from the input text. We follow the guideline "Use a structure to separate input sentences", as described in [5], and created Table 1, which shows the sentences that are selected from the History SBD, and the phrases that are extracted from them. In each row, the first column contains the input sentence, and the second column has one or more extracted phrases. After the extraction, we **Determined the relevance** of each phrase together with the domain expert, and **Eliminate homonyms and synonyms** across the phrases. The last column explains the analysis decisions made for the raised issues, possibly with a reference to the guideline on which the decision is based. After that we **List the final phrases**, which are the *emphasized phrases* in the table. For clarification, we explain one row (highlighted in gray) of the table: "Therefore, jobs that are too old will automatically be removed from the history". First, we extracted two phrases: "TO remove job from history" and "Job IS too old". We already had "to Delete" so we asked what is the difference with "to Remove". The domain experts said they are synonyms, and chose the term

"to Delete". Following the guideline "Detect type of adjectives and adverbs", we asked what kind of thing "too old" is. The involved domain experts could not immediately provide clarity and were discussing about it. So, we applied the guideline "Postpone too long analysis discussions" and kept the information as is, and postponed the discussion to the model engineering phase, which will solve the issue because then the discussions are more directed due to the use of specific viewpoints like the the object lifecycle, and model engineering criteria like (data) normalization.

Table 1: Selection of the grammatical analysis

Input sentence	**Extracted (including *final*) phrases**	**Decisions (including new *final phrases)***
When a print job is completed, it will be archived in the so-called "History".	*TO complete job* TO archive job in history	To archive and to move are synonyms. Chosen: to Move.
The History is a job store that will be used as a local temporary job store and is not intended for long term archiving purposes.	*History ISA job store* TO use history as local temporary job store TO intend History for purpose	To intend and to use are ignored because of guideline "Ignore intention phrases".
Only jobs that have been completed will end up in the History.	TO complete job Job TO end up in history	To end up is not a domain activity. "job is in History" is a state after "to archive". Chosen: *to move job from job store to job store*
Proof prints initiated from the waiting room and system jobs will not end up in the history when completed.	TO initiate proof print from waiting room *System job ISA job*	To initiate is considered out of scope. (It is in the scope of Job scheduling.), but *Proof print ISA job.*
Also jobs that have been aborted or deleted will not end up in the History.	*To abort job* *To delete job*	
The Settings editor provides functionality to clean up the History at specified time periods. The following time periods can be specified: One day, One week, One month, Forever.	To clean up history at time period. TO specify time period. One day, one week, one month, forever ISA time period.	Use retain period instead of time period. Furthermore, it is the retain period of the History which is specified, giving *TO specify retain period of history*, and *TO clean up History at retain Period.* One day, one week, one month, forever are possible values of retain period.

Input sentence	**Extracted (including *final*) phrase**	**Decisions (including new *final* phrases)**
Jobs that have been longer in the History than the specified time period for the automatic cleanup are removed from the History	*History HAS jobs* To specify time period	
Therefore, jobs that are too old will automatically be removed from the history.	TO remove job from history *Job IS too old*	To Remove and to Delete are synonyms. Chosen: to Delete. Following the guideline: "Detect type of adjectives and adverbs", we asked what kind of thing "too old" is. We did not get a clear answer. So, we kept it.
If the history is disabled new completed jobs will be removed from the system, so they will not end-up in the history.	*TO disable history* TO complete job TO remove job from system	System and controller are synonyms. Chosen: controller. Giving: *Controller HAS history*
A job can be reprinted from the History by copying them from history to waiting room.	TO reprint job from history *TO copy job from history to waiting room*	Is "reprint" the activity or "copy"? Answer: To copy. Reprint is the intention. And, what is a waiting room? Answer: *Waiting room ISA job store. Giving: TO copy job from job store to job store.*

7.4 Text-to-model Transformation for the History Domain Model

This section discusses the creation of the initial models from the results of the grammatical analysis.

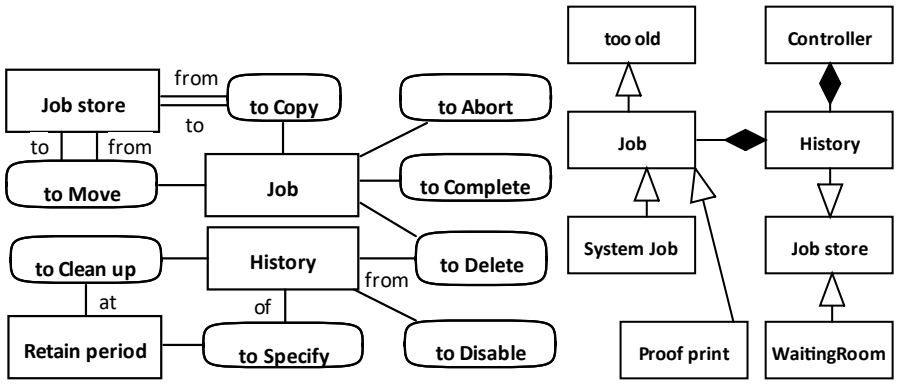
The first steps are **Identify candidates** and **Classify candidates** as described in Section 6. In this case, every term becomes a domain class if it is a noun, or a domain activity if it is a verb. "Too old" is an adjective probably indicates a possible value of a context class. But we classified it as a class, for reasons explained in the previous section.

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

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

The last step is to **Create the initial models** from the phrases, which resulted in Figures 4a and 4b. All the *emphasized phrases* are present in the

diagrams. The more elaborate report of the case study contains more phrases (see [4]).



(a) Interaction view (b) Static view

Fig. 4: Initial model of the History domain.

8 Discussion

In the following we reflect on the research questions and discuss the findings from the case study. We first discuss the support for GA (RQ1 from Section 3) and how it fits into the rest of MuDForM (RQ2).

Figure 2a presents the steps for the conversion of a text into an initial model. The steps form a clear structure on how to organize this process. Each step can be planned and executed accordingly. Though, we found out during the case study that in practice it is easier to first focus on the basic elements of the domain model and not on the constraints and other aspects of the feature model. This means there are at least two iterations. The first iteration focuses on the extraction of HAS and ISA phrases, and interaction phrases that have no actor, *i.e.,* most phrases starting with "TO". After that, conduct the model engineering until the domain model is stable. The second iteration is about the extraction of interaction phrases with actors and constraint phrases, which can immediately be rewritten to match the created domain model. This second iteration is also a validation of the created domain model. Namely, all the constraints should be expressed in terms of the domain model and possibly context model. If not, then either the constraint phrase is unclear or incorrect, or the domain model must to be adapted. Thanks to this insight, we have added the guideline "First do the domain model, then the feature model" to the MuDForM method flow [5].

Section 4 presents how the steps for the text-to-model conversion fit into MuDForM. The partial metamodel of Fig. 3 addresses how the concepts fit. All the possible values for Term type and Phrase type correspond to classes and

relations from the rest of the MuDForM metamodel [5]. The fact that we do not have all the classes from the MuDForM metamodel as a possible term type is due to two pragmatic reasons. First, we have only put in the metamodel classes that we have actually used in one of our past modeling projects. Second, the main purpose of the model engineering phase, which comes after the phase described in this paper, is to bring preciseness, consistency, and completeness to the model. The modeling environment is more suitable to do that than the natural language environment. Though, it might be possible that we change the possible Phrase types and Term types due to new insights in later projects.

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

On the metamodel. The presented metamodel (Fig. 3) has concepts that are specific for GA, which are related to the MuDForM modeling concepts via the classes Phrase type and Term type. For an other method, other phrase types and term types may be used. For example, most domain modeling methods do not have a primary modeling concept for specifying behavior, like the domain activity concept in MuDForM. They just model classes, attributes, and relations between classes, and often capture behavior in class operations or in generic data-oriented operations like create, update, and delete.

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

On the method steps. The four main steps of the MuDForM method flow (Fig. 2a cf. page 6) can be generalized into: Scoping, Discovery and Elicitation (for capturing specific knowledge from a knowledge source), Switch to modeling, and Model engineering. In general, the GA step can (partially) replace Discovery and elicitation step from another method. Having different modeling concepts may also imply that the step of switching from text to model will differ.

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

9 Related Work

Deckers and Lago performed a SLR on domain-oriented specification techniques [6]. It identified several approaches that extract models from text [1, 2, 8, 9, 11, 15, 16, 19]. None of these approaches provides a metamodel for GA.

MuDForM is based on the KISS method [11], which is the only approach from the mentioned SLR with an explicit phase and concepts for GA, and a distinction between domain and feature. It however does not provide a metamodel, fine-grained method steps, or guidelines.

Abirami *et al.* [1] give guidelines for conceptual modeling of non-functional requirements. They overlap with the MuDForM guidelines for extracting phrases, but do not distinguish an explicit intermediate step for GA.

Arora *et al.* [2] present an approach for extracting domain models from natural- language requirements. They give guidelines for creating classes, associations, and attributes from sentences. Some of those guidelines are also present in MuDForM. The main difference is that they do not distinguish behavioral concepts, such as the domain activity concept in MuDForM.

Elbendak *et al.* [8] describe an approach for automatic generation of class diagrams from use case descriptions. They solved the issue of multiple binary associations representing one action by using n-ary associations. Though, they do not distinguish between domain, feature, and context, and let the creation of a class in the target model depend on the number of occurrences that its corresponding noun has in the text. The same holds for the paper from Sagar and Abirami [16], which reuses and improves many of the rules given by Elbendak, and introduces a clear distinction between a strict text-to-model transformation and suggesting model candidates. Though, it is limited to models that can be captured fully in standard UML class diagrams. Ibrahim and Ahmad [9] introduce a tool for the automatic extraction of class diagrams from textual requirements, which follows many of the rules from the other papers.

Although we are open to automating part of the text-to-model process, we think that the involvement of domain experts in the GA process is essential. They do not only provide missing information and help to eliminate homonyms and synonyms, but often feel more comfortable with discussing natural language sentences than with discussing graphical models, which mostly have their own specific metamodel. The paper from Hoppenbrouwers *et al.* [19], which is based on the KISS method [11], makes a claim for partially automating the text-to-model phase, such that domain experts are still actively involved via natural language. MuDForM also supports the involvement of domain experts via the verbalization of models in natural language, which is also addressed by Proper *et al.* [15], Kristen, [11], and Hoppenbrouwers *et al.* [19].

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

10 Conclusion and Future Work

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

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

The results from our study fill an important gap in the state of the art, which to the best of our knowledge lacks in providing methodical support in the first place. It lays the foundation for our future work on building a validated and reusable set of guidelines, for which we plan the following: (i) Building a community that actively validates, identifies, and manages guidelines. (ii) Conducting a literature review to find, and analyze guidelines from natural language processing approaches, *e.g.,* the primary studies from [20] *et al.*, to possibly integrate in the GA step of MuDForM.

To facilitate industrial adoption, we plan to create a MuDForM handbook for practitioners, and manage its evolution via an open platform, as replacement of the document that contains the method definition [5]. We are currently investigating the requirements and possibilities for a modeling tool that supports MuDForM, in order to replace MS Word and Enterprise Architect [18].

References

1. S. Abirami, G. Shankari, S. Akshaya, and M. Sithika. Conceptual modeling of non-functional requirements from natural language text. In Lakhmi C. Jain, Himansu Sekhar Behera, Jyotsna Kumar Mandal, and Durga Prasad Mohapatra, editors, *Computational Intelligence in Data Mining - Volume 3*, pages 1–11, New Delhi, 2015. Springer India.
2. Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Extracting domain models from natural-language requirements: Approach and industrial evaluation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 250–260. ACM, 2016.
3. Gerald Czech, Michael Moser, and Josef Pichler. A systematic mapping study on best practices for domain-specific modeling. *Software Quality Journal*, pages 1–30, 2019.
4. Robert Deckers. From text to model for the sbd history. Technical report, Atom Free IT, online at https://github.com/robertdeckers/CaseStudySBDHistory, 2022.
5. Robert Deckers. Mudform method definition. Technical report, Atom Free IT, online at https://github.com/robertdeckers/MuDForM, 2022.
6. Robert Deckers and Patricia Lago. Systematic literature review of domain-oriented specification techniques. *Journal of Systems and Software*, pages 1–23, 2021.
7. Robert Deckers, Dennis van den Brand, and Patricia Lago. Modeling features in terms of domain models: MuDForM method definition and case study. https://research.vu.nl/en/publications/modeling-features-in-terms-of-domain-models-mudform-method-defini, under submission.
8. Mosa Elbendak, Paul Vickers, and Nick Rossiter. Parsed use case descriptions as a basis for object-oriented class model generation. *Journal of Systems and Software*, 87:1209–1223, July 2011.
9. M. Ibrahim and R. Ahmad. Class diagram extraction from textual requirements using natural language processing techniques. In *2nd International Conference on Computer Research and Development (ICCRD'10)*, pages 200–204. IEEE, 2010.

10. Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.
11. Gerald Kristen. *Object Orientation, The KISS Method, From Information Architecture to Information System.* Addison Wesley, 1994.
12. K. Kronlöf. *Method integration, concepts and case studies.* John Wiley and Sons, 1993.
13. Kai Petersen, Cigdem Gencel, Negin Asghari, Dejan Baca, and Stefanie Betz. Action research as a model for industry-academia collaboration in the software engineering context. In *Proceedings of the 2014 international workshop on Long-term industrial collaboration on software engineering*, pages 55–62, 2014.
14. Kai Petersen, Cigdem Gencel, Negin Asghari, and Stefanie Betz. An elicitation instrument for operationalising gqm+ strategies (gqm+ s-ei). *Empirical Software Engineering*, 20(4):968–1005, 2015.
15. H. A. Proper, A. I. Bleeker, and S. J. B. A. Hoppenbrouwers. Object–role modelling as a domain modelling approach. In *Proceedings of the Workshop on Evaluating Modeling Methods for Systems Analysis and Design (EMMSAD'04)*, pages 317–328, 2004.
16. V. B. Vidya Sagar and S. Abirami. Conceptual modeling of natural language functional requirements. *Journal of System and Software*, 88, 2014.
17. John Smart. *BDD in Action: Behavior-driven development for the whole software lifecycle.* Simon and Schuster, 2014.
18. Sparx Systems. Enterprise architect version 15.2. https://sparxsystems.com/products/ea/, 2021. Accessed: 2021-08-19.
19. B. van der Vos, J. Hoppenbrouwers., and S. Hoppenbrouwers. Nl structures and conceptual modelling: the kiss case. In *Applications of Natural Language to Information Systems: Proceedings of the Second International Workshop*, page 197, 1996.
20. Tao Yue, Lionel C Briand, and Yvan Labiche. A systematic review of transformation approaches between user requirements and analysis models. *Requirements engineering*, 16(2):75–99, 2011.