Atom Free IT
for true business agility

MuDForM Definition

Metamodel, method flow, and viewpoints

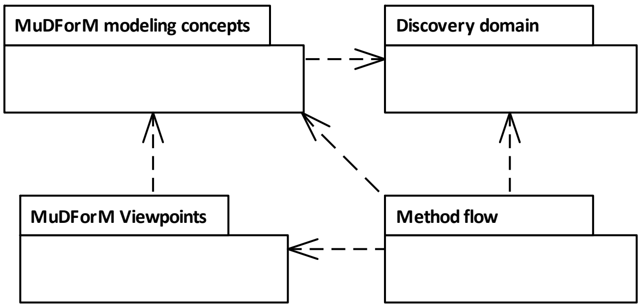
Version:3-9-2022 13:31:01
Author: Robert Deckers

Distribution and usage limited to: Atom Free IT

Table of Contents

1 MuDForM

The MuDForM metamodel is defined in four packages. The core is formed by the MuDForM modeling concepts the Method flow. The Discovery domain contains the concepts for elicitation of knowledge from experts and text. The viewpoints package describes the different viewpoints, which are used in the method flow.



MuDForM Definition

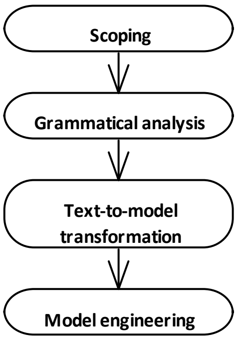
2 Method flow

This package describes the steps and guidelines of MuDForM. It builds upon the concepts defined in the Modeling concepts package and the Discovery package.

Each step is described, and per step preconditions, postconditions and guidelines are given. Preconditions describe what must be valid before the step can be performed successfully. Postconditions define what must be valid at the end of the step, which means that the step is not finished until the postcondition is true. Guidelines give help in achieving the postconditions or how to perform the step in practice. (The order in which the guidelines are presented here doe noet have a specific meaning.)

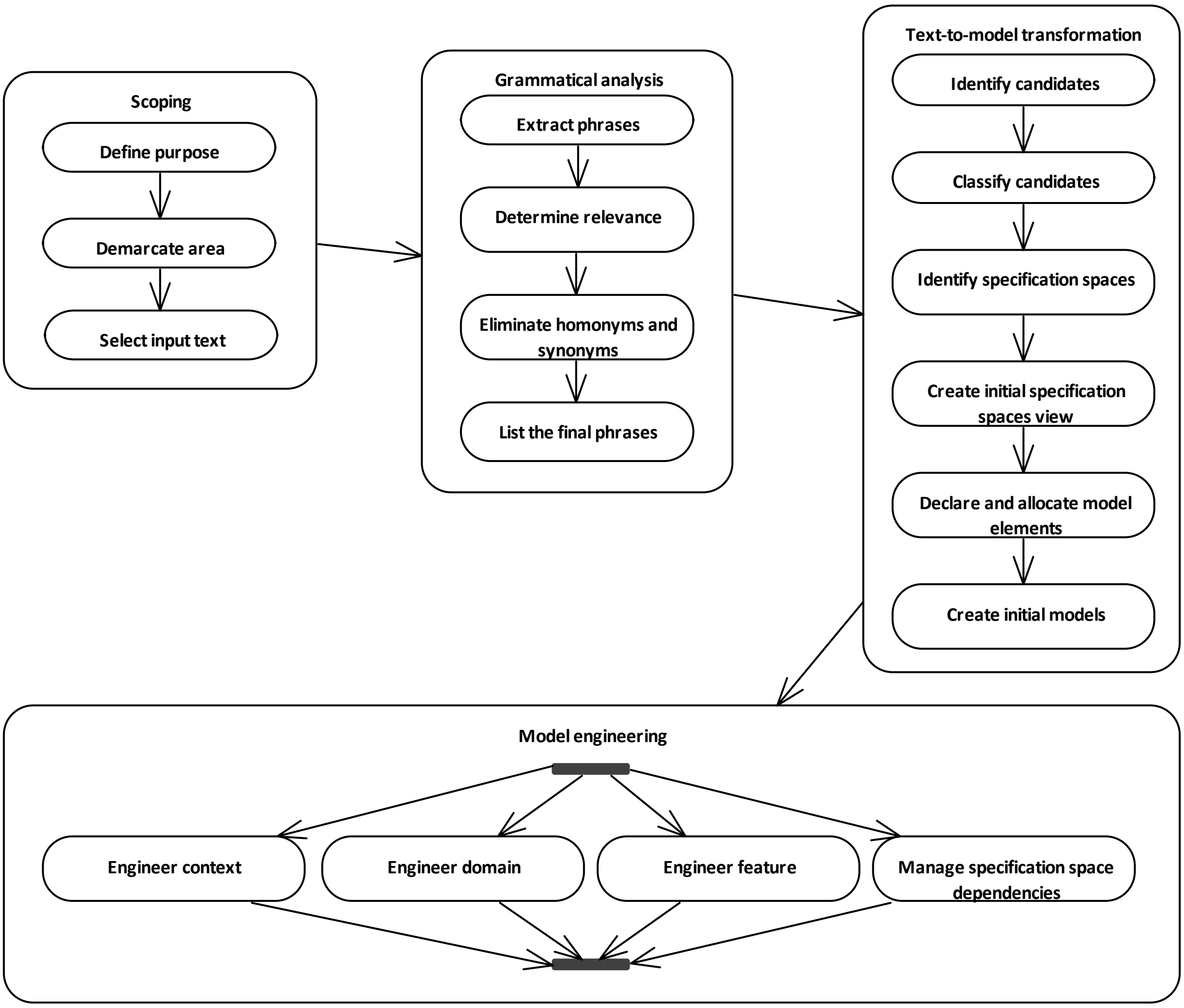
An experienced modeler may be inclined to skip steps or to follow guidelines of a step further in the process, for example by already "knowing" how a concept will end up in future modeling steps. This may save work in the short run, but the risk is that less experienced modelers, and especially involved domain experts, loose track of the reasoning behind the modeling step and perceive the modeling method as a very difficult and opaque activity. The advice is to stick to the method steps, unless the domain experts agree to apply “look ahead” insights in earlier steps.

2.1 Main flow diagram



Main flow

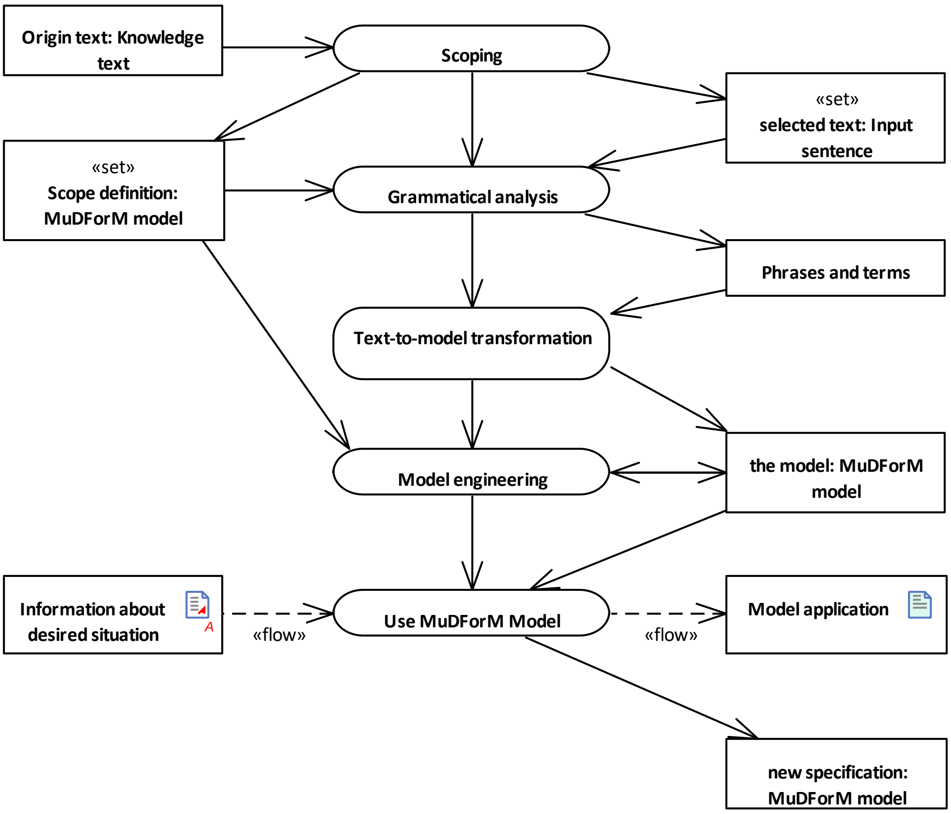
2.2 Main flow and sub flows diagram



Main flow and sub flows

2.3 Main flow with objects diagram

MuDForM consists of five main activities that typically happen sequentially, but also may overlap in practices. It starts with setting the scope where the input text is selected and people are involved. Then, knowledge from these people and text is extracted, grammatically analyzed, and transformed into a format that makes it suitable for modeling. After that, a first model structure is defined, and the analysis results are transformed into model elements. The core of the process is the model engineering phase in which all the modeling rules and guidelines are applied when iterating over the different model views. The process ends with applying the made MuDForM model in some context. This can be making another MuDForM model or any other usage.


Origin text: Knowledge text
Scoping
«set»
selected text: Input sentence
«set»
Scope definition: MuDForM model
Grammatical analysis
Phrases and terms
Text-to-model transformation
Model engineering
the model: MuDForM model
Information about desired situation
«flow»
Use MuDForM Model
«flow»
Model application
new specification: MuDForM model

Main flow with objects

2.4 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.

In the case that there is no explicit starting text, you can make an initial model by putting sentences in a tool together with a group of domain experts, and of course you skip the Grammatical analysis step. Such a tool can be a whiteboard, a modeling tool like Enterprise Architect, or like KISS Domino.

The goal of scoping is to have the proper 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.

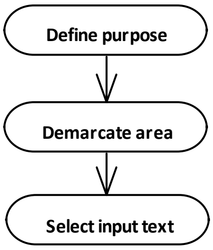


Figure 1: Scoping

Precondition: Have relevant text

Assure that you have text available that you can select from and that the involved experts acknowledge as being relevant. This can be an existing document or notes from interviews with the experts for example.

Precondition: Stakeholders involved

A set of stakeholders should be involved in the whole analysis and modeling process, either directly or via representatives. There should be at least one expert for each of the targeted domains, and one customer (user) for each of the targeted specifications (domains and/or features). They do not have to be different people. Examples of stakeholders are the people that must use a targeted system, implement the specification in a system, specify tests for a system, or are responsible for the delivery of the system and thus the correctness of its specification.

Guideline: Specifications with a purpose
A clear purpose specification for each (expected) specification space.

2.4.1 Define purpose

Define what the customer/user of the targeted model wants to do with the model that you are going to make. What kind of specifications do you want to make with the model? What applications/systems do you want to build from the targeted model? One could define the purposes as use cases of the model.

Guideline: Common domain model purposes
When in doubt about the purpose, check if these common purposes for domain models are applicable:

- Provide the terminology for specifying a specific feature.
- Provide terminology for specifying other domain models, i.e., to extend the domain model into several other domain models.
- Derive specifications in another domain, typically a software domain. For example, in a code generation for a specific target platform.
- Provide terminology for specifying requirements for a system that operates in or controls the domain, e.g., specify the requirements for a fuel saving system in terms of the car driving domain.
- Provide terminology for specifying tests to check if a system operating in the domain passes such test.
- Provide the terminology to rewrite texts about the domain such that hose texts are aligned regarding the used terms.

Guideline: Common context model purposes
When in doubt about the purpose, check if these common purposes for context models are applicable:

- Provide the terminology to specify a specific domain or feature. The targeted context model should contain terms that have meaning outside the targeted domain or feature, and that are needed to define behavior and structure of domain classes, domain activities, functions, and their attributes.
- Provide the terminology to specify the *interaction* of a feature with this context, i.e., the external actors and their capabilities, or events that features must react to or generate.

Guideline: 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.

Guideline: 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.

Guideline: Involve a stakeholder for each purpose
Have at least one stakeholder involved in the modeling process for each purpose. ALso, each stakeholder that intends to use the targeted specification should have a purpose defined for it.

Postcondition: A purpose expresses an activity
A purpose description explains what happens with the model, and what goal that activity has. It should be clear what the value and role of the mode is for that activity.

2.4.2 Demarcate area

Give an indication of the concepts that are in scope and the concepts that are out of scope. If you already think in modeling concepts, then you can name classes and activities for domains, and functions and actors for features.

Otherwise, you can name nouns, verbs, or phrases. It is also possible to use a describing rules to demarcate the scope. For example, things that this type of user interacts with, or concepts provided through the API of that system.

Guideline: Keep revisiting the demarcation
Demarcation is an ongoing activity. After every step throughout the modeling process, you have more information available to discriminate between in "in scope" and "out of scope". This means that the demarcation typically becomes more accurate throughout the modeling process. Instead of doing the demarcation once:

- Make the demarcation a regular discussion point during the rest of the modeling process.
- Do not spend much time on it in the beginning of the modeling process. 15 minutes is a maximum.

Guideline: Mention adjacent specification spaces
Mention specifications spaces (domains features) that are related in some way. For example, name other perspectives on the same concept, like sitting on a chair vs. crafting a chair vs. selling a chair. Or, functionality that your features interacts with, like selling goods interacts with buying goods and delivering goods.

Guideline: Explicit inclusion and exclusion of concepts
Make not only a list of concepts that are in scope, but also a list of concepts that are out of scope. In practice, involved (domain) experts already have an idea about these. Be practical, because theoretically the rest of the universe is out of scope. But, mentioning things just outside the boundary of the domain can be helpful. The goal is to have a criterion for including/excluding a phrase or term and in which specification space to allocate it to.

For example, in the taxi driving domain, passenger, destination, luggage, fee, are in scope. But, repairing and assembling a car are not.

Guideline: Start with the top-of-mind concepts
Do a five-minute brainstorm with a domain expert, or a short model storming session with a group.

2.4.3 Select input text

Select sentences from the text, which form the input for the grammatical analysis. The starting document can be a specific report, article, and/or set of interviews with domain experts. Examples documents are a project requirements document, a specification of a process in the domain, an application specification, use case scenarios, or an official standard that is applicable to the scope of the targeted model.
If there is no existing document, you can write an initial text with the help of domain experts, or just start making an initial model using, for example, Model storming or KISS domino.

Guideline: Avoid long relevancy discussions
Do not try to select the "perfect" text in the beginning. The later modeling steps will filter out irrelevant information better than one can achieve by selecting and analyzing the right pieces of natural language text in the beginning.
When choosing text is difficult, choose quickly and start working with the choice. This way, you get relevant feedback quickly. Don't be afraid to drop the choice and choose again.

Guideline: Start with concepts from the demarcation
If you cannot easily select a piece of text, because the document is too large and it is unclear where the relevant sentences are, then search for sentences with the most important terms according to the demarcation.

Guideline: 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.

Guideline: Involve the text expert
Involve the person who is responsible for the content of the selected input text, like the author or an expert on the content. The modeler might have his own ideas, but the domain experts have the knowledge and, more important, must feel they are the owner of the specification. Trust their hunch and use their input as direct as possible.

Guideline: Ignore contextual text
Skip text that does not directly address the targeted topic, like an overview of the document structure, or a section about background or future work.

Guideline: 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.

Guideline: Format your interviews
In case you interview experts to obtain text, try to format the sentences from the interview according to one of the phrase types. If you do so, then the phrase extraction can be skipped. Also, you can direct the interview separating questions for the domain model from questions for the feature model, i.e., asking what is possible vs. asking what should happen. It is advisable to only apply this guidelines if you are an experienced modeler. If not, then just stick to plain natural language.

2.5 Grammatical analysis

The input text is analyzed to acquire a set of structured phrases.
The first step is to extract phrases from the selected input text. For each of the extracted phrases the relevance is determined. Then the phrases are checked for homonyms and synonyms, and other ambiguities. This results in a set of phrases and terms for transformation into the model.
During grammatical analysis, all decisions regarding the phrases or terms must to be made with the approval of the involved (domain) experts.

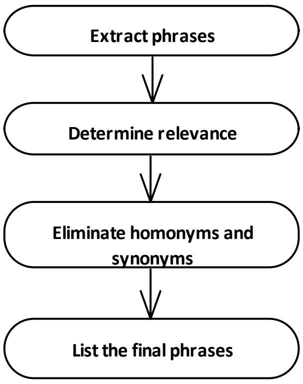


Figure 1: Grammatical analysis

Guideline: Combine steps for efficiency
Experienced modelers may combine or skip method steps (of grammatical analysis) for efficiency of the process. For example, to not extract irrelevant phrases, or already replace synonyms, which you identified before, when you extract a phrase. This prevents unnecessary work. Explain that you do this, and only do it if the involved people agree.
Be aware, if involved people (such as domain experts) have to follow the whole analysis process, you may loose them if you take steps that they can not follow. In case of doubt, do not skip or combine steps.

Guideline: Limit the amount of phrases to process in one go
To avoid an overload of information that must be discussed with the involved experts, you should limit the amount of phrases for such discussions. This can be achieved by picking phrases that are connected to a chosen noun, and center the discussion around that noun.

Guideline: Postpone long analysis discussions
When a discussion about a term or phrase takes too long (e.g., more than two minutes), or when involved domain experts disagree, then keep the information and postpone the discussion to the model engineering phase. Namely, the model engineering provides more overview and multiple viewpoints, which puts modeling questions in a clearer perspective.

Guideline: Raise issues
If the modeler cannot easily make a decision about a phrase or term during grammatical analysis, then raise an issue for it. Then discuss it with the involved expert.
Whenever an issue is raised and you cannot resolve it quickly, write it down as “Open Issue” and move on. In some tools you can enter an issue at any place and moment, and you can query for them later when you want to discuss them.

Guideline: First address the domain, when there is no existing domain model or the text is large.
Do at least two iterations with a text when you target at both a domain model and a feature model; especially when the text is large, and when you do not have an existing domain model.
The first iteration focuses on the extraction of static and state phrases, and interaction phrases that have no actor, i.e., most interaction phrases starting with "TO". Then do 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. If not, then either the constraint phrase is unclear or incorrect, or the domain model must to be adapted.

2.5.1 Extract phrases

Extract phrases from each sentence in the selected input text. Format each phrase according to one of the phrase types. This should be a very mechanical activity (which probably can be automated). The meaning of the phrases or validity of the phrases is not important in this step, nor its validity. The primary objective is to capture as much knowledge as possible from each input sentence. If knowledge in a sentence cannot be formulated in a static, state, or interaction phrase, then capture it in a constraint phrase.

Guideline: Extraction is more important than the phrase type
When it is not clear what phrase type to choose for a part of a sentence, then just choose one and continue. It is easier to understand the grammatical relation between terms in a phrase when having an overview of all its occurrences, than to derive it from a single sentence.

Guideline: Investigate what behavior constitutes a verb
When it is unclear what kind of changes/behavior an active verb indicates, then investigate (with the domain expert) what behavior makes up the behavior denoted by a verb. This might lead to candidate operations for an activity model, or steps in a function lifecycle. Though, do not overload the grammatical analysis with phrases about data transformation, transfer, or simple calculus.

Guideline: Replace verb clauses with a preposition object
Sentences with a clause that expresses a means or reason to the main verb, can be replaced with a preposition and an object. For example, "He uses the key to open the door." can be replaced by "He opens the door with the key.". The verb in the clause is often a generic verb like "to use" or "to apply".

Guideline: Use a structure to separate input sentences
Make a structure to document the analysis for each original sentence. For example, a table where each sentence has its own row. Put the original sentence in the first column. Put all the extracted phrases in the second column. Add a third column for open issues and decisions. Follow the order of the input text.
A row in such a table could be:

1. **Original Sentence:**

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.

2. **Extracted phrases:**

History ISA job store
TO use local temporary job store
TO intend History for purpose

3. **Issues and decisions:**

To intend and to use are ignored because of guideline “Ignore intention phrases”.
Of course, other structures besides a table could be used, possibly supported by a dedicated grammatical analysis tool.

Guideline: Ignore actors in domain model extraction
If you are (only) modeling a domain model, then the subject of a interaction structure phrase is often not relevant.
You can avoid a specific subject by using the term “Someone” for the subject or by using the infinitive form of the verb, like “to order”.
The subject can be relevant if you can make the verb reflexive, like in “The customer identifies himself with a passport.”.

Guideline: 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.

Guideline: Detect type of adjectives and adverbs
Check the relevance of an adjective or adverb in a state phrase, i.e., an IS-phrase of the form <noun/verb> is adjective/adverb>. For example, given "the car is blue", then ask if there are also non-blue cars. If not, then you can ignore the adjective or adverb, or keep it when it essential in the wording of "noun/verb phrase". If yes, then replace state phrases of the kind <noun/verb> IS <adjective/adverb> with a static phrase of the kind <noun/verb> HAS <aspect> and a state phrase of the form <adjective/adverb> ISA <aspect> . The replacement is a noun that expresses the aspect that the adjective is about. In the example: "car is blue" is replaced by "car HAS color" and "blue ISA color" . Clearly, this only makes sense if cars can have other colors besides blue.

Guideline: Leave out indefinite articles
Leave out indefinite articles ("a" and "an") from extracted phrases. This way the extracted phrases contain only words that can go directly into the models.

Guideline: A genitive case indicates a static structure.
Genitive cases in noun phrases indicate a static structure.
For example, "the color of the car" indicates "car HAS color".

Guideline: A possessive indicates a static structure
A possessive apostrophe indicate a static structure. For example, "the car's color" indicates "car HAS color".

Guideline: Containment indicates a static structure
If a noun phrase has the preposition "in" between two nouns, a verb that expresses containment, indicate a static structure. For example, "the basket contains apples , or "the basket has apples in it", indicate "basket HAS apple".

2.5.2 Determine relevance

Determine the relevance each extracted phrase. Ignore phrases that do not fit the scope definition, or that are (partial) duplicates. Terms in a phrase can be replaced with a verbalization of an already existing model element, i.e., use the term from the existing model if it is applicable. Also check if phrases are still valid in case legacy text is analyzed. In case the relevance is unclear, keep the phrase. Namely, it is easier to discard a phrase later in the process, then to find it back and fit it in.

Guideline: 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).

Guideline: Ignore intention phrases
Ignore interaction phrases that are about the intention of the text content, like "... is used to", "... is meant to", "the purpose of ...", or "it is the intention that ...". Unless those verbs are about the domain of interest, or if the phrase expresses a constraint like a postcondition.

Guideline: Find a verb that expresses the change
Verbs that express a result or an effect can be ignored. For example, from the phrase "the list grows by adding items to it" the phrase "to add item to list" can be extracted because it expresses a change. But the verb "to grow" can be ignored, because it expresses an effect. For example, in the phrase "the items finally end up in the trash bin" the verb "to end up" expresses an effect. One can ask what activity leads to that effect. The answer can be "to delete an item into the trash bin".

2.5.3 Eliminate homonyms and synonyms

Avoid that one term has more than one distinct meaning and avoid that one meaning is represented by more than one term. So, replace terms that are a homonym or synonym in phrases, with the chosen term. The chosen terms may also come from already existing models.

Guideline: The term must be recognized by the expert.

Let the domain expert choose the term in case of synonyms. Let the domain expert choose a new term in case of homonyms.

Guideline: 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".

Guideline: Verbs with different sets of related nouns indicate a homonym
A verb is probably a homonym if it occurs in multiple phrases and the verb has different objects or prepositions related to it in those phrases. For example, in the phrases "I pay attention to the waiter" and "I pay the bill", "pay" is probably a homonym.

Guideline: When in doubt, assume a noun is a homonym.
In case it is not clear if a noun is a homonym, apply the following: say it is, and then decide per interaction phrase which verb is connected to which of the two terms for the noun. If you cannot decide, it was probably not a homonym. If needed you can also use the static phrases and decide per direct object of such phrase to which of the two possible subjects (nouns) it belongs.

Guideline: When in doubt, assume a verb is a homonym
In case it is not clear if a verb is a homonym, apply the following: say it is. The modeling of the object lifecycles of the domain classes that correspond with the nouns connected to the verb, will make clear if they are different, because the two verb will lead to different steps in the object lifecycle. If needed, then making the activity views that correspond with each of the two verbs, will clarify if it was a homonym.

Guideline: When in doubt, do not assume two terms are a synonym
The model engineering will make clear if they are synonyms, because then they will have the same views, e.g., object lifecycles and attribute views for nouns, and same occurrences in object lifecycles, attribute views, and activity view for verbs.

Guideline: Use the most occurring term in case of synonyms
If it is difficult to choose a proper term for two synonyms, then use the term that occurs in the most phrases.

Guideline: put the direct object in a homonym verb
If a verb is a homonym and it is difficult to choose a new term, then create the new terms by putting the direct objects of the corresponding interaction phrases after the verb. If the direct objects are the same, then use one of the indirect objects. In case this does not help, just postfix the verb with a number.

Guideline: 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".

2.5.4 List the final phrases

Form the set of phrases for the initial model via:

- All extracted phrases that are marked as relevant and are not discarded.
- All newly added and rewritten phrases.
- Replace the possible homonyms and synonyms in those phrases with the proper term.

- Combine phrases with the same verb but with different nouns into one phrase.

2.6 Text-to-model transformation

To create an initial model from the results of grammatical analysis, i.e., assign a model element type to a candidate, and identify and relate the specification spaces of the model, and position the model elements and phrases in those spText-to-model traces.

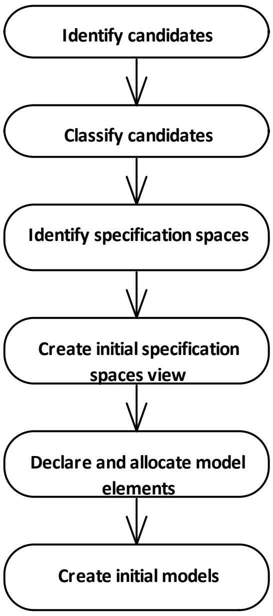

Identify candidates
Classify candidates
Identify specification spaces
Create initial specification spaces view
Declare and allocate model elements
Create initial models

Figure 1: Transform text to model

2.6.1 Identify candidates

Select which terms (subjects, nouns, verbs, adjectives, adverbs) from the phrases are a potential element for the targeted model. Some phrases might be a candidate themselves as a whole, in the case of a condition.

Guideline: Keep a term if you are not sure
When in doubt, make it a candidate. The modeling engineering process will solve the uncertainty.

Guideline: Check if a verb has more objects related to it
Check (with the experts) if a verb has more objects related to it than the ones you already found. You can just ask, or use a list of prepositions and check if any of the prepositions should be present in the phrases that involve the verb.

For example, you have "to drive a passenger". Ask: Where to drive to or from? Answer: to drive from a departure location to a destination.
List of frequent prepositions: about, above, across, against, along, among, around, at, behind, between, beyond, by, down, following, for, from, in, into, like, near, of, off, on, onto, out, over, past, throughout, to, towards, under, up, upon, up to, via, with, within.
There are about 150 prepositions in total in the English language. So, keep it pragmatic.

Guideline: Elicit more phrases for important nouns
Ask the involved stakeholders to come up with more sentences (phrases) for an identified noun. This is a way to be more complete and depend less on the completeness of the initial text. You typically do this for nouns that are important, but for which you only have a few phrases.

2.6.2 Classify candidates

Select which type of modeling element each identified Term, i.e., noun (phrase), adjective, verb (phrase), and adverb, is. The possible types are given by the metamodel class Term type.

Guideline: Classify changes by default as activities
If it is not clear if a verb must be classified as an activity, a function, or an operation, then classify it as a domain activity. Then begin with searching for domain model concepts by building the interaction view of the domain model. Then it will

become clear if some verb is a combination of other activities, and most likely is a function. If it is not a function because it is not a behavioral unit in a feature, nor a unit of change in a domain, then it is part of a context, and thus an operation.

Guideline: 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.

Guideline: 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.

Guideline: Value setting verbs indicate operations
Verbs that are about setting a value to object properties indicate an operation. Think of verbs like assign, copy, inherit, instantiate, add, and set. They indicate that object attributes get a value inside a certain activity, which makes them candidates for operations.

For example, when the taxi starts with a new ride, then the departure location of the ride is set to the current location of the taxi. In this case, to start is the domain activity and set is an operation in that activity.

Guideline: Check if a noun is related to more verbs
If a noun occurs in only one phrase with a verb that indicates a change, then check (or ask) if there are more relevant interaction phrases (with other verbs) in which that noun occurs. If the answer is yes, then classify it as a domain class. if not, then it is either a context class or no class at all.
For example, given "To drive a passenger to a destination". Ask: "what else do you do with a destination?" Answer: "to search for new passenger at that destination."

Guideline: Check if a candidate indicates a set or an instance
If it is not clear that a candidate indicates a set (classifier), then ask if there are more instances possible that fall under that candidate's term and if they fit the scope. The candidate can be a noun, pronoun, or verb.

Guideline: Focus on the core concepts
Focus the classification on the domain classes, domain activities, and functions. Spend less time on attributes, and on candidate classes and operations that go into contexts.

Guideline: The candidate type may differ in different phrases
If you cannot determine the type of a candidate, then just keep the phrases that have the candidate and decide the type per phrase.

Guideline: Not all subjects are candidate actors
Subjects that not actually perform or control behavior are not an actor. They are probably a domain class, or sometimes a function.

Guideline: Domain activities must change objects
Verbs that indicate an action in which derived information is generated, and that are not changing the state of one or more objects, are not a domain activity. They are so called inspections, i.e., queries. So, they are either a candidate function, operation, or a function step.

For example, For example, calculating the expected driving distance of a taxi ride, based on the departure location and destination is not a domain activity, because nothing happens to the ride, nor to the two locations.

Guideline: 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.

Guideline: The types of attributes are classes from a context
The type of a domain class attribute or a domain activity attribute is probably a class in a context model. These types occur as direct object in a static phrase.

2.6.3 Identify specification spaces

Identify the specification spaces and classify each of them as context, domains, or features. Of course, the existing specification spaces must be taken into account.

Guideline: Begin with one context, one domain, and one feature
In case there are no existing specifications spaces, and there are no obvious boundaries, start with one context, one domain, and one feature.

Guideline: Separate incoherent contexts
Separate contexts if a group of context candidates is clearly not related to another group of context candidates.
For example: a taxi ride has a fee, which is an amount of money. The taxi ride also has departure location, and a destination, which are locations. Location and money amounts are unrelated things, and should be defined in separated context models.

Guideline: Separate specification spaces for different owners
Identify separate specification spaces for pieces of model that are the responsibility of different (involved) people. This way the ownership of a piece of model is clear. Also the link between the responsibilities must be made explicit through the dependencies between the separate specification spaces.

Guideline: 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.

2.6.4 Create initial specification spaces view

Define a view (e.g., a diagram) with all the specification spaces. Create a dependency between two spaces if you think that one will use concepts from the other. Typically, domains depend on contexts and possibly other domains, and features depend on features, domains, and contexts.

For example, the taxi driving domain depends on the navigation domain, and the context of Payments.

Guideline: Begin with one context, one domain, and one feature.
In case there are not already existing specifications spaces, and there are no obvious borders, it is best to start with one context, one domain, and one feature.

2.6.5 Declare and allocate model elements

Create a model element per candidate in the right specification space.

Guideline: In case of doubt, position it in the domain model
When it is difficult to determine to which specification space an element belongs, one should add it to a domain model. The model engineering step will reposition the element if needed.

Guideline: 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.

Postcondition: All candidates have a corresponding model element.
Per candidate at least one model element is created. A candidate that had multiple types has a corresponding number of model elements. For example, a grammatical subject might become a domain class and an actor, or a verb might become a domain activity and a function.

2.6.6 Create initial models

Create a first version of the views in the specification spaces from the list of final phrases, based on the candidate types and the positioning of the candidates in the specification spaces. All interaction phrases become a relation between a behavioral element (activity, operation, function) and a class, all static phrases become an attribute of the subject with as type the object of the phrase. All state phrases become a generalization. For the constraints it depends; they can become invariants, preconditions, postconditions, or a temporal ordering in a object lifecycle, or function lifecycle.

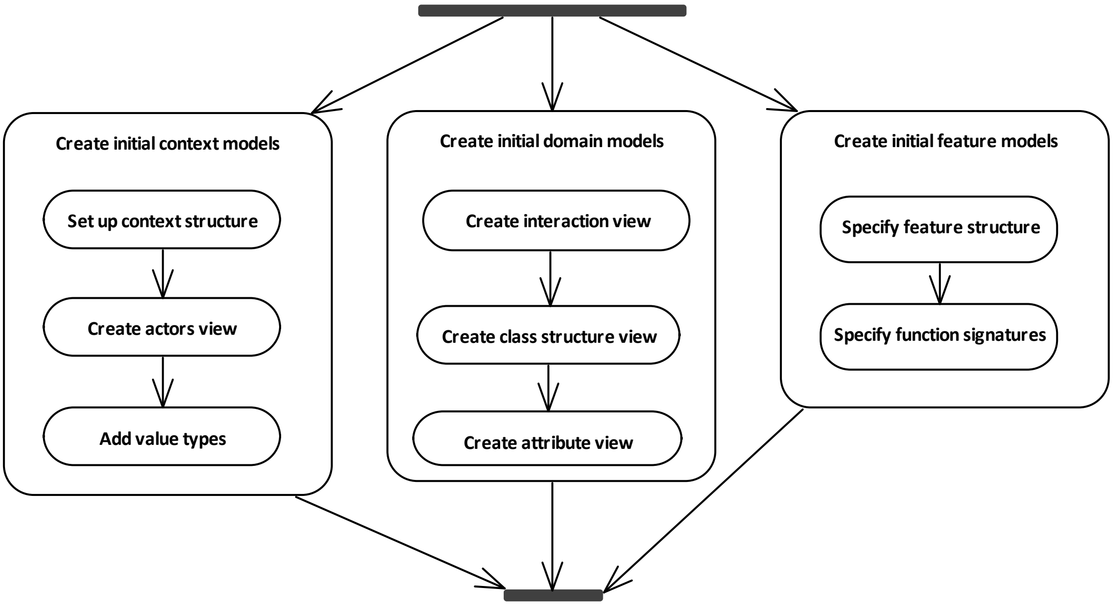

Create initial context models
Set up context structure
Create actors view
Add value types
Create initial domain models
Create interaction view
Create class structure view
Create attribute view
Create initial feature models
Specify feature structure
Specify function signatures

Figure 1: Create initial models

2.6.6.1 Create initial context models

Make the initial views per identified context.

Figure 1: Set up context structure

Make a view containing the classes, operations, and their relations (class relations, attributes, and generalizations).

Figure 2: Create actors view

Define function events for each actor that is not a domain class candidate, from the phrases where the actor is the subject.

Figure 3: Add value types

All the classes that have no attributes or are not a specialization of another class, can be a value type. Add a reference to a value (data) type from a formalism for each value type.

2.6.6.2 Create initial domain models

Make the initial views per identified domain.

Figure 4: Create interaction view

Put all the interaction phrases in a view (e.g., a diagram). Leave out the subjects if they are not a candidate domain class.

Guideline: Only connect the most abstract class to an activity role.
If two interaction phrases have different classes connected to them for the same activity role, and those classes also have an ISA-relation, then only draw a relation between the activity and the parent of the ISA-relation. For example, given "to drive a vehicle", "to drive a car", and "car ISA vehicle" then do not model "to drive a car", but only "to drive a vehicle" and "car ISA vehicle".

Figure 5: Create class structure view

Create the pieces of model for the static phrases and the state phrases, but only if the relation is between two or more candidate domain classes from the same domain.

Guideline: Just use a class for adjectives and adverbs in state phrases
If you do not know the type of an adjective or adverb yet, and it is used in a state phrase (IS-phrase), just draw a generalization relation between the corresponding class or activity and the adjective or adverb. For example, "Car is blue" is modeled as a class "Car", class "blue", and a generalization from Car to blue.

Guideline: Combine class structure view and attribute view
If the classification of candidates did not lead to significant number of context classes, or operations, then you can combine the domain class structure view and the attribute view into one static view. During model engineering, this static view might become too big, leading to a need to split it again.

Figure 6: Create attribute view

For all the static phrases, add attributes to domain activities and domain classes, and a reference to the classes from the contexts for the type of an attribute.

2.6.6.3 Create initial feature models

Make the initial views per identified feature.

Figure 7: Specify feature structure

For all the functions, define a function event for all the behavioral elements that can happen in the life of that function.

Figure 8: Specify function signatures

Define function attributes for static phrases in which the function is a subject.

Guideline: 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.

Guideline: 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.

2.7 Model engineering

During model engineering, the initial models are iteratively transformed into engineered domain models, context models, and feature models. This means making the models comply with the set of rules that MuDForM prescribes. These rules address completeness, consistency among views, and restrictions on the metamodel.
The main principles that guide the modeling are 1) keeping the views consistent, and 2) acquire information from experts or documents to achieve a complete specification.

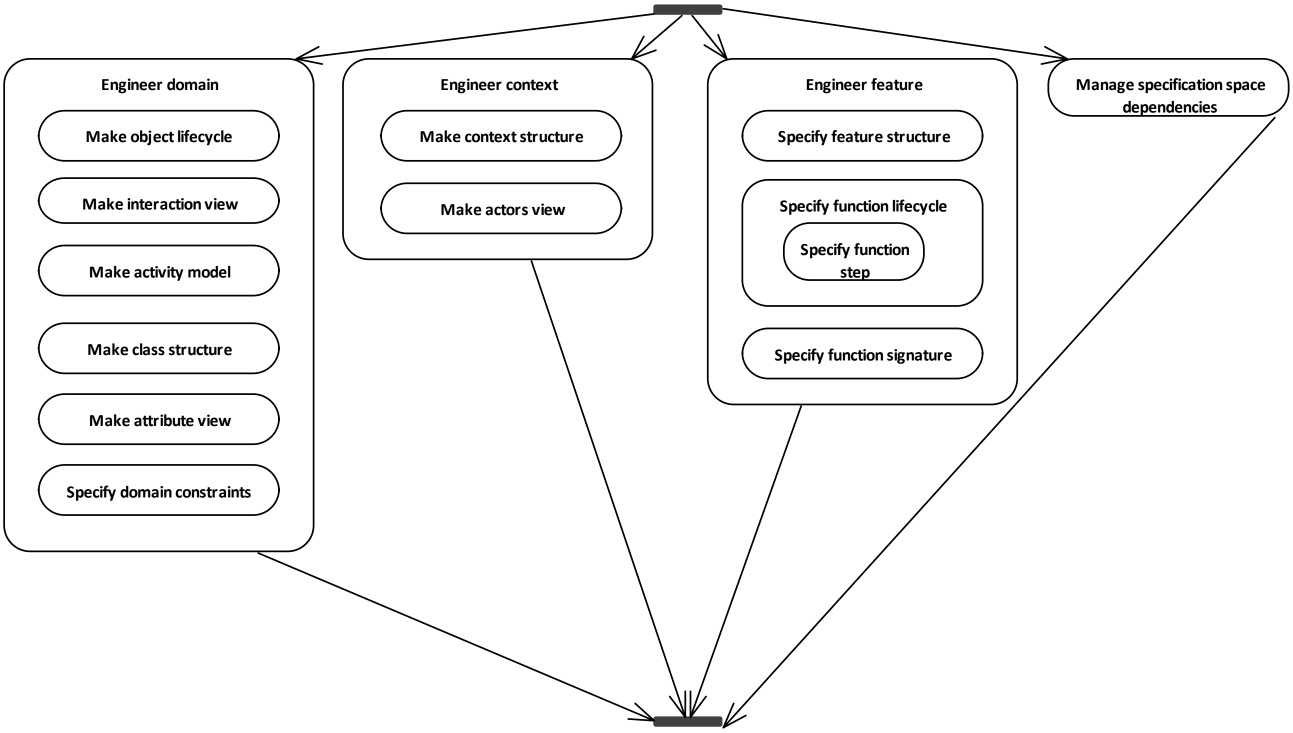
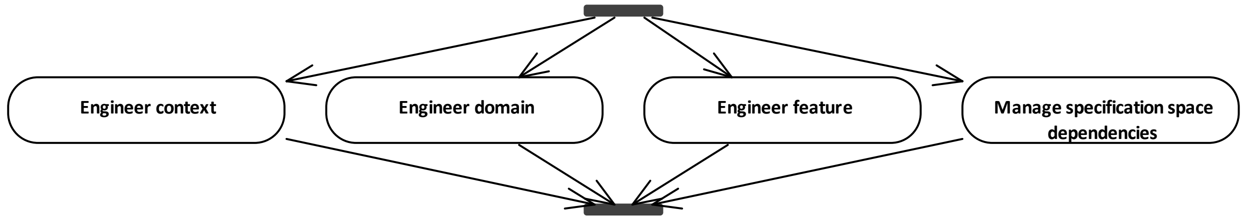


Figure 1: Model engineering



MBT feature engineering:
het selecteren van een object niet een aparte operation (step) hoeft te zijn. Het is gewoon een conditie op de relatie tussen function attribute die gekoppeld is aan de step participant. Kijken of dat object bestaat is denk ik de guard van een selection omdat je ook iets moet als die niet bestaat.
Ik snap natuurlijk de behoefte dat je een soort van default beslissing wilt als een step niet uitgevoerd kan worden omdat niet aan alle precondities voldaan kan worden. Mijn voorstel is dat je dan niks expliciet specificeert, maar het in vertaling naar een werkend systeem stopt. We moeten een soort van default semantiek afspreken dan wat het betekent voor de functie. Bijvoorbeeld: abort.
Even sparrend over wat je allemaal moet checken om een functiestep te kunnen uitvoeren:

- Mag het van het functiemodel:
  - Is dit een mogelijke step, d.w.z. een step is afgelopen en dit is een van de mogelijke volgende steps.
  - Mag het van de vorige step? In het geval van een disrupting step die vorige step beëindigt.

- Mogen alle deelnemende objecten het van hun OLC.
- Zijn de precondities van de het type van de step geldig? (type kan zijn operation, domain activity, function) Denk aan condities voor attributen en involvements (=objecttype in actierol).
- Gelden alle extra condities die op stepnivo gedefinieerd zijn?
- En…. Is de actor er klaar voor?

mbt life existence dependency:

1. De potentiele bestaansafhankelijke ouders zijn de objecten die mee doen aan de instantierende actie van een kind. De vraag die je moet stellen is: betekent het kind nog iets als de ouder niets meer betekent? (guideline in domain model engineering)
2. Sommige deelnemende objecten worden ene ouder en sommigen niet.
3. Met mudform komen daar nog complexiteiten bij door het expliciet onderkennen van een context.
4. Wat doe je met objecten uit de context die deelnemen aan de instantierende actie, maar die geen leven hebben in het model.
5. Mijn antwoord: als die een leven hebben in de context dan moet de actie ook voorkomen in de OLC van de ouder. Daarmee komt er een nieuwe issue: namelijk binnen welk domein(en) wordt die actie geplaatst?
6. Mijn antwoord: er is blijkbaar een overkoepelend domein waarin die actie geplaatst moet worden.
   Voorbeeld: als een imposition “verwijderd” wordt, dan zit natuurlijk de content nog steeds op de imprint, en ook verwijst de imprint naar een impositionID. Ahaaa. Hier hebben we het verschil: als een domainobject een “foreign key” heeft, dan zijn er twee opties: het foreign object kan verwijderd worden en het is dus gewoon een verwijzingsattribuut. Of het foreign object mag niet verwijderd worden. De laatste optie levert een probeel op, want dan moet een functie/domain restricties opleggen aan de context. Ik kan opereren met een context die zich aan mijn restricties houdt.
   Hetzelfde vraagstuk treedt niet alleen op bij het verwijderen van het context object, maar gewoon met wijzigen ook al.

KISS was niet helemaal compleet en “natuurlijk/logisch”: wat doe je als een kind tijdens zijn leven ouder kan verwisselen? In KISS moest je dan het kind end-of-life actie geven en vervolgens een nieuw kind instantieren. Voorbeeld uit bankvoorbeeld: rekening een andere rekeninghouder geven. Dit soort acties wil je kunnen doen. Daarmee zou de bestaansafhankelijke ouder veranderd worden. In mudform wil ik dat zeker toestaan. (in het metamodel: het gaat via domain operations in het actiemodel). Daarmee is er een loskoppeling van semantisch bestaansafhankelijk (typenivo) en objectafhankelijkheid (instantienivo).
Overigens vraagt het verwisselen van ouder wel veel checks” namelijk al die condities die in het leven van een ouder geschonden kunnen worden door acties op een kind. Bijvoorbeeld: een persoon mag maximaal 3 rekeningen hebben. Dit mpet een guideline worden voor acties waar een kind van ouder verwisseld.

Figure 1: Model engineering black box

Guideline: Start with the domains
Although the sub activities of model engineering can de done in parallel, it is probably efficient to start with he domain model(s). During domain model engineering elements will typically be "pushed out" into contexts and features.

Guideline: Describe a view with a story
A view description is a story that talks the reader through the view. It should say something about all the elements that are visible in the view.

Guideline: Ensure commitment of stakeholders
Make sure stakeholders, and especially domain experts are “on board”. They spend time and share their costly knowledge. They must see the model as the representation of their language and knowledge.

Guideline: The definition of a specification space should help to position elements
A specification space definition should help to decide if a specification element belongs to that specification space. The definition contains a purpose and a demarcation of concepts that are in scope and concepts that are out of scope.

Postcondition: Attributes have a type
All elements in an attribute structure are either a substructure with at least one element, or are a reference to a class, i.e., the type of the attribute.

Postcondition: Classifier types of a generalization must match
The metamodel type of the referred classifier in a generalization must be the same as the type of the referring classifier in the case they are declared in the same specification space. (With classifier we mean class, domain class, activity, operation, or actor.)
If they are not declared in the same specification space, then:

- Domain class may refer to context class
- Domain activity may refer to context operation
- Feature actor may refer to context actor
- Feature actor may refer to context class
- Feature actor may refer to domain class

Postcondition: Conditions must operate within scope
All the Condition operands of a Condition may only be bound to specification elements in the specification space of its container. This is to prevent that conditions are defined in terms of elements that are not in the scope of the condition.

Postcondition: Every substructure should have at least two elements
It makes no sense to coordinate (AND, OR, XOR) over one element.

Postcondition: References and specializations may not violate the type/parent
If an element p refers to an element q, i.e., q is the type of p or p is a specialization of q, then the reference p may only change the properties (like attributes, and constraints), if those properties are compliant to q. So, a typed element must always fit the definition of its type. In logic terms: p may not be weaker than q.
This does not only hold for properties that are defined in a structure of q, but also for constraints that are defined upon q, like invariants and preconditions.

Postcondition: Unique names
All specification elements in a specification space must have a unique name.

- Context: classes, class relations, actors, and operations.
- Domain: domain classes, domain class relations, domain activities, and domain invariants.
- Features: functions, and actors.

2.7.1 Engineer context

Make a context model by defining the context structure and actor specifications, cross checking with domain models and features, and using the formalisms for value types and specifications of operations.

2.7.1.1 Make actors view

Specify the actors, their attributes, and their generalizations. Specify the events that the actors can call and react to.

2.7.1.2 Make context structure

Specify the operations and classes, and their attributes. Specify the class relations, the class relation roles, and the role connections. Specify the generalizations between classes and between operations.

Guideline: A class description positions the class in its context
A class definition spends one sentence on the class, and then one or more on the parents of the class and on the other associations of the class. Preferably, give an example instance of the class.

Guideline: Define operators
The meaning of a generic operator might be ambiguous for a used value type (class). Ensure that such operators are defined in a context model. For example, a phone number is a number. But the meaning of "my phone number is larger than yours" is not obvious. Make sure that the meaning of the "larger" operator on phone numbers is defined explicitly.

Postcondition: All value types have a default value

The default value should of course match with the datatype of the value type, which is defined by its formalism.

2.7.2 Engineer domain

Make a complete domain model by iterating over the sub activities and following the guidelines and assuring the postconditions.

Guideline: The Intention of a domain model

- A domain model clarifies what can be managed and controlled in the domain.
- A domain model defines what can happen and what can exist in the domain. “Can happen” excludes should (not) happen, does happen, is likely to happen, has always happened, how can it happen. For example, the functional specification of any system in the domain is not in the domain model; it typically belongs to one of the sytem's features.
- The domain model does not contain elements that are a result of the current way of working in the domain.
- The domain model serves as a shared lexicon. People that are active in the domain should recognize its terms and agree to their definition.
- In the case that a system is analyzed to make a domain model of the functionality domain, the domain model should not contain technologies and elements that are used to make the system work. The domain model should focus on the changes that the execution of the functionality establishes.
- In the case that a system is analyzed to make a domain model of the solution technology, the domain model should not contain information about the targeted application, or about the why the technology is suitable for the application.

Guideline: Do not freeze the domain model too soon.
Stopping the domain modeling too soon may give a false sense of clarity and stability, and may lead to unnecessary complex feature specifications. One can time box the domain engineering activity, but if discussions with domain experts still cause major changes, then it is better to continue with the domain model before applying it in another specification, e.g., a feature specification. If discussions with domain experts lead to reoccurring changes, then applying the domain model in a feature specification must be considered, in order to validate it first and get new insights for the domain model due to feature modeling. Only freeze and release the stable part, and keep the immature part open for change.

Guideline: Analyse if managing and maintaining a domain model will pay off
It costs significant time of several people to make a good domain model and to maintain it. This is time to structure and understand knowledge. It will save coding time, but it is not programming an executable. Consider if domain modeling pays off by asking the following questions:

- Are there multiple applications/systems that involve the domain model?
- Is the domain model used in multiple stages of the development process?
- Must several people understand it? Is the shared terminology essential?
- Are the developers (and others) new to the domain?

If not, one could just do context and feature modeling. Though, in those cases it could still pay off to do a little domain modeling to get the essential concepts clear and to discuss them separately from their usage in different applications. Be aware that most modeling methods, e.g., FODA, OPM, do not separate a domain model from a feature specification (in the same way that MuDForM does).

Guideline: Criterion for distinguishing a domain class
Introduce a different domain class if and only if:

- It has “different” domain activities on it then on another class, .i.e., you do something different with it.
- It has “significant” different attributes.
- It has an interesting life in the considered scope, i.e., it undergoes multiple domain activities in the considered scope. If this is not the case, then it is probably just a class in a context.

One could define domain classes with any kind of granularity. This may lead to two extremes:

- A very generic and reusable model, without enough semantics. The extreme form is a small generic model, namely something has a relation with something.
- A very detailed and large (in the number of elements) model, which leads to an unmanageable and unreadable model. The extreme form is that every object is a class.

Guideline: Criterion for specializations
Only introduce a specialization if:

- Two or more classes share the same relation to another class or action.

- Two ore more classes share attribute definitions. In this case they probably also share a (possible implicit) relation/activity
- A clear case for the above in the future of the model. (This is obviously a vaguer criterion).

Guideline: Criterion for compositions in domain models
A composition (a domain class having another domain class as an attribute type) of a whole and a part is only interesting if:

- The part is reused in other compositions.
- Several parts are instances of the same type (like your left and right eye, or the wheels of a car).
- There is a need to communicate about the part independently from the whole, i.e., the part has a life outside the composition.

It is often said that a domain models should resemble the real-world domain. But this does not help in making a useful model. Especially physical decompositions of a real-world object may not be needed in a model, because it may lead to unnecessary complex structures, i.e., not normalized structures.

Postcondition: Activity attributes refer to classes outside the domain
Attributes of an activity, activity role, or involvement, may not refer to a domain class from their own domain. In those cases, it would not be an Attribute, but an Involvement (of a domain class).

Postcondition: Activities must have at least one domain class involved.
Each domain activity must have at least one role, and that role must be an involvement of at least one domain class.

Postcondition: Life dependencies are set in the instantiating action
The parents that are specified in the life dependencies of a domain class, must have involvements in roles of the instantiating activities of that domain class, or must be types of attributes of the instantiating activities of that domain class.

Postcondition: No derived information
A domain model may not have derived information in the elements. For example, no derived attributes, relations, or classes. This implies that the domain model is in the third normal form.

2.7.2.1 Make object lifecycle

Define the object lifecycle of a domain class, in which involvements of the domain class in activity roles are placed in the OLC, and become an OLC step.

Guideline: Split up a non-atomic domain activity
If a domain activity is not atomic, i.e., the domain class can undergo other activities during the activity, then introduce an explicit begin-activity, and end-activity.
If the activity can be active multiple times for the same object at the same time, then introduce a (weak) domain class to identify an instance of the non-atomic activity. The latter is called normalization.

Guideline: An object can leave a role
A role class should have at least one reverse transformation activity, i.e., an activity which cause the object to exit its role. This is just a guideline, because it can be the case that once a role is entered, the object always keeps it.

Guideline: Split non-atomic activities
Determine for an activity (verb) if it is non-atomic, i.e., it is a composition of other activities, or if just other activities can happen during the activity. If it is non-atomic, then split it into sub-activities; at least a begin and end moment. Ask if there are specific verbs that indicate the beginning and ending of an activity. if not, then just use "begin/start" <activity> and "end/stop" <activity> as names.

Postcondition: An OLC has steps for all involvements.
If a domain class has an OLC, then each of its involvements are at least referred to once by an OLC step.

Postcondition: An OLC must describe a life
Each Object lifecycle must have at least two actions that follow each other. Otherwise, the objects of that class don't have a life, i.e., don't have changeable state.

Postcondition: Each domain class must have at least one instantiating domain activity

A domain class should at least have one involvement in an instantiating activity role. That involvement should be the referred item of the first step in the object lifecycle. There can be more instantiating involvements for a domain class. In that case, the OLC starts with a selection between those involvements.

Postcondition: An OLC includes the OLC of abstract parents and abstract children
If an abstract class that has one or more concrete parent classes (such an abstract class is called role class), then each of the parent classes must specify how the role fits into that parent class. This means that the object lifecycle view of the parent class must clarify how it incorporates the object lifecycle of the role class.

Postcondition: Only proper involvements are the type of an OLC step
An Object Lifecycle may only contain references to Involvements of the Domain class of the Object Lifecycle, or involvements of abstract classes that are a superclass or sub class of the domain class.

2.7.2.2 Make interaction view

Define the domain activities, their roles, and their involvements. Define the generalizations between domain activities.

Guideline: Define a domain activity
An activity definition contains the involved classes and the roles they have in the activity. Preferably an example instance is given.

Guideline: Define a domain class
A domain class definition spends one sentence on the class and then one or more on the parents of the class, and on the other associations of the class. Preferably an example instance is given.

Guideline: Use an abstract class for reoccurring involvements
If a group of domain classes is referred to from multiple activity roles (from the same or different activities, then introduce an abstract domain class. Make a generalization from each of the domain classes in the group to the introduced abstract class.

Postcondition: A class without activities is not a domain class
Each domain class (including abstract domain classes) must have at least one action in which it has an involvement.

Postcondition: Objects enter a role via an action
An abstract child class (=role class) must be associated to at least one activity that transforms an object into that role. Such an activity is called a transformation activity of that role class. This may be the instantiating activity of the domain class to which the role class belongs.

Postcondition: Objects from the domain must be instantiated in the domain
Each domain class must have at least one instantiating domain activity that is declared in the same domain.

Postcondition: The roles of a domain activity must have a unique name within the scope of that activity
Default names are:

- <No name> or “d.o.” to indicate the direct object of the verb that the activity denotes.
- “w.r.i.” for “which results in”. This is often used when an activity is objectified into a domain class.
- “subj” for “subject” to indicate the grammatical subject of the verb that the activity denotes.
- <preposition> in the sentence to indicate an indirect (prepositional) object.

Postcondition: Activities change only objects from the same domain
A domain activity may only change the status of objects of domain classes that are in the same domain as the activity. If an activity involves domain classes from different domains, then an extra domain must be introduced, which contains the activity, to ensure than the OLCs of those involved classes have the activity in their scope.

2.7.2.3 Make activity model

Define the order of operations in the activity model. Operations can be a local operation defined with some formalism, or invocations of operations defined in a context.

Guideline: Ensure the realizability of an activity
Sometimes activity models are skipped. Often this is not a problem for the rest of the specification, because the activity model is a structure of a single activity. But when implementing the activity in software, one might discover that it

cannot be implemented with the available activity attributes, or object attributes. In this case making the activity model will make clear what attributes or domain classes are missing from the scope of the activity.

Guideline: Activity attributes lead to changes in domain class attributes
All input attributes should be used in an operation in the activity model. Activity operations change either local activity attributes, object attributes, or are a domain operation (like creating or deleting a link between two objects).

Postcondition: Activities attributes are not output attributes.
Each activity attribute is either an input attribute or is an internal local activity attribute. An activity attribute is not an output attribute in the sense that the activity sets its value.

Postcondition: An activity model may only refer to related objects
An activity model may only change attributes of involved objects, and it may only refer to attributes or inspect attributes of the action itself, of involved objects, or of parents or those objects. This means that all the actual parameters of the actual parameter structure of the Operation invocations in the Activity model, may only be parameterized by such attributes (and of its roles and involvements).
The activity model may also refer to the history of the involved objects, i.e., refer to a previous object state past. Of course, it cannot change the past state of an object.

2.7.2.4 Make class structure

Define the class relations, generalizations, and life dependencies between domain classes.

Guideline: Names of life dependency relations
By default, life dependency relations are named the same as the role in which the parent is involved in the instantiating action. In the case that role is "direct object", the life dependency relation is named "of".

Guideline: Domain class relations are deleted via a domain activity
All domain class relations (except compositions) should refer to at least one action in which the instance of that association is broken (via a "delete link" operation in the activity model). This is a guideline and not a rule, because it is not necessary that a link can be deleted within the domain. But typically, all things that can be created in a domain, can also be eliminated in that domain.

Postcondition: Life dependency starts at instantiation
All life dependencies of a domain class must be linked in the instantiating activity. The parents of domain class, as defined in the life dependencies, must be involved in the instantiating activity of the class.
qqq check if this is covered in the metamodel qqq

Postcondition: A domain object is an instance of precisely one concrete domain class
An object cannot be instance of just an abstract class. Each abstract domain class must have at least one non-abstract subclass or superclass in the same domain.

Postcondition: Domain class relations are instantiated in the domain
All domain class relations between domain classes (including compositions) must be created in at least one activity in which those classes are involved or must be part of a composition and the composition has an instantiating activity.

Postcondition: Domain class relations are instantiated via a domain activity
All class relations between domain classes must refer to at least one domain activity in which the relation is instantiated. This activity must have involvements for all the domain classes that are in the relation. In the case that one class is associated to two or more roles of the activity, the association must refer to the roles that result in the link of two objects. That activity must have a domain operation of the type “Link objects” in its activity model.

2.7.2.5 Make attribute view

Define the attributes of domain activities and domain classes.

Guideline: Each domain activity should have input attributes
Activities most likely change objects depending on properties of the activity itself. This is just a guideline, because participating in an action with another object, without explicit attribute changes, could also be a state change.

Guideline: Domain classes have attributes
Each domain class should have at least one attribute. If it doesn’t have an attribute it might be a domain class relation.

Postcondition: All object attributes get a value in an action
All domain class attributes must have at least one operation in at least one activity model that sets its value. If an attribute does not explicitly get a value in the instantiating activity of the class, then it has a default value, possibly defined by the attribute, but at least by its value type.

Postcondition: A strong domain class must have identifying attributes
A domain class is strong if its objects do not depend on the existence of another object (from another domain class in the same domain). As a result, a strong domain class must have identifying attributes, because an object from a strong class cannot be identified via other domain objects, because it has no parents in the domain.

Postcondition: Attribute types are classes from a context
Referred classes of all attributes must be an element of a context. The classes must belong to the same domain in case it is a composition between domain classes. Otherwise, it is a class in a context.

2.7.2.6 Specify domain constraints

Add invariants to domain classes, domain class relations and attributes. This step is typically done at the end of domain engineering.

Guideline: Only use invariants if other concepts cannot cover it
Use the modeling concepts (other than the constraints) as intended, and only apply an invariant if another modeling concept cannot or should not be used to specify something. For example, do not use a constraint to restrict the possible order of actions on an object, when it can be modeled in the object lifecycle. Or, do not use a constraint to separate a path from other paths in the object lifecycle of a domain class, if that domain class should be split in two domain classes.

Guideline: Do not skip the invariants
The modeling language part that is formed by the metamodel structure, i.e., the modeling concepts and their relations, does not cover every possibly relevant aspect of a domain. It is very likely that you have to model some domain properties in the form of a domain invariant.

2.7.3 Engineer feature

A feature is engineered by working in parallel on the feature structure, function lifecycles, and function signatures.

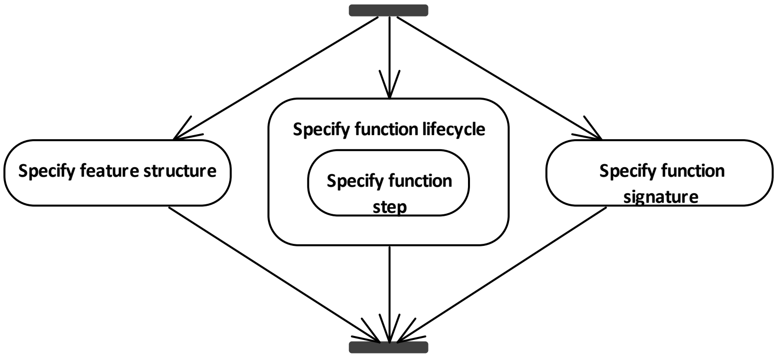

Specify feature structure
Specify function lifecycle
Specify function step
Specify function signature

Figure 1: Engineer feature

Guideline: 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 positioned into 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 any actor that performs the function.

Guideline: Define a function for behavior that will be assigned to an actor.
Define functions for units of behavior that will be assigned to and performed by an actor.

Guideline: 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.

Guideline: 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.

Guideline: Find functions from system specifications
System functions can be found from three 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 indicate often 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.

2.7.3.1 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 resulting tree structure has the feature as a root.

Guideline: 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.

Guideline: 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,

Guideline: 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 positioned into 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.

Guideline: Find functions via reoccurring behavior
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 function.

Guideline: 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.

2.7.3.2 Specify function lifecycle

During this step, the control flow of each function is specified. This means describing the order of the steps in the function. The function steps refer to sub-behaviors of the function. The sub-behaviors are references to domain activities, operations, or functions of a specification space that the function's feature depends on. A lifecycle view typically has arrows for the control flow, as well as for specifying which function attributes participate in each function step. When these arrows cross each other too much, the view may get messy and the lifecycle view could be split into two views: 1) a view just the order of the steps and without any function attributes related to them, i.e., just the control flow, and 2) a view in which is specified which function attributes are participating in which step.
Another option is to use a textual notation for the function step, like with a regular programming language, where a function call contains the link between variables and the actual parameters of the function call.

Guideline: 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 participating in a function steps.

Guideline: 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.

Postcondition: Check if all function events 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.

Postcondition: Function flow consistent with domain model
For all actions in a function lifecycle, i.e., function steps that are an instance of a domain activity:

- The objects linked to the action, must be an instance of an involved domain class.
- The input attributes of the action must be parameterized by a function attribute.
- The function flow must not violate the OLC of an involved object.

Figure 9: 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 have to be specified:

- The function attributes, which must belong to the same function or a container function, have to be allocated to (the actual parameters of) the function steps.
- The preconditions for this step, i.e., the constraints on the step participants. A condition is typically expressed in a logic 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, are an instance of 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.

Guideline: 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.

Guideline: 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 don't need to be allocated manually.

Guideline: 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.

2.7.3.3 Specify function signature

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.

All function signatures can be put in a single diagram, 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 superfluous because MuDForM follows a whitebox perspective on function specifications. Though, a postcondition could help to guide the design of the function lifecycle.

Guideline: Begin and end with function signature
Make a preliminary signature of the function before specifying the function lifecycle, but do not spend too much time on it. Finalize the signature after the function body is complete (lifecycle complete, constraints & decisions complete). Check for any "free attribute", i.e., an attribute that is not a property of an object that participates in one of the function steps, nor is computed from such properties. A free attribute can only come from the outside and must be an input attribute of the function.

2.7.4 Manage specification space dependencies

Define the dependencies between the different specification spaces. Define a dependency from P to Q (the dependency structure of P has a reference to Q) if P has elements that refer to elements of Q.

Guideline: Only define dependencies that are used
Do not define a dependency from specification space P to specification space Q, if nowhere in P a reference to an element of Q is set.

Postcondition: Only refer to elements of specification spaces that you depend on
Specification elements in the specification declarations of the specification space may only refer to specification elements in the specification spaces that this specification space is dependent on.
If an element in specification space P refers to an element from specification space Q, then P must be dependent on Q. If the dependency is not intended, then the element form Q may not be used in P.

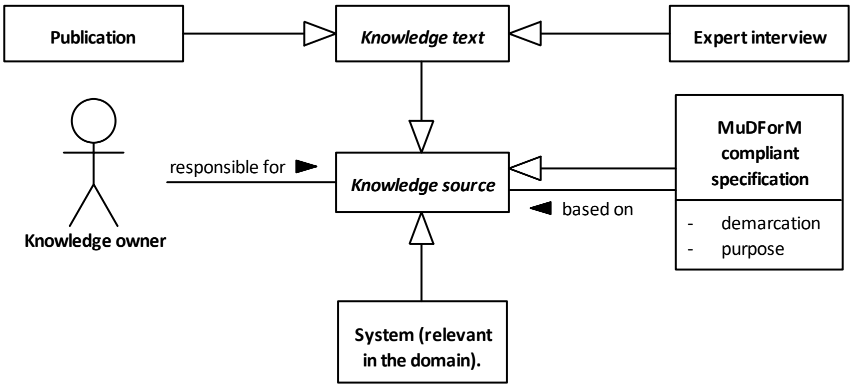
Postcondition: No cyclic dependencies
The graph of all dependencies between specification spaces may not contain cycles. If a cycle appears, it is most likely that a specification space must be split, because it contains parts that differ in abstraction or aggregation level.

3 Discovery domain

The Discovery domain contains the concepts that are used to gather input (text) and analyze it, in order to acquire an initial MuDForM compliant model. It is the basis for the definition of the steps Scoping, Grammatical analyssis, and Text-to-model transformation of the MuDForM method flow. It also contains concepts to record analysis decisions.

3.1 Knowledge containers (static view)

A modeling process may start with knowledge sources with relevant content, on which the targeted model will be based. In practice, this can be a text from a publication or an interview with a domain expert. In some cases an existing system (domain system) is used to extract domain knowledge from.



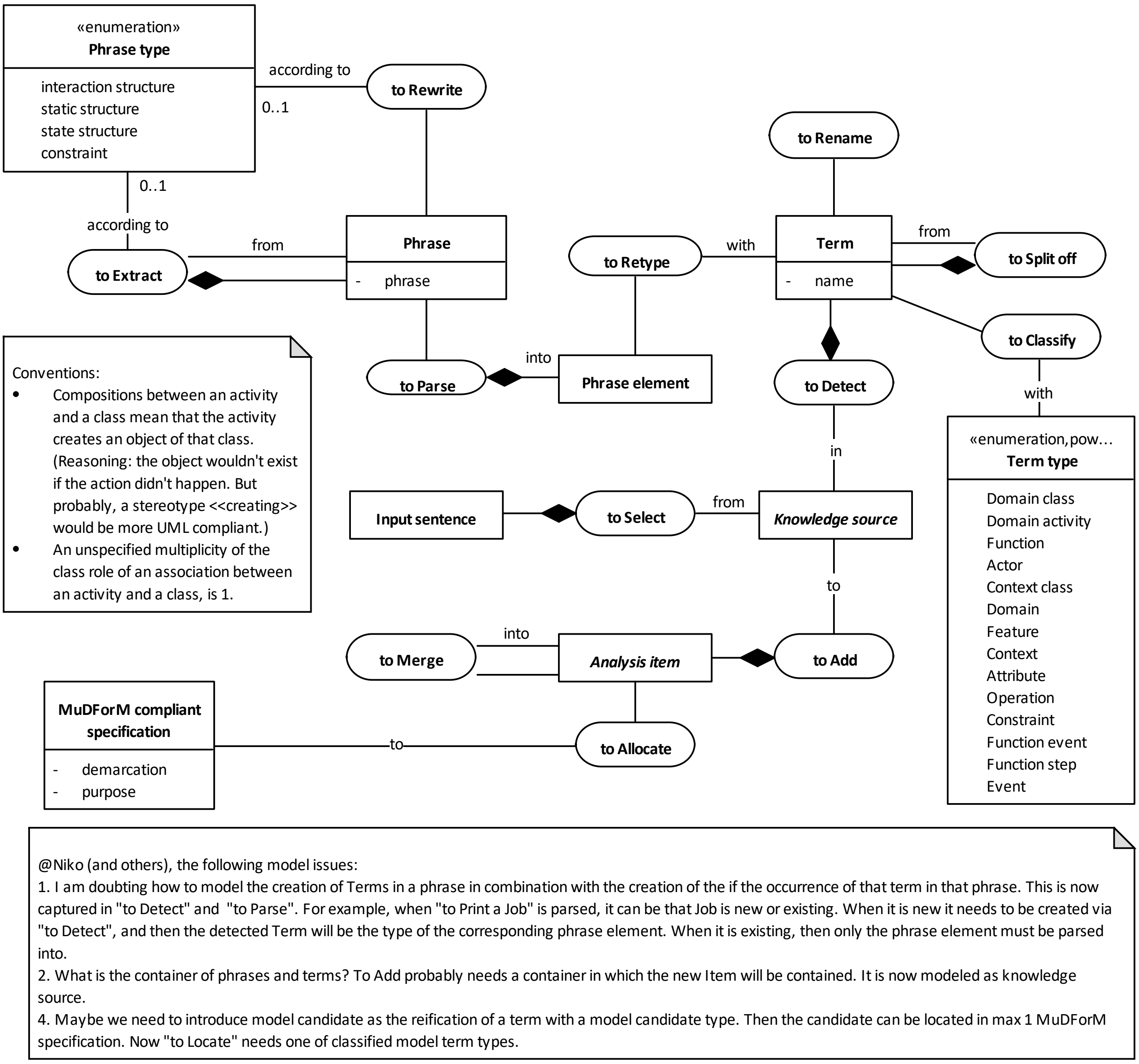
Knowledge containers (static view)

3.2 From text-to-model (interaction view)

This diagram presents the activities and related classes that are needed to conduct the method steps Grammatical Analysis, and Text-to-model transformation. A set of source sentences is Selected from a Knowledge source. Phrases are Extracted from other Phrases, which can be Source sentences. Phrases can be Extracted or Rewritten according to one of the Phrase types.

A Phrase can be Parsed into Phrase elements, in which each Phrase element is typed with a Term. If a Term did not exist yet, it is first Detected in the scope of the Knowledge source. Later, due to new insights, a Phrase element can be (re)Typed with another Term, typically, because of a detected homonym or synonym. Terms can also be Renamed. A new Term can be Split off from an existing Term, mostly when an existing Term has two different meanings and a new Term is needed to separate two meanings. During analysis, it is also possible to add new Analysis items, which can be a Term or Phrase. One Analysis Items can be Merged into another Analysis Item.

To go from text to model, Terms will be classified with one or more model term types, and all (relevant) Analysis items will be located in a MuDForM specification (context, domain, or feature).



Conventions:

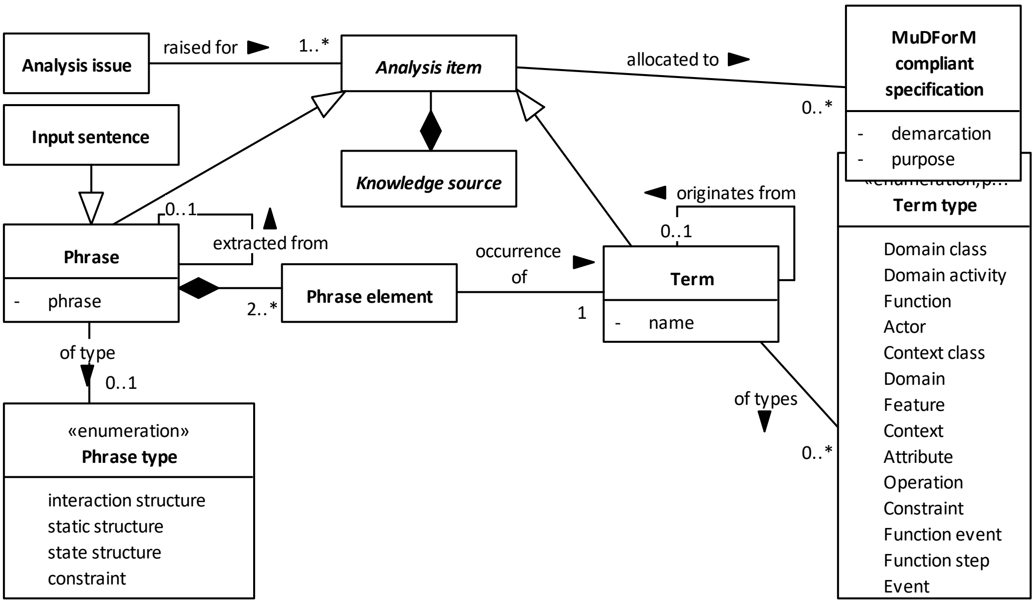
- Compositions between an activity and a class mean that the activity creates an object of that class. (Reasoning: the object wouldn't exist if the action didn't happen. But probably, a stereotype <<creating>> would be more UML compliant.)
- An unspecified multiplicity of the class role of an association between an activity and a class, is 1.

@Niko (and others), the following model issues:
1. I am doubting how to model the creation of Terms in a phrase in combination with the creation of the if the occurrence of that term in that phrase. This is now captured in "to Detect" and "to Parse". For example, when "to Print a Job" is parsed, it can be that Job is new or existing. When it is new it needs to be created via "to Detect", and then the detected Term will be the type of the corresponding phrase element. When it is existing, then only the phrase element must be parsed into.
2. What is the container of phrases and terms? To Add probably needs a container in which the new Item will be contained. It is now modeled as knowledge source.
4. Maybe we need to introduce model candidate as the reification of a term with a model candidate type. Then the candidate can be located in max 1 MuDForM specification. Now "to Locate" needs one of classified model term types.

From text-to-model (interaction view)

3.3 From text to model (static view)

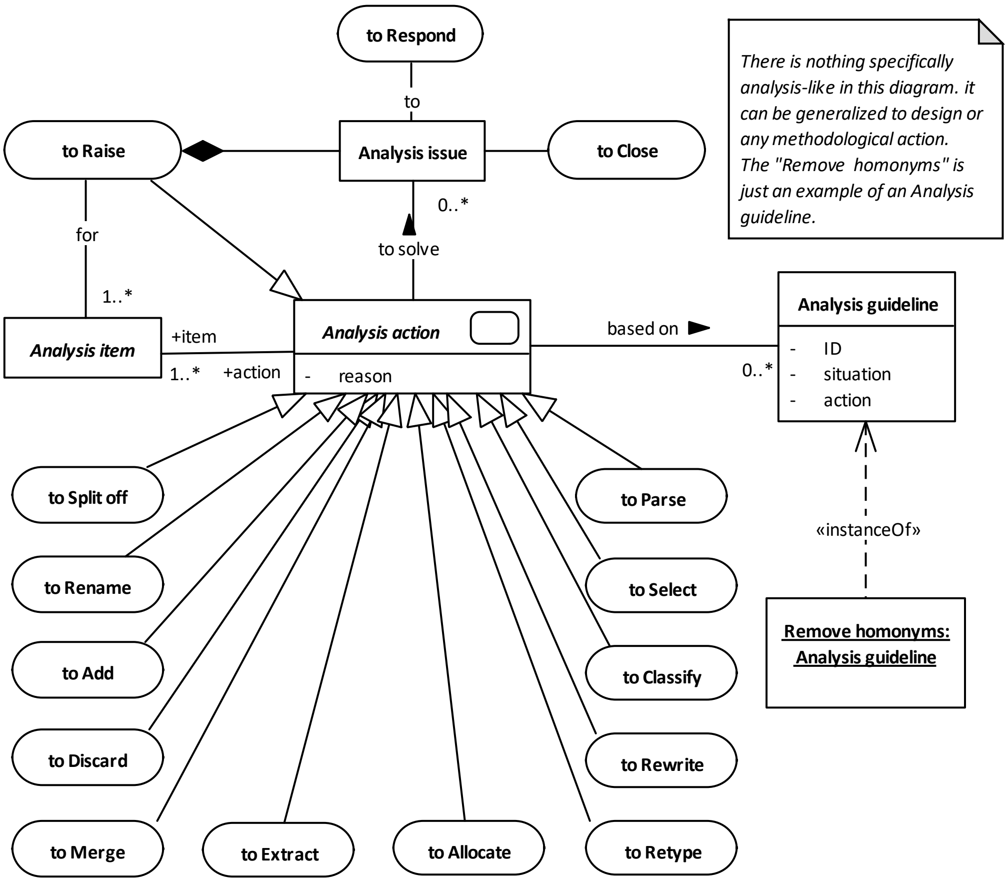
The Knowledge text contains Source sentences. After that, phrases are extracted from the Source sentences. An alternative approach is not to have an initial text, but to acquire knowledge more interactively. This could take place in a brainstorming session with a group domain experts. In such a session, one could already format the sentences according to the phrase types.
Due to discussions with the domain experts or other knowledgeable people, Phrases may be discarded and new Phrases may be introduced. After a set of phrases is obtained, Model candidates are identified. Per candidate a first indication of its type is determined. Phrases and Model candidates are analysis items.



From text to model (static view)

3.4 Analysis issues (interaction view)

Analysis issues can be Raised for Analysis items. Analysis actions act upon analysis items and can be based on Analysis guidelines. Analysis actions can be performed to solve zero or more Analysis issues. Analysis issues can be Closed.



*There is nothing specifically analysis-like in this diagram. it can be generalized to design or any methodological action.
The "Remove homonyms" is just an example of an Analysis guideline.*

Analysis issues (interaction view)

Element	Description

Knowledge owner	Person or organizational unit that is responsible for the contents of one or more knowledge sources.
Analysis issue	Issue raised for one or more Analysis items.
Expert interview	Documentation of the interview with an expert. Typically, the interviewed expert is the Knowledge owner of the interview.
Input sentence	Sentence that is contained in a source text and selected as input for the grammatical analysis. Most likely, an input sentence is not formatted according to one of the phrase types.
Knowledge source	A source with relevant knowledge. A knowledge source can have one or more actors (organization or person) that are responsible for the knowledge that is contained in the source.
Knowledge text	A piece of text with relevant knowledge, which can be used for grammatical analysis.
MuDForM compliant specification	A MuDForM compliant specification. The scope of a specification is addressed in the demarcation and the purpose. The demarcation lists what are the targeted concepts in the specification, and what concepts are out of scope. The purpose explains what the targeted application of the specification is. A MuDForM specification itself can also serve as a knowledge source. **attribute:** demarcation type: Demarcation is done from two perspectives: • Intrinsic: a set of terms that are expected to be in scope, or to be out scope. • Application: a list a set of uses cases, features, functions that the domain specification should be usable for. **attribute:** purpose type: What is the purpose of the specification? What is it used

	for?
Phrase	Phrase in natural language or according to a Phrase type. A Phrase is an Analysis item. A Phrase can be rewritten in one or more other Phrases. **attribute:** phrase type:
Phrase element	The occurrence of a term in a phrase.
Phrase type	Predefined format for a phrase which can be transformed into a piece of model. **attribute:** interaction structure type: Phrase expressing a change to one or more objects, and or subject. The format is: Subject **TO** verb object (preposition/indirect object)* or **TO** verb object (preposition/indirect object)* **Interaction structures** will end up in the model as relations between acclivities/operations/functions and classes. They 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. **attribute:** static structure type: Phrase that expresses a static relation. The format is: noun **HAS** noun OR verb **HAS** noun OR verb **HAS** verb Static structures typically end up in the model as attributes. **attribute:** state structure type: Phrase that 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 State structures typically end up in the model as specializations or as possible values for the type of an attribute. An example of the latter is the phrase "the car is blue" leads to an attribute "color" of the class "car", and that "blue is a possible value for "color".

	attribute: constraint type: Phrase that expresses some condition, typically written with operators of 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 before X seconds after B".
Publication	A (part of) a document, or document set, that contains text. Typically, one of the authors is the knowledge owner of the publication. But, it can also be someone who is acceptably knowledgeable about it.
System (relevant in the domain).	(Software) system that applies and exposes relevant knowledge about a domain or feature. This typically is visible by the use of domain terms in the design, (user) interface, code, or database of the application.
Term	A single word that is used in the analyzed text. This can be a noun (phrase), verb, proper name, adjective, adverb. A Term can be instantiated when parsing a phrase, or introduced in another analysis action. In the latter case, an introduced term can be based on an original term, in which case the originates-from association has an instantiated link. A term can be typed with Model candidate types, which makes the term a candidate for the targeted model. **attribute:** name type:
Term type	Modeling concepts that are used in a MuDForM specification., MuDForM offers different types of specification elements. The type of specification space, i.e., domain, feature, or context, determines which types of specification elements are allowed, and what is their semantics. The three different specification spaces all have concepts to specify state, concepts to specify change, and concepts to specify the relation between state and change. Besides the concepts that are specific for a type of specification space, almost all specification elements can have **attributes** and **specializations**, and have **constraints** attached to them. We now list the specification elements that can be the input of the model engineering step, and thus the output of the grammatical and text-to-model transformation. Each of the specification elements occurs as a possible phrase type or a possible term type in the grammatical analysis. **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. • **Domain classes** define what objects can exist in a

	domain. They are elements for the creation of compositions and serves as the types of function attributes. Instances of domain classes are objects with an interesting life. • **Attributes** define properties of Domain Classes or Domain Activities. Each attribute refers to a class form a context of the domain. **Feature** models can contain the following concepts: • **Functions** are behavior elements. They specify what must happen when the function is active. • A function can use other behavioral elements, which can be other functions, domain activities, and operations. The usage of a behavioral element in a function structure is called a **Function event**}. Some function events are not performed by the function, but are interactions with behavior outside the function. Such events are 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. • Function lifecycles describe the control flow of the function's behavior in a process algebra style, i.e., in terms of sequence, selection, parallelism, and iterations of **Function steps**, which are typed by a function event. **Context** models contain specification elements that do not belong to the scopes of the targeted domains and features, but that are needed to specify the elements in those domains and features. There are typically two kinds of context elements: • **Context classes** represent either physical quantities like length, time, or speed, or concepts whose definition is not determined by the owners of the domains and features of interest, like Name, Address, Phone number. • **Operations** to inspect and change instances of context classes, like an operation to determine the postal code of an address, or converting inches to centimeters, or just to divided distances by time. • Concepts related to the interaction of features, such as external **actors** or **events**. **attribute:** Domain class type: **attribute:** Domain activity type: **attribute:** Function type: **attribute:** Actor type: **attribute:** Context class type: **attribute:** Domain type: **attribute:** Feature type: **attribute:** Context type:

	attribute: Attribute type: **attribute:** Operation type: **attribute:** Constraint type: **attribute:** Function event type: **attribute:** Function step type: **attribute:** Event type:
Analysis guideline	Guideline for analysis actions. **attribute:** ID type: **attribute:** situation type: A description of the situation in which the guideline is applicable. **attribute:** action type: A description of the analysis (or design) action that should be done on the involved items.
Analysis item	Item that is the subject of grammatical analysis. An Analysis item can undergo Analysis actions. An item can be located in a MuDForM specification, when it has a proper type, i.e., it has a phrase type of a Term type.
Remove homonyms	When one term occurs has different meanings in different phrases, a new term should be split off and get a different name.
to Add	To introduce a new Analysis Item. This might happen because of new insights of the domain expert, possibly triggered by an open issue.
to Allocate	To position an Analysis item in a MudForM model. This means that you determine in which domain, context, or feature the item will be put. Only Terms that are classified with at least one term type can be located. Only phrases that are formatted according to a (model) phrase type can be modeled.
to Classify	To assign a Model candidate type to a Term. A Term can have more types.

to Close	To state that an issue is solved.
to Detect	To find a new Term in the analyzed Knowledge Source.
to Discard	To state that an Item is not of interest anymore.
to Extract	The state a new phrase based on an existing phrase that has more phrases in it. The extracted phrase is part of the same knowledge source that the original phrase is part of.
to Merge	To unite two Analysis items into one. Typically it gets the name of one of the merged items, for example when two Terms are considered to be synonyms, or when two active phrases are considered the same, but they do not have all the same Terms as Phrase elements.
to Parse	to Identify the terms in a phrase, which results in one or more phrase elements that have
to Raise	To open an issue for one or more analysis items.
to Rename	To give the Term a new name, i.e., Term.name gets a new value.

to Respond	To comment on an issue, e.g., provide a solution suggestion, or asking for clarification.
to Retype	To replace the existing term of a phrase element with another term.
to Rewrite	To rewrite a phrase such that it matches one of the Phrase types.
to Select	To indicate that a sentence form a knowledge source will be analyzed.
to Split off	To create a new term from an existing term, most likely because the existing term was a homonym. Typically, this is followed by retyping some of the phrase elements from the existing term to the new term. New term.origninates from.Term := from.Term New term.name is given a value
Analysis action	Activity that is the placeholder for all actions that change the set of analysis items in some way. The reason explains why the action is done. An analysis decision might be taken based on a guideline and/or to address some issue. **attribute:** reason type: A justification for the action.

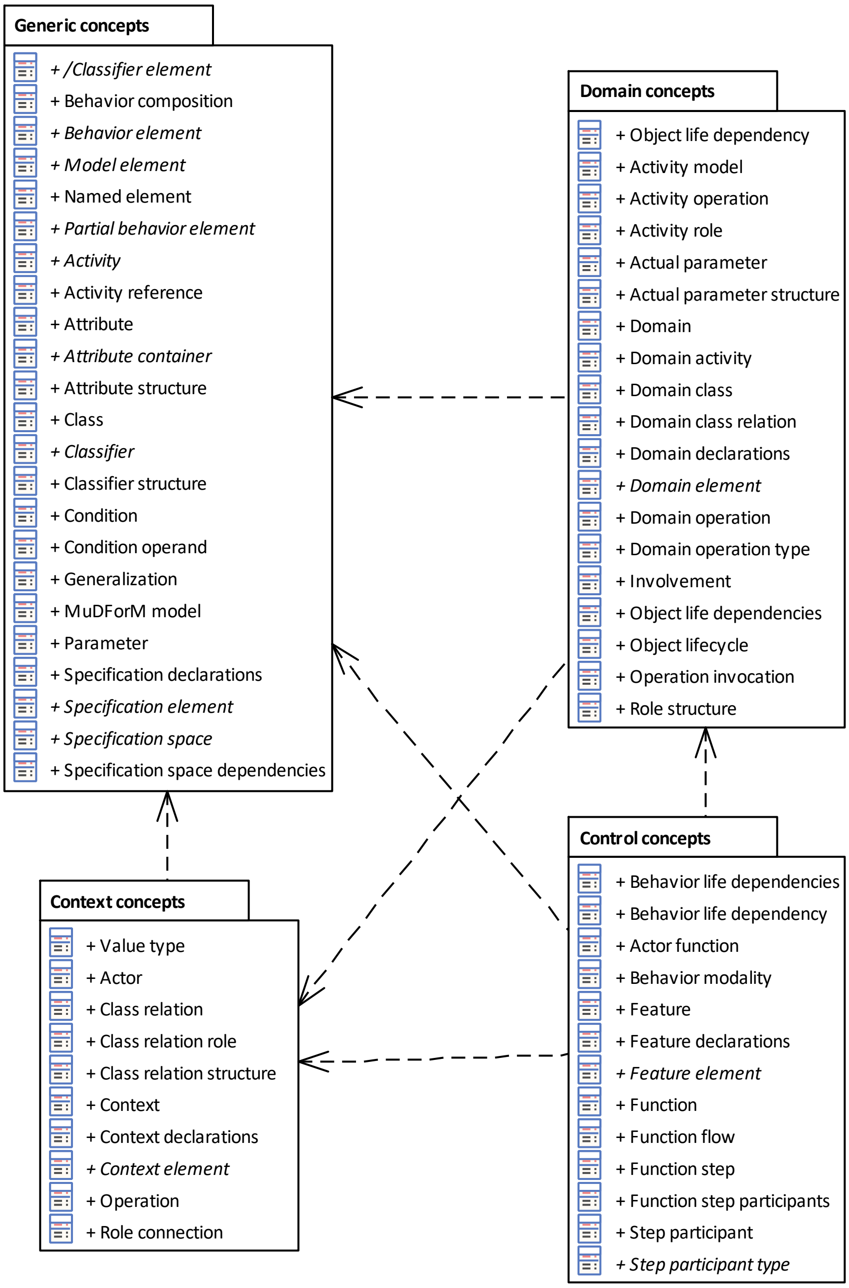
4 MuDForM modeling concepts

Modeling concepts are the concepts for defining a MuDForM compliant model. The major modeling concepts (like value type, domain class, function) are the specification elements that can be declared in a specification space. Each major modeling concept is defined by a set of cognitive aspects that are modeled via one of the modeling constructs. The sub concepts are the ones that are created in a structure of a major concept (like attribute, involvement, function event).

4.1 MuDForM modeling concepts

The modeling concepts are captured in four packages:

- The generic concepts, which are used as a basis for the concept in one of the other packages, or are used in all of the other packages.
- The context concepts are used in context models, and can be based on generic concepts.
- The domain concepts are used in domain models, and can be based on generic concepts and context concepts.
- The control concepts are used in feature models and can be based on generic concepts, context concepts, and domain concepts.



MuDForM modeling concepts

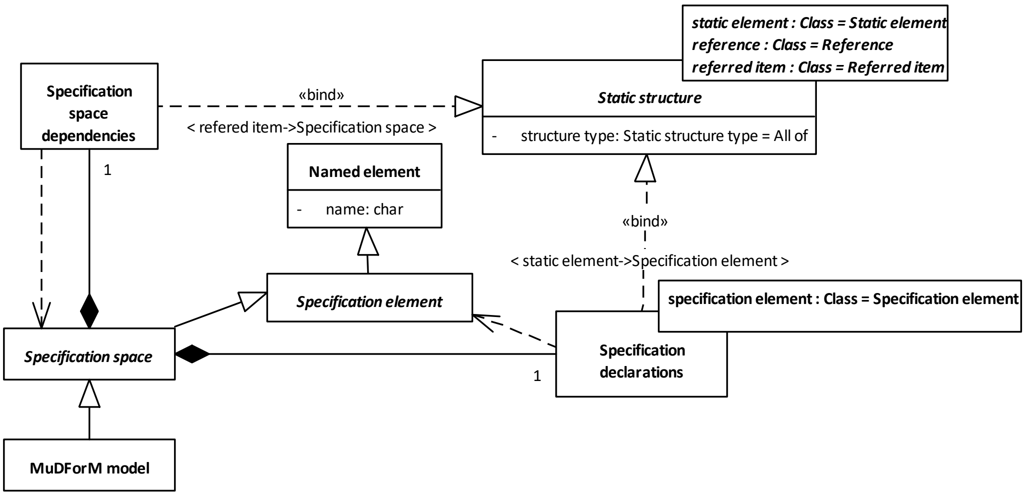
4.2 Generic concepts Package

This package defines the concepts that are used in the three different types of specification spaces, i.e., contexts, domain, and features. The package also contains concepts that form the basis for the definition of concepts defined in the other packages.

4.2.1 Specification Spaces

A specification space contains declarations, which is a static structure of specification elements. A Specification space also contains dependencies, which is a static structure of references to other specification spaces. The specification elements in the declarations of a specification space may only refer to specification elements in specification spaces that this specification space depends on (similar to the include-construction in many programming language).
There are four types of specification spaces:

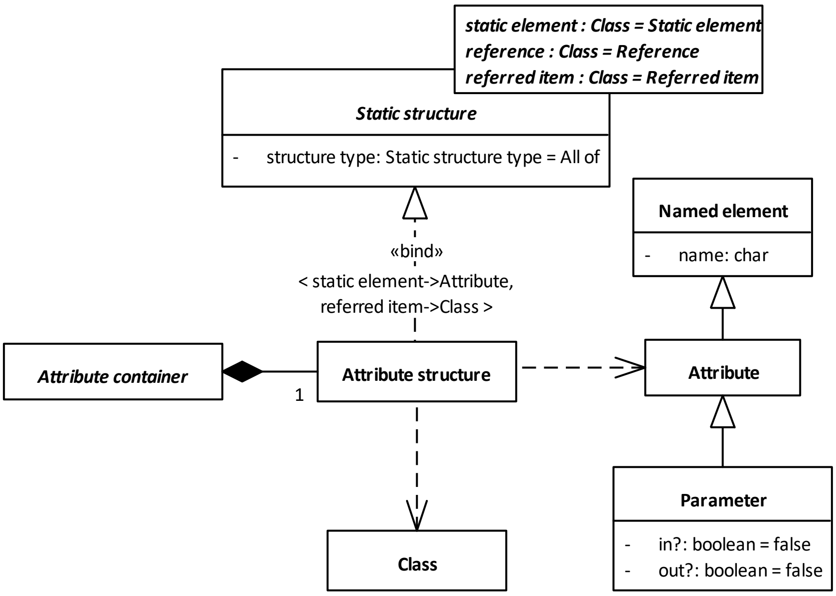
- MuDForM models contain contexts, domains, and features. A MuDForM model forms the root of a specification. It doesn't contain other elements besides specification spaces.
- Context models contain concepts that have a clear meaning without defining their internal properties.
- Domain models contain classifiers that define what lives (objects with state), called domain classes, can exist, and in which changes (actions), called domain activities, those lives may be involved.
- Feature models may depend on one or more domain models, other features, or contexts. A feature defines what instances shall exist and what changes shall take place in the domains that the feature depends on.


static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Specification space dependencies
«bind»
< refered item->Specification space >
Static structure
- structure type: Static structure type = All of
1
Named element
- name: char
«bind»
< static element->Specification element >
Specification element
specification element : Class = Specification element
Specification space
Specification declarations
1
MuDForM model

Specification Spaces

4.2.2 Attributes

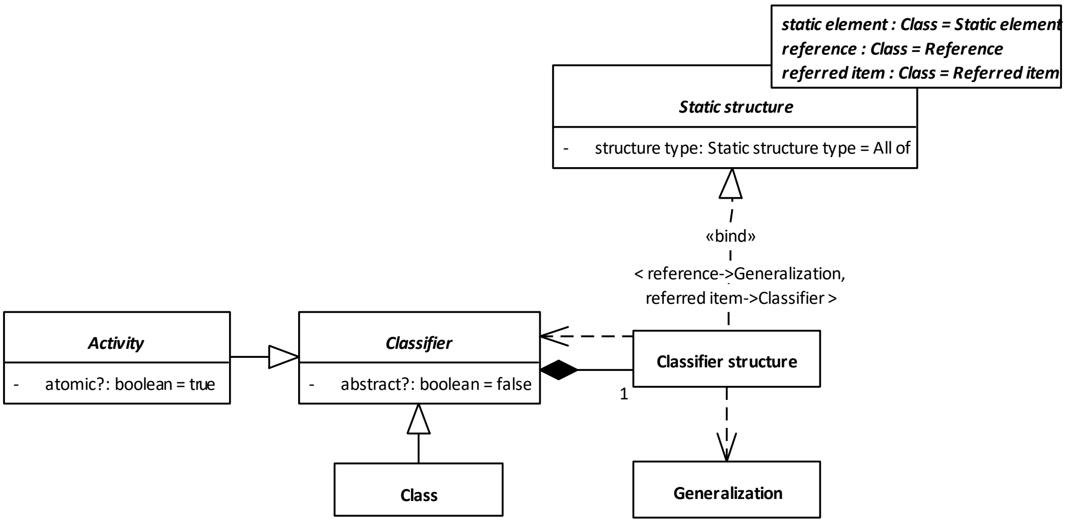
An Attribute container can have an Attribute structure. An Attribute structure consists of Attributes that refer to a class. Some Attributes are Parameters. Only behavior elements (Activities, Activity roles, Involvements) can have parameters.


static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Static structure
- structure type: Static structure type = All of
Named element
- name: char
«bind»
< static element->Attribute,
referred item->Class >
Attribute container
1
Attribute structure
Attribute
Parameter
- in?: boolean = false
- out?: boolean = false
Class

Attributes

4.2.3 Classifiers

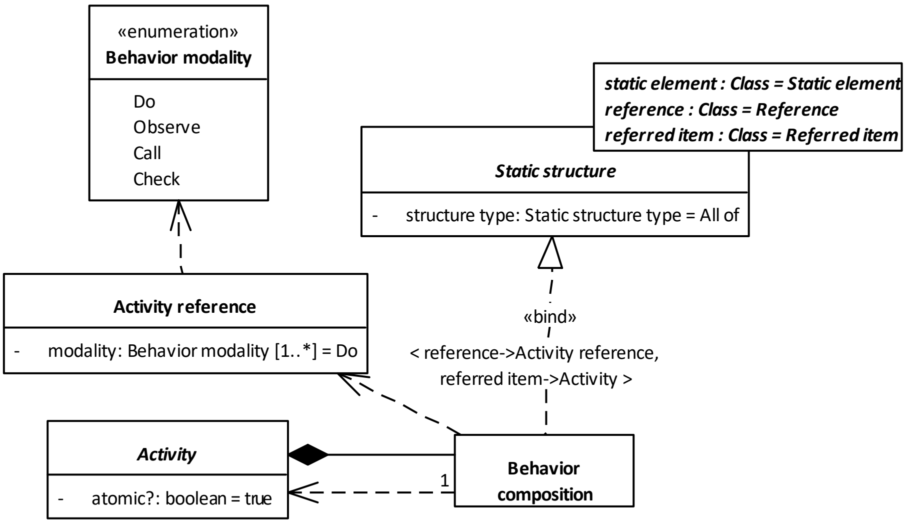
A classifier can have a classifier structure that expresses what the super classes are of the classifier. The super classes occur as generalizations in the classifier structure. Note that, in contrary to UML and inheritance in programming languages, the concept of generalization is specified on type level, but does not imply that it is always exist on instance level. This is because the classifier structure type can be "Some of"or "One of". For example, an abstract class Thief with a classifier structure of type "One of" has generalizations towards domain classes Person and to Crow. This means that a Thief is either a Person or a Crow. This also expresses that some Persons and some Crows are Thieves (but not all Persons or Crows). Whether a Person is a Thief or not might also vary over time.


static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Static structure
- structure type: Static structure type = All of
«bind»
< reference->Generalization,
referred item->Classifier >
Activity
- atomic?: boolean = true
Classifier
- abstract?: boolean = false
Classifier structure
1
Class
Generalization

Classifiers

4.2.4 Behavior composition

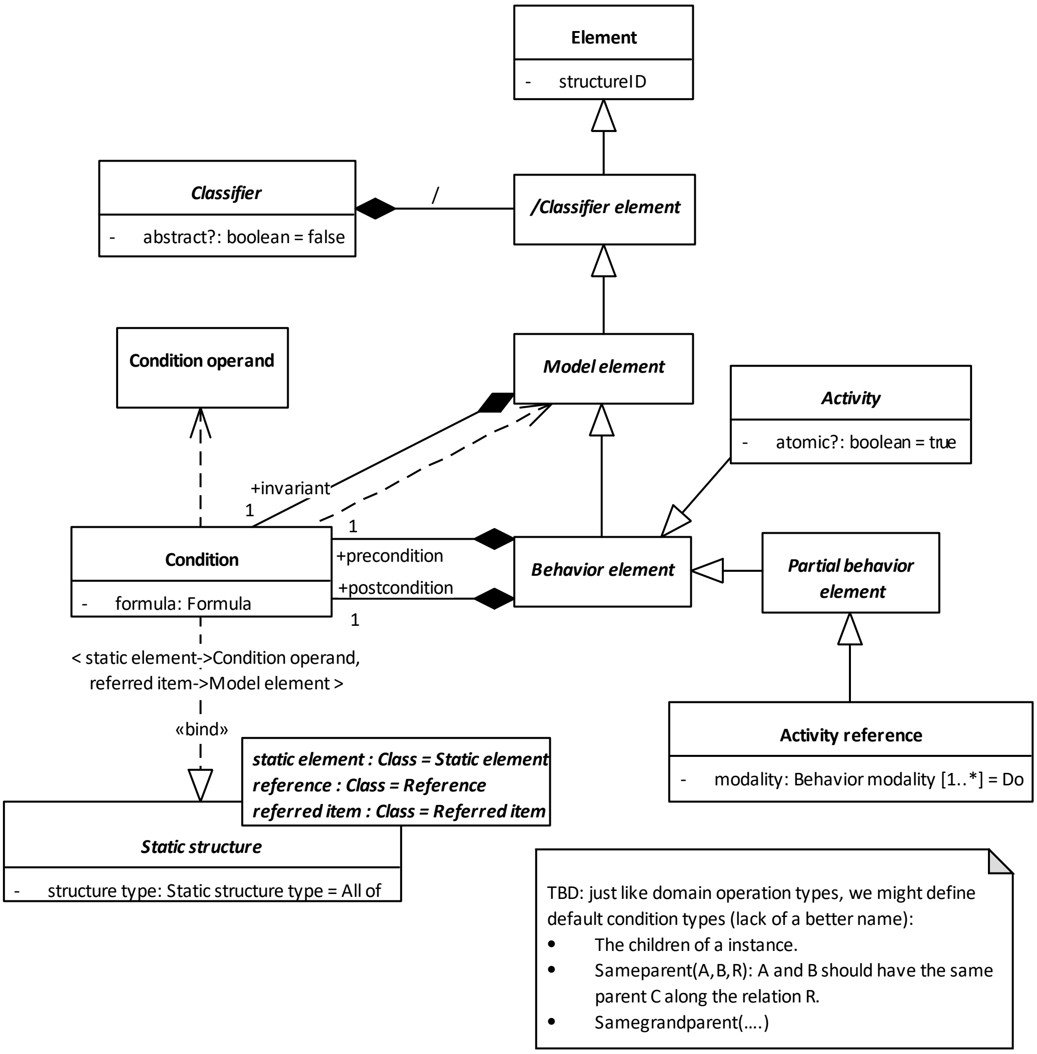
An Activity has a Behavior composition, which is a Static structure in which the references are Activity references, and the referred items are Activities. A Behavior composition specifies which activities are used in the execution of the containing activity. The Behavior modalities of an Activity reference specify how the referred Activity is used by the containing Activity.


«enumeration»
Behavior modality
Do
Observe
Call
Check
static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Static structure
- structure type: Static structure type = All of
Activity reference
- modality: Behavior modality [1..*] = Do
«bind»
< reference->Activity reference,
referred item->Activity >
Activity
- atomic?: boolean = true
1
Behavior composition

Behavior composition

4.2.5 Conditions

Any element in any structure can have conditions. A condition is a structure where the elements are Condition operands. A Condition operand is bound to an Conditionable element. That element must be part (possibly recursively) of the Conditionable element that the Condition holds for. Any Identifiable element can have invariants. Conditionable behavior elements can also have a precondition and a postcondition.


Element
- structureID
Classifier
- abstract?: boolean = false
/
/Classifier element
Condition operand
Model element
Activity
- atomic?: boolean = true
+invariant
1
1
Condition
- formula: Formula
+precondition
+postcondition
1
Behavior element
Partial behavior element
< static element->Condition operand,
referred item->Model element >
«bind»
static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Static structure
- structure type: Static structure type = All of
Activity reference
- modality: Behavior modality [1..*] = Do
TBD: just like domain operation types, we might define default condition types (lack of a better name):
• The children of a instance.
• Sameparent(A,B,R): A and B should have the same parent C along the relation R.
• Samegrandparent(….)

Conditions

Element	Description
/Classifier element	Any Element of a Structure of a Classifier can be a conditionable element. As such, the Classifier elements of a Classifier are derived and defined by the union of all the elements in any structure that is contained by the classifier.
Activity	Any behavioral element that is executable, which means it can be a referred Item in an activity model or a function flow. Every activity can be seen as an event (type). That is why there is no separate modeling concept called event type. **attribute:** atomic? type: boolean Is the activity divisible or not? An atomic activity either happened, or did not happen. An non-atomic activity has a begin moment and an end moment, and as a results, its instances can be in a state between begin and end.

Activity reference	Behavior that an activity executes, reacts to, or calls. Activity references are declared in the same way as attributes. They are local within the scope they are declared and their type is an Activity. (Note that this differs from most languages that use the event concept, in which events are global and can be defined without any restriction to their scope). Activity references are expressed in terms of behavior elements; thus can be: • Function activation or de-activation. Every function has these two events. • Calling or executing an atomic domain activity. • Execution of a Operation. • A condition defined on some object attributes becomes true or false. This can be a timing condition. Notice that such a moment always takes place during the execution of a behavior element. (Disclaimer: this kind of activity reference might need some extra constraints to be modeled.) **attribute:** modality type: Behavior modality
Attribute	Property definition for instances of attribute containers. An attribute refers to a Class that defines the type of the attribute.
Attribute container	Abstract class for model elements that can have attributes.
Attribute structure	A static structure belonging to an attribute container, in which the static elements are attributes, and the referred items are classes. **constraint:** All Attributes are a Reference or a Substructure
Behavior composition	A Behavior composition is a Static structure, in which the references are Activity references, and the referred items are Activities. A Behavior composition specifies which activities are used in the execution of the containing activity. **constraint:** All Static elements are a Substructure or an Activity reference (=Reference).
Behavior element	Any element that defines an Activity or a part of an Activity can have a pre- and postcondition.

Class	A set of objects. A class is a Classifier and an Attribute container and can be part of a Class relation.
Classifier	Classifier is an abstract class. A classifier is something that has instances. A classifier can have a classifier structure that expresses the generalizations of the classifier. **attribute:** abstract? type: boolean An abstract classifier does not have elements that only belong to the classifier. They must always belong to at least one of its specializations (sub classes) or generalizations (super classes) as specified by a classifier structure.
Classifier structure	A static structure of a classifier, in which the references are generalizations, and the referred items are classifiers. **constraint:** All Static elements are a Substructure or a Generalization (=Reference)
Condition	A Static structure in which the static elements are Condition operands, and the referred items are Model elements. A Condition states what must be true for its container. All Model elements can have invariants. Conditionable behavior elements can have a precondition and postcondition. A condition is specified by a formula. That formula is specified in a formalism that is not part of MuDForM. **constraint:** All Condition operands are a Substructure or a Reference **attribute:** formula type: Formula specification of he formula that expresses the condition. It should be a boolean expression.
Condition operand	The use of a Conditionable element in a condition, i.e., the condition operand is bound to the Classifier element. Example: Invariant of person: person.age > 0. The Condition operand person.age in this condition is bound to the Classifier element person.age.
Generalization	Reference to a classifier in a classifier structure of a classifier, meaning that the classifier is a subclass of the referred classifier.
Model element	Any element that is part of a structure, i.e., is a (classifier) element, can be identified, and as such can have invariants. An invariant states what always must be true for the element.

MuDForM model	This is the root of a complete MuDForM compliant specification. A MuDForM model contains contexts, domains, and features. Typically, it contains at least one of each type. *Side note: The root domain is not contained in an any domain. Mmm, what to do for this exception? Philosophy: as long as the universe is infinite, a domain is always part of another domain. But the latter domain might just not be modeled. So here we need to take the finiteness of a model into account. This means that syntactically: the Void is the root of all domains. It is the only domain that is a domain element of its own domain elements declarations itself. (Nice example of an infinite loop). Check: does this also hold for Context and Feature.* **constraint:** The specification elements in a specification declarations of a MuDForM model must be a specification space.
Named element	**attribute:** name type: char
Parameter	A Parameter is an Attribute of the Attribute structure of a Activity. An input parameter must get a value when the Activity is instantiated. An output parameter might get a value during the execution of the instantiated Activity. Note: a "local" parameter is just an attribute, meaning that "in?" and "out?" are both false. *TBD: there will be nuances for attributes that get a value during the execution of the activity. For example, function attributes that get a value during the reception of a function step that is activated from outside the function.* **attribute:** in? type: boolean Does the parameter get a value form the environment of the behavior element? **attribute:** out? type: boolean Does the behavior element give a value to the parameter?
Partial behavior element	Behavioral element that is part of an activity. It is not executable, but it can have an explicit precondition and postcondition.
Specification declarations	Static structure belonging to a specification space, in which the static elements are specification element. The declarations contains all the specification elements that are declared in this specification space. **constraint:** Specification declarations do not consist of References

Specification element	Specification elements are things in a specification space, which have there own autonomous identity in that space, i.e. they are not contained like an attribute, generalization, or activity role. Which types of elements are allowed depends on the concrete subclass of the Specification space.
Specification space	Container for specification elements. The specification elements will be in the declaration structures of the sub classes of specification space.
Specification space dependencies	A static structure belonging to a specification space, in which the referred items are specification spaces. There are some restrictions to the dependencies: • 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. **constraint:** All Static elements are Substructures, or References to a Specification space

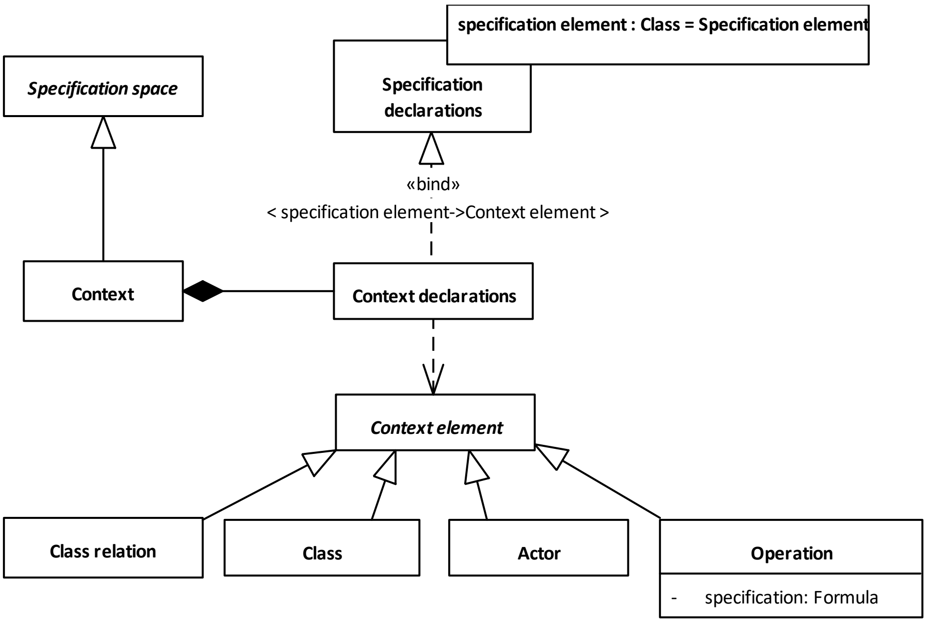
4.3 Context concepts Package

This package defines the concepts that can be contained in a context model.

4.3.1 Context definition

A context is a specification space in which the elements are actors, classes, value types, class relations, and operations. The elements may be referred to in the definition a specification elements of dependent specification spaces, domains and features in particular.

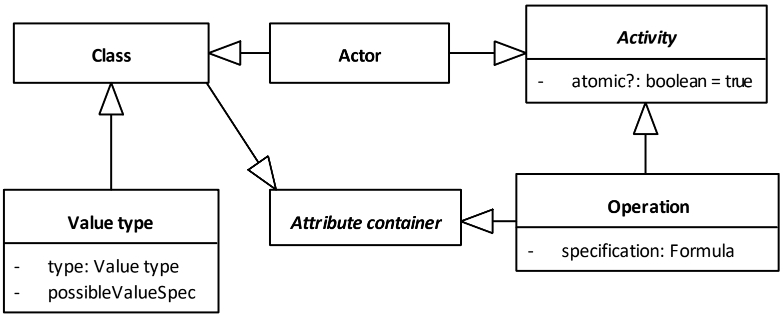
Contexts contain concepts that have a clear meaning without defining their internal properties. A context model typically contains two types of concepts. Firstly, physical concepts like length, time, power, speed, and their relations. Secondly, concepts from a domain that you do not want to model, but that you want to refer to. In this category you define classifiers and constraints for their identifying terms. Think of Name, Address, Phone number. You need these type of concepts to define specification elements in domains and features, but the internal specifications of these context elements are not of interest, and, thus, not specified. For example, you are typically not interested how an address changes over time.



Context definition

4.3.2 Context-specific concepts definition

This diagram shows the specific relations that the context elements have with other modeling concepts. An Actor is a Class and an Activity. A Value type is a class who's instances are just represented with a single value, like a number or a sting, or a single value from an enumeration of values. An Operation is an activity, for which the behavior is specified in a formula.

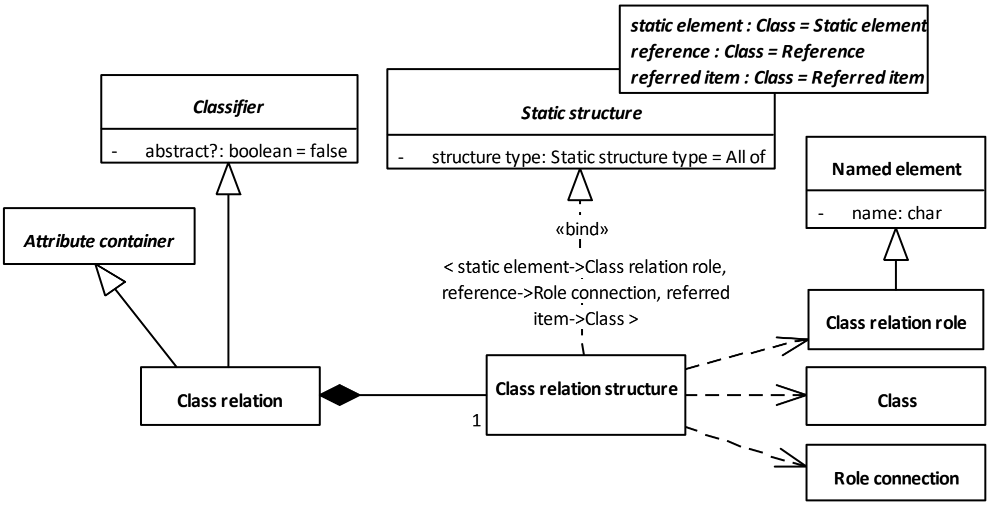


Context-specific concepts definition

4.3.3 Class relations

A class relation has a class relation structure in which the static elements are Class relation roles, the references are Role connections, and the referred items are Classes. Class relations are used when there is not a clear containment/composition relation between two classes.
MudForM distinguishes different kinds of relationships between classes:

- association: the result of a shared activity or coming into the domain as an aggregate. This is the one we mean with Domain class relation.
- lifecycle dependency: modeled via the life dependency structure of a domain class.
- composition: end of life of parent makes children inactive. This is the opposite of an object coming into scope with parts attached. E.g., you buy a car with wheels and you sell a car with wheels. This is very common in the physical world.


static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Classifier
- abstract?: boolean = false
Static structure
- structure type: Static structure type = All of
Named element
- name: char
Attribute container
«bind»
< static element->Class relation role, reference->Role connection, referred item->Class >
Class relation role
Class relation
Class relation structure
1
Class
Role connection

Class relations

Element	**Description**
Actor	Class that can check, execute, observe, and control (call) activities, which is specified in the Contained behavior of the Actor. *TBD: allocate functions to actors, and actors have capabilities. These capabilities can be specified in the context.*
Class relation	A relation between two or more classes in which none of the class can be seen as the container of the relation. Namely, in that case the relation is an attribute in the attribute structure of the container class. Class relations are used when one wants to express and communicate a relation between a number of classes. A Class relation is an Attribute container and a Classifier. A class relation is a attribute container. The attributes of a class relation are always immutable (otherwise the class relation would be a domain class).
Class relation role	A role of a class relation to which classes can be connected.
Class relation structure	A Static structure of a Class relation, in which that static elements are Class relation roles, references are Role connections, and referred items are Classes. **constraint:** All Class relation roles (=Static elements) are a Substructure or a Role connection (=Reference)

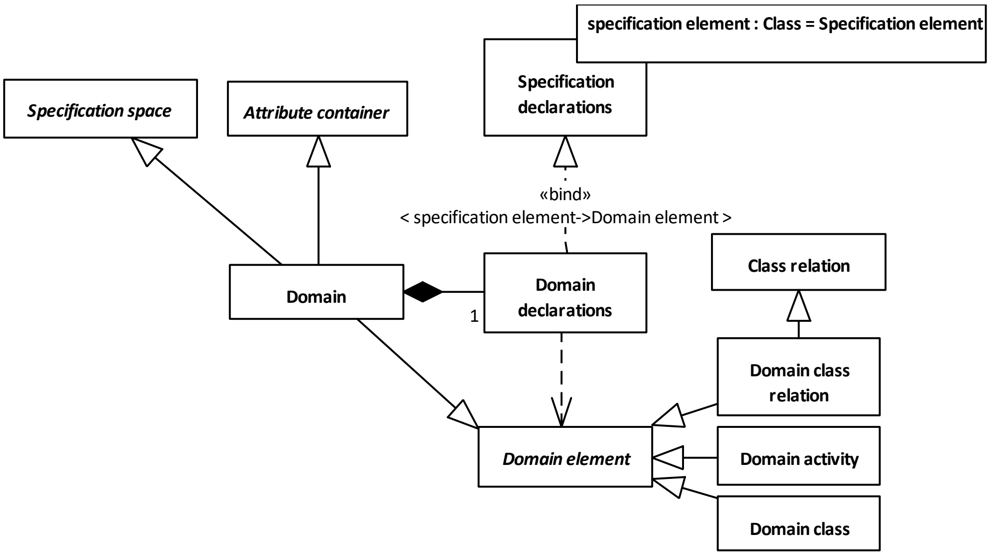
Context	A Specification space that contains elements that you need to define elements of other specification spaces, especially domains and features. Contexts form the boundaries of the feature and domain specifications. In a Context you define the things that are either defined somewhere else, or things you do not want to model in detail, i.e., the things that are not in your domain-of-interest or feature-of-interest. Typically these context elements are either concepts that you assume trivial, like datatypes and their operators, or systems (actors) that you want to use because of their capabilities.
Context declarations	A specification declaration structure belonging to a context, in which the specification elements are context elements.
Context element	Specification element in a Context declarations.
Operation	An operation defines an atomic change expressed by a formula. An operation has operands which can be input for the operation and/or output of the operation. **constraint:** atomic?=true **attribute:** specification type: Formula Specification of the formula in some Formalism.
Role connection	A Role connection defines that instances of the referred Class can participate the Class relation role instance of instances of the Class relation.
Value type	A Value type is meant for objects that just represent a single value, like a name, a phone number, or a length. The type of the value should be defined by some formalism, like a string, a fixed length string of digits, or a positive real. **attribute:** type type: Value type The type of the values of this value type. That type is typically from one of the used formalisms. **attribute:** possibleValueSpec type: A specification of the possible values of this Value type, e.g., a range of values, or an enumeration.

4.4 Domain concepts Package

this package defines the modeling concepts that are contained in a domain.

4.4.1 Domain definition

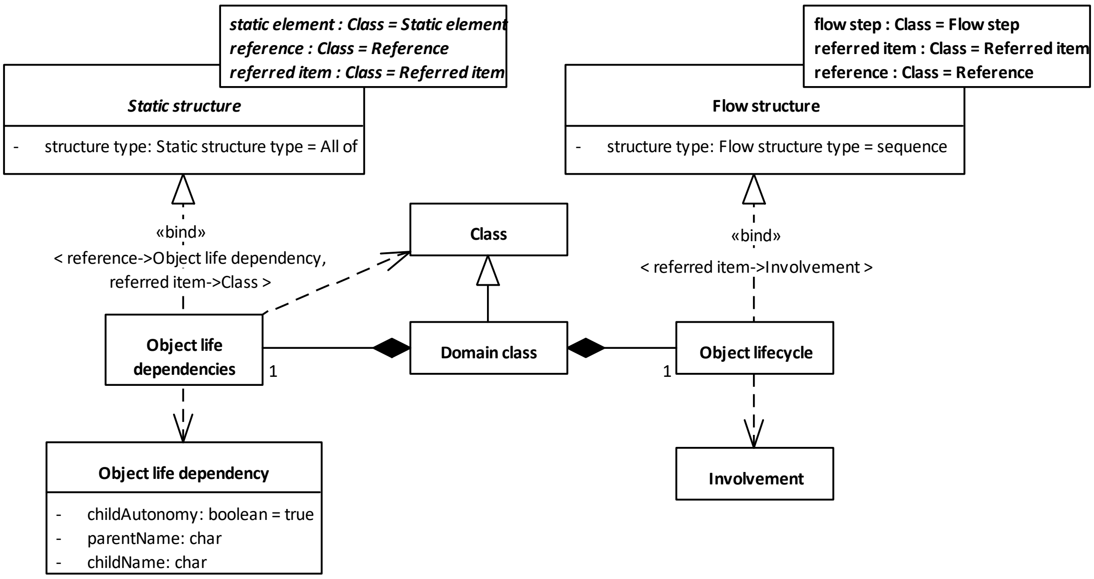
A Domain is a Attribute container and contains Domain elements, declared in the Domain declarations. Domain, Domain Class, Domain activity, and Domain class relation are Domain elements.
Domains contain classifiers that define what objects (with state) can exist, defined by domain classes, and in which actions (changes) those objects may be involved, defined in domain activities.



Domain definition

4.4.2 Domain class definition

A Domain class is a class of which the objects follow a defined Object lifecycle, which is a flow structure of involvements of the domain class. A Domain class can also contain a life dependency structures, which is a static structure with references to classes. Such a reference means that the instances of the domain class can not be alive without the referred objects in the life dependencies. This also means that a referred object cannot be deleted if it is a context object, or must be alive when it is a domain object.

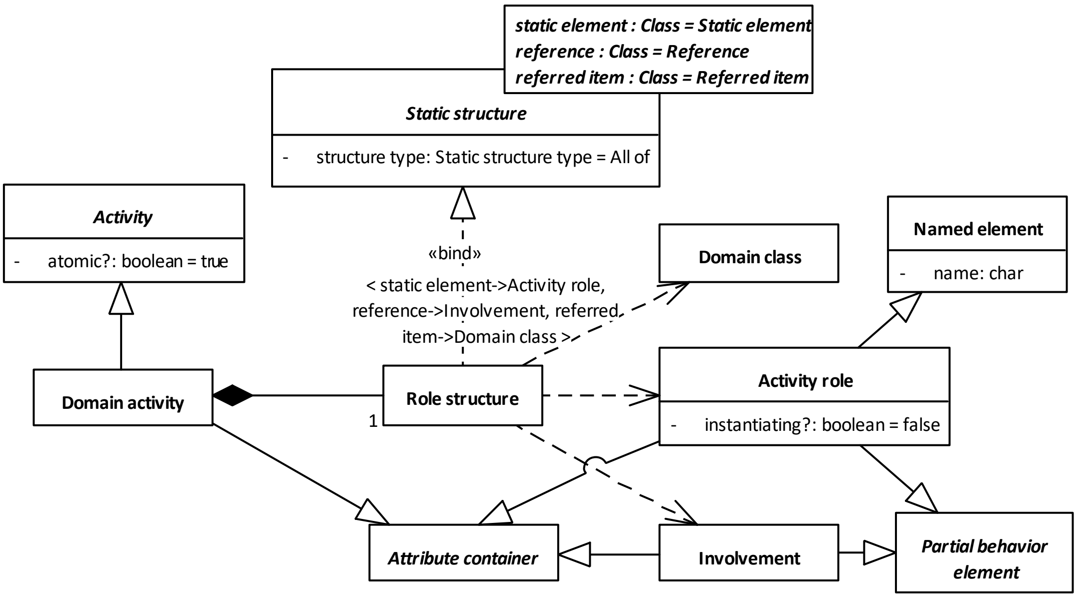

static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Static structure
- structure type: Static structure type = All of
flow step : Class = Flow step
referred item : Class = Referred item
reference : Class = Reference
Flow structure
- structure type: Flow structure type = sequence
«bind»
< reference->Object life dependency,
referred item->Class >
Class
«bind»
< referred item->Involvement >
Object life dependencies
1
Domain class
1
Object lifecycle
Object life dependency
- childAutonomy: boolean = true
- parentName: char
- childName: char
Involvement

Domain class definition

4.4.3 Domain activity definition

A domain activity has a Role structure. A Role structure is a Static structure in which the static elements are Activity roles, the references are Involvements, and the referred items are Classes. Domain activities, Activity roles, and

Involvements are Attribute containers. Activity roles and involvement are Partial behavior elements, which means they can a separate pre- and postcondition.

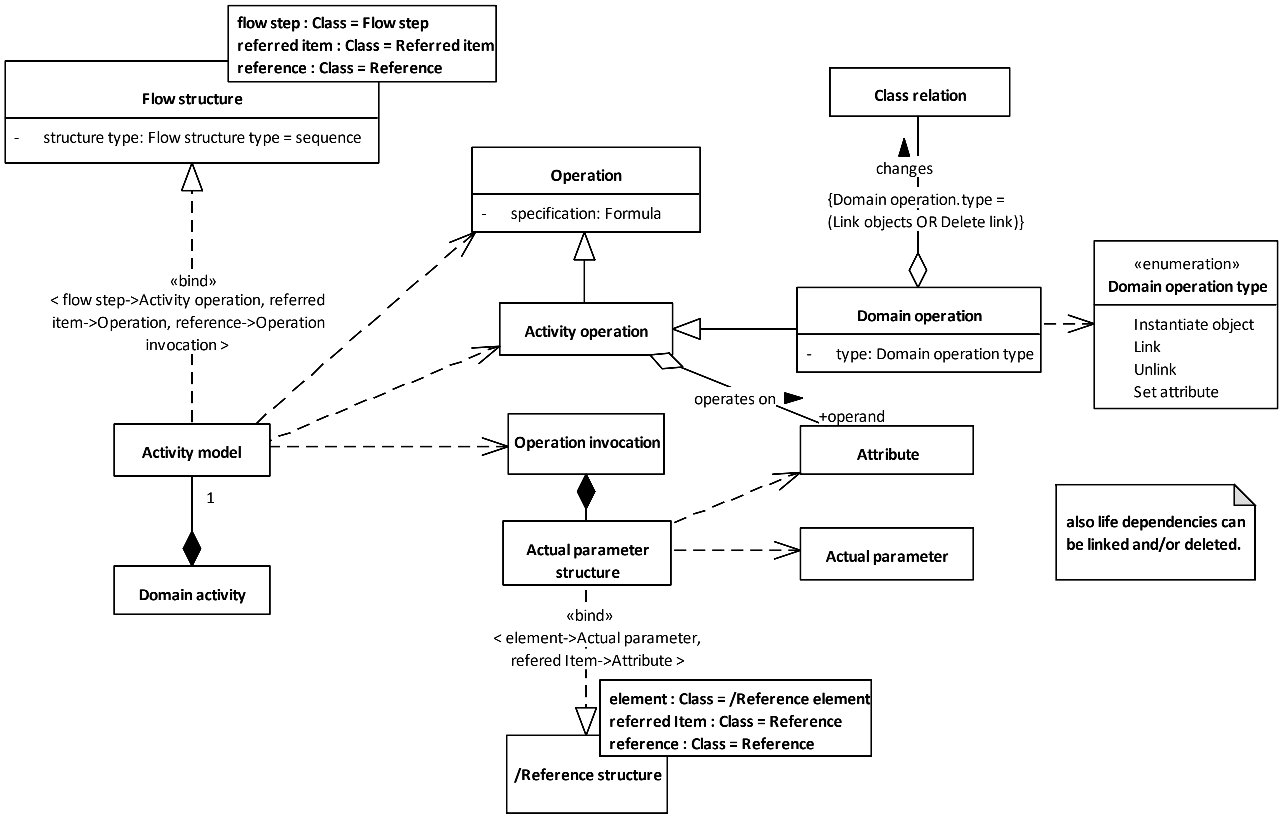

static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Static structure
- structure type: Static structure type = All of
Activity
- atomic?: boolean = true
«bind»
< static element->Activity role, reference->Involvement, referred item->Domain class >
Domain class
Named element
- name: char
Domain activity
Role structure
1
Activity role
- instantiating?: boolean = false
Attribute container
Involvement
Partial behavior element

Domain activity definition

4.4.4 Domain activity model

A domain activity may have an activity model that expresses the behavior of the activity. The behavior consists of operations to set attributes, to initialize objects, and to create or break relations between involved domain classes of the activity. Operations can be locally defined in an Activity Operation or it can be an Operation invocation of an Operation from a Context.

An operation invocation has a Actual parameter structure, in which the actual parameters of the invocation are parameterized by Attributes. These attributes must be in the scope of the Domain activity.


flow step : Class = Flow step
referred item : Class = Referred item
reference : Class = Reference
Flow structure
- structure type: Flow structure type = sequence
Class relation
changes
{Domain operation.type = (Link objects OR Delete link)}
Operation
- specification: Formula
«enumeration»
Domain operation type
Instantiate object
Link
Unlink
Set attribute
«bind»
< flow step->Activity operation, referred item->Operation, reference->Operation invocation >
Activity operation
Domain operation
- type: Domain operation type
operates on
+operand
Activity model
Operation invocation
Attribute
1
also life dependencies can be linked and/or deleted.
Actual parameter structure
Actual parameter
Domain activity
«bind»
< element->Actual parameter, refered Item->Attribute >
element : Class = /Reference element
referred Item : Class = Reference
reference : Class = Reference
/Reference structure

Domain activity model

Element	**Description**
Activity model	A flow structure belonging to an activity, in which the flow steps are activity operations, references are operation invocations, and the referred items are operations. An activity model express the internal behavior of an activity, i.e., what happens when the activity is executed.
Activity operation	An operation that is defined in the scope of an activity model. The behavior of Activity operation is specified in terms of the supported formalisms.
Activity role	Role within an activity in which domain classes can be involved. An activity role must be involved by at least one role class in the domain. An activity role can have also attributes. Sometimes it makes more sense to connect some attributes to a role instead of to the activity. **attribute:** instantiating? type: boolean Is an object instantiated in this role?
Actual parameter	Placeholder for the objects that must be "fed" to an Operation invocation, aka the actual parameters of the Operation invocation. A step participant must be bound to a function attribute.

Actual parameter structure	Reference structure in which the elements are actual parameters, and the referred items are the attributes of the operation that is invoked. Derivation: the structure has an actual parameter for each attribute of the invoked operation, and the actual parameter refers to its attribute.
Domain	A Domain is an Specification space and an Attribute container. A domain is an area of activity or knowledge, that is managed as one. A domain contains domain elements that define the domain. A domain should have a scope description that helps to determine if a certain concept belongs to the domain.
Domain activity	Domain element that defines a unit of change in the domain. Domain activities have instances called actions. Actions are atomic by deault. A non-atomic activity has a start action and stop action at instance level.
Domain class	A class with instances that have an interesting life in the domain of the class. The life is described via the Object lifecycle.
Domain class relation	A class relation between domain classes. A domain class relation instance between two objects may be established by participation of those objects in one domain activity instance (action). And a domain class relation instance can be broken by participation in a domain activity instance. There is a strong similarity between non-atomic activity and domain class relation. Difference: an domain class relation be changed (created or deleted) by multiple activities, and thus has its own identity.
Domain declarations	A specification declarations structure belonging to a domain, in which the specification elements are domain elements.
Domain element	Specification element that is part of a domain.

Domain operation	An activity operation that is an invocation of a Domain operation type. **attribute:** type type: Domain operation type
Domain operation type	The possible types of Activity operations in an Activity model. **attribute:** Instantiate object type: The operation creates an object of the given type and sets its attributes to a default values and sets the initial relations for the object. **attribute:** Link type: Create a link between two or more objects given the Class relation. **attribute:** Unlink type: int Delete an existing link between two or more objects. **attribute:** Set attribute type: Give one ore more attributes a value.
Involvement	An involvement states that instances of a domain class may be participating in a specific role of an action (instance of domain activity). An involvement may have attributes that are specific to the involvement.
Object life dependencies	A static structure in which the referred items are classes. The life dependencies of a domain class express what type of parent objects an instance of the domain class must have. This can be classes from a context or domain that the containing domain is depending or from the containing domain itself. **constraint:** All Static elements are a Substructure or a Object life dependency (=Reference)
Object life dependency	An Object life dependency (a reference in the Life dependencies of a child domain class) means that the parent (referred item) cannot end its life if it has living children. The child object doesn't have meaning without the parent object. When the childAutonomy is true, then the parents's life cannot be ended before its children's lives are ended. E.g., An order always belongs to a customer. You cannot end the life of a customer when there are still living orders. If you could, the order would become meaningless. When the childAutonomy is false, then the life of a parent can be ended, but the then also the child cannot participate in an action anymore. **attribute:** childAutonomy type: boolean Indicates if the life of the child is ended when the life of the parent is ended. Autonomous (true) means the life of the parent cannot be ended if the life of one if its children has not ended yet. Contained (false) means that the child is not alive anymore when the life of its parent ends. (Being alive means that an object can still participate in actions). **attribute:** parentName type: char The role name of the parent, i.e., the containing class, in the dependency relation.

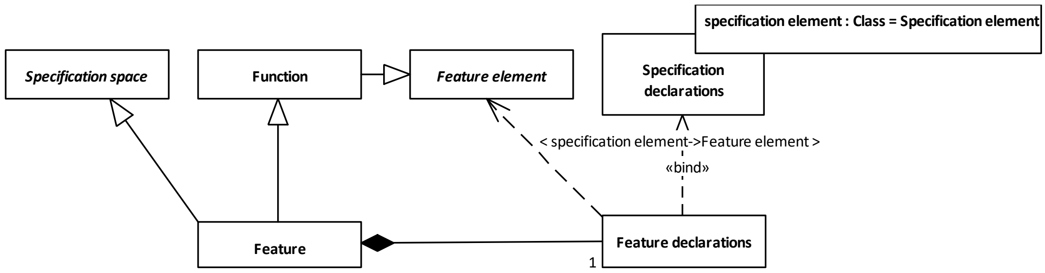
	attribute: childName type: char The role name of the child, i.e., the dependent/contained class, in the dependency relation.
Object lifecycle	Involvement flow that specifies in which order an object of a domain class may participate in related activity roles. Derived property: an object of a domain class is **Alive** when is still can do an involvement step according the OLC of the domain class. *We must consider if abstract classes have an OLC. But then it becomes an issue on how to integrate OLCs of multiple abstract classes. An idea could be to allow the role class appear as an involvement step. It could be that this way, parallel role classes or mainstream involvement steps need to be synchronized on an common involvement. (probably we need event synchronization anyway for parallel functions)* *The answer involves that an activity role occurs only once as involvement in a specialization structure, e.g., eat an apple is the same role as eat a piece of fruit. This is also logical, because if it are to OLC steps, then they refer to the same involvement, i.e., the involvement of piece of fruit in eat-d.o..* **constraint:** All Flow steps are a Substructure, or a Reference to an Involvement
Operation invocation	Invocation of an operation in an activity model. The operation must be defined in a context of the domain of the activity.
Role structure	Static structure belonging to an Domain activity, in which the static elements are Activity roles, the references are Involvements, and the referred items are Domain classes. The Role structure expresses which domain classes can and must be involved in an activity instance (=action). **constraint:** All Activity roles are a Substructure or an Involvement (=Reference)

4.5 Control concepts Package

This package defines the modeling concepts that are contained in a feature, aka control model.

4.5.1 Feature definition

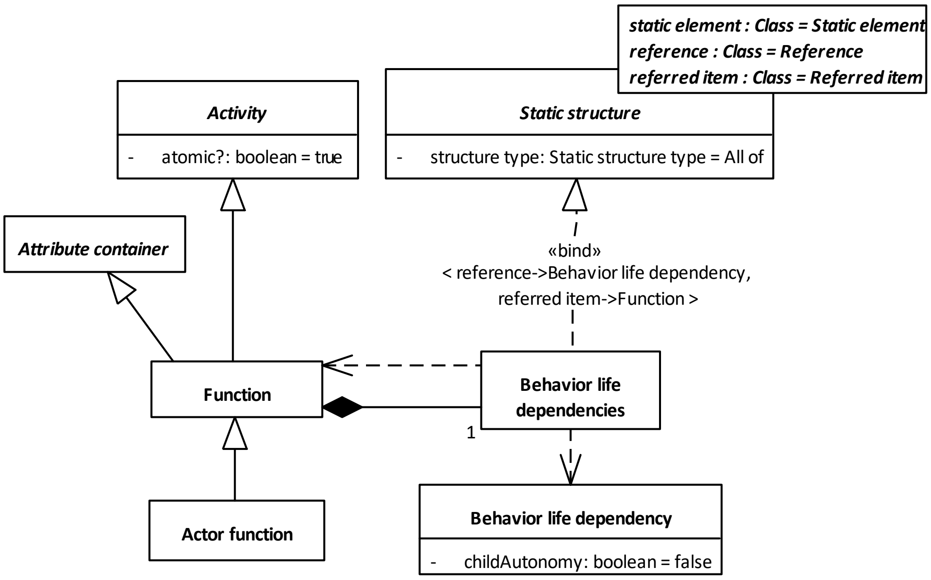
A feature is a specification space in which the specification elements are feature elements, i.e., functions. Feature models may depend on one or more domain models, other features, or contexts. A feature defines what instances shall exist and what changes shall take place in the domains that the feature depends on.


specification element : Class = Specification element
Specification space
Function
Feature element
Specification declarations
< specification element->Feature element >
«bind»
Feature
Feature declarations
1

Feature definition

4.5.2 Function definition

A function is an Activity and an Attribute container. A function has Behavior life dependencies which specifies in which other functions the functions should be executed.

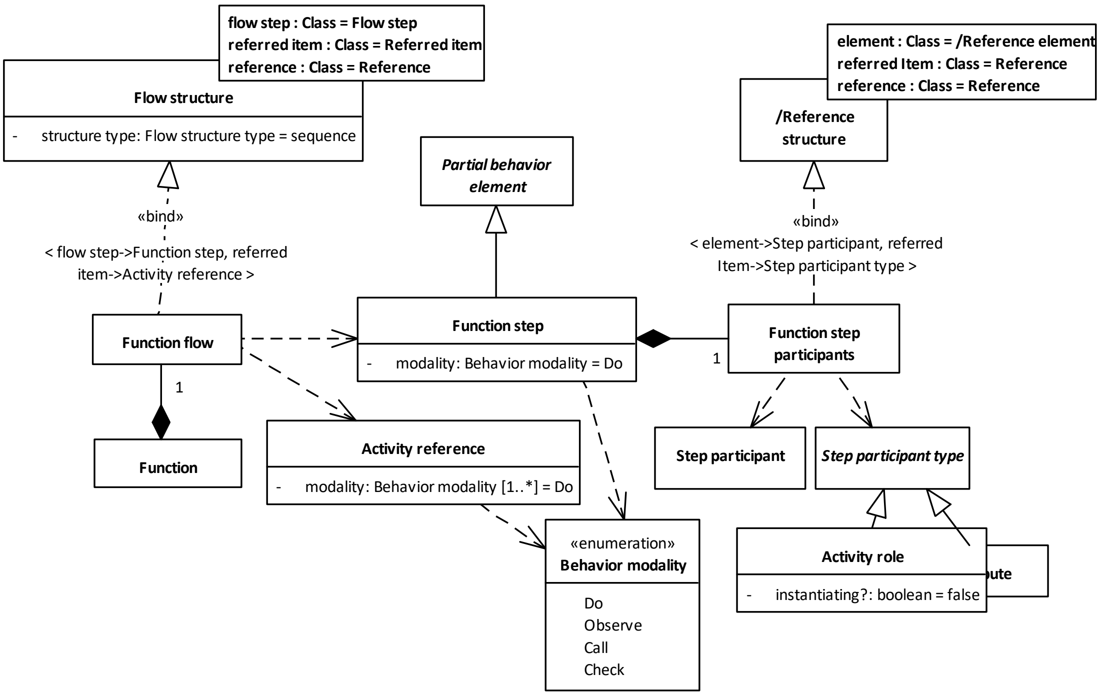

static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Activity
- atomic?: boolean = true
Static structure
- structure type: Static structure type = All of
Attribute container
«bind»
< reference->Behavior life dependency,
referred item->Function >
Function
Behavior life dependencies
1
Actor function
Behavior life dependency
- childAutonomy: boolean = false

Function definition

4.5.3 Function flow

A Function has a functions flow, which expresses what shall happen when the function is instantiated. The flow steps refer to Activities. The referred Activities can be:

- A domain activity in one of the domains that the feature of the function depends on.
- An operation in one of the contexts that the feature of the function depends on.
- A function from the same feature as the function or from one of the features the feature of the function depends on.


flow step : Class = Flow step
referred item : Class = Referred item
reference : Class = Reference
Flow structure
- structure type: Flow structure type = sequence
«bind»
< flow step->Function step, referred item->Activity reference >
Function flow
1
Function
Partial behavior element
Function step
- modality: Behavior modality = Do
Activity reference
- modality: Behavior modality [1..*] = Do
«enumeration»
Behavior modality
Do
Observe
Call
Check
element : Class = /Reference element
referred Item : Class = Reference
reference : Class = Reference
/Reference structure
«bind»
< element->Step participant, referred Item->Step participant type >
Function step participants
1
Step participant
Step participant type
Activity role
- instantiating?: boolean = false
ute

Function flow

Element	Description
Actor function	A function that can be assigned to an actor and can be invoked, canceled (devoked?), suspended, and resumed. *We need to consider concepts for invoke, devoke, suspend on actor level.*
Behavior life dependencies	A Behavior life dependencies is a Static structure in which the references are a Behavior life dependency, and the referred items are a Function. The Behavior life dependencies of a function state which function instantiations must exist in order for the function to be instantiated, the functions in which the life of the function is contained. A dependency also means that the function can access the attributes of the containing function. **constraint:** All Static elements are a Substructure or a Behavior life dependency (=Reference)
Behavior life dependency	A Behavior life dependency states in which container Functions a Functions must be executed. This means that the contained function can access the attributes of the container Function. When the child is not autonomous (childAutonomy=false), the termination of the container function execution also terminates the contained function. When the contained Function is autonomous (childAutonomy=true), the the container function cannot terminate its execution when the contained function is not finished yet. **attribute:** childAutonomy type: boolean

Behavior modality	The modalities that are possible for a function event. **attribute:** Do type: Execute the behavior. So, the actor that executes the function also executes the step. **attribute:** Observe type: React to the execution of this behavior in the environment of the function. In this case the Function must have a Function event that refers to the same behavior element as the Function step. **attribute:** Call type: Tell the environment to execute the behavior. In this case the Function must have a Function event that refers to the same behavior element as the Function step. **attribute:** Check type: Check if the behavior element instance can happen, according to all constraints that apply to this event. *(Disclaimer: the usefulness of this one is doubtful)*
Feature	A Feature is a specification space and a Function that operates in or more domains. A feature typically specifies a desired situation in those domains. A feature should be self-contained, i.e., has not implicit context dependencies, and hence, is reusable for other specifications. A Feature is a specifica
Feature declarations	Specifications declarations in which the specification elements are feature elements.
Feature element	Element in a Feature declarations.
Function	A Function is a Feature element. A function expresses what shall happen if the function is active.
Function flow	The control flow of a function, specified by a Flow structure in which the flow steps are Function steps and the referred items are Activity references.

Function step	Step in the Function flow of a Function. A Function step is a Reference to an Activity. **attribute:** modality type: Behavior modality The modality indicates what the actor that executes the function should do with the behavior that is referred by the function step.
Function step participants	A reference structure in which the elements are Step participants, and the referred elements are Step participant types. The functions step structure must be bound to Function attributes. Derivation: The Function step participants structure has a Step participant for each Attribute and activity role (if applicable) of the Behavior element that the Function step refers to, and the Step participant refers to that Step participant type.
Step participant	Placeholder for the objects that must be "fed" to a Function step, i.e., the actual parameters of the function step. A step participant must be parameterized with a function attribute.
Step participant type	Placeholder for the objects or attributes that are participating in a function step. In case the step refers to an Activity these can be Activity roles or activity attributes. In case the step refers to a function or an operation, these can be attributes.

5 MuDForM Viewpoints

This package defines the viewpoints in a MuDForM specification. Each viewpoint is defined by a constraint in terms of the metamodel. i.e., in terms of the MuDForM concepts, and a notation. The constraint says which modeling concepts are allowed in a view of that viewpoint. The notation says which symbols and layout conventions hold for the viewpoint. We distinguish the following viewpoints.

For a MuDForM model:

- Declarations view
- Dependencies view

For contexts:

- Declarations view
- Context structure
- Actors view

For domains:

- Declarations view
- Interaction view
- Class Structure view
- Attribute view
- Object lifecycle view per domain class
- Activity view per domain activity

For features:

- Declarations view
- Functional composition
- Function life cycle per function
- Function signature per function
- Context integration model (**Maybe move this one to the context model or integration model)**

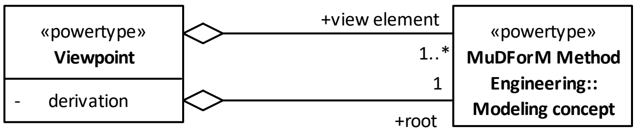
Integration model:

- TBD.
- Think of system use cases, where a system is (partially) defined by integrating a feature and a context.

The viewpoints are defined in a separate document called "MuDForM Viewpoints".

5.1 Viewpoints pattern diagram

A MuDForm viewpoint presents one or more view elements, which are modeling concepts. Each Viewpoint has one modeling concept as its root. The content of a view according to the viewpoint is specified by the derivation.



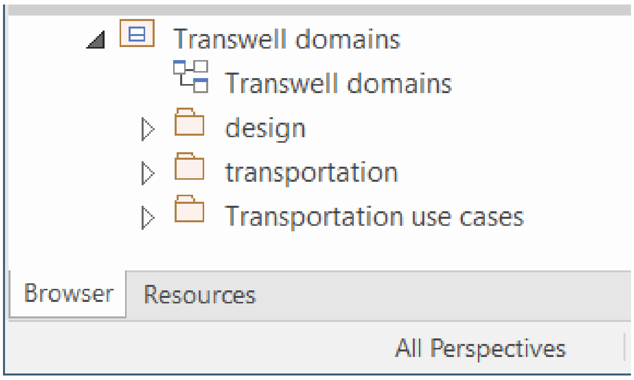
Viewpoints pattern

Name	Definition
Viewpoint	Type of view of a MuDForM compliant specification that shows a defined set of view elements. A viewpoint belongs to a root Modeling concept, and represents other modeling concepts (view elements). **attribute:** derivation Specification of the content of a view according to this viewpoint for a given root. The derivation explains which view concepts are visible in a view on the model for this viewpoint and the root.

5.2 For a MuDForM Model

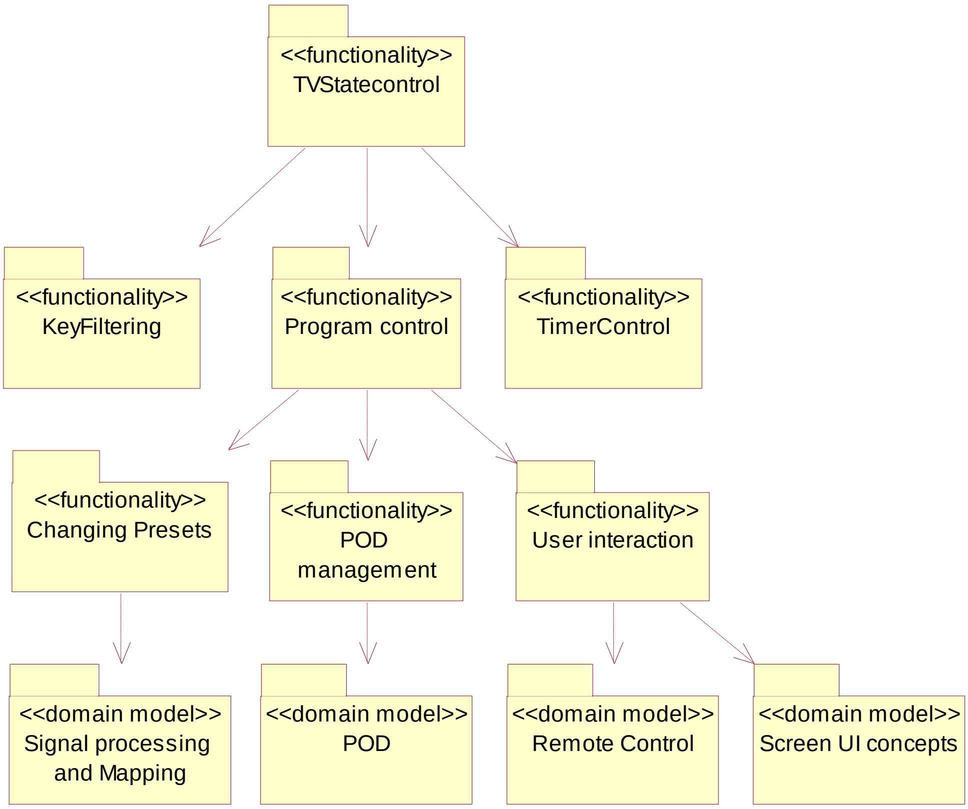
5.2.1 Declarations view:

- Root: MuDForM model
- Derivation: The specification elements in the specification declarations of the MuDForM model.



5.2.2 Dependencies view

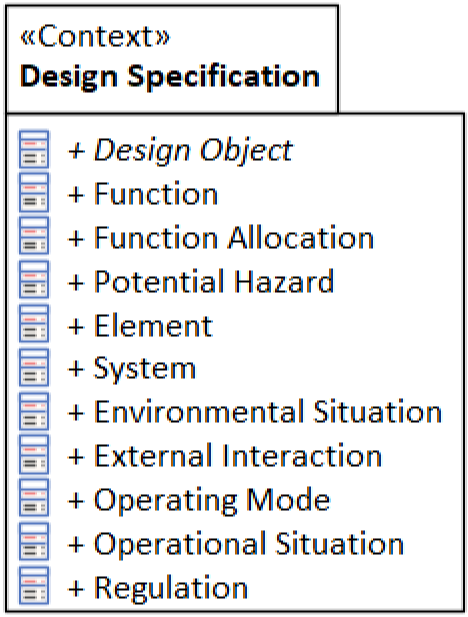
- Root: MudForM model
- Derivation: the references in the specification space dependencies of all the specification spaces in the specification declarations of the MuDForM model. So, show all the specification spaces contained in the MuDForM model, and show the dependencies between those spaces.



5.3 For a context

5.3.1 Declarations view

- ○ Root: Context
- ○ Derivation: The specification elements in the specification declarations of the context.



5.3.2 Context structure

- ○ Root: context

- Derivation:
  - The context elements in the specification declarations of the context.
  - The attributes in the attribute structure of the context elements.
  - The class relations, the class relation roles, and role connections of the class relations.
  - The generalizations in the classifier structure of the context elements.

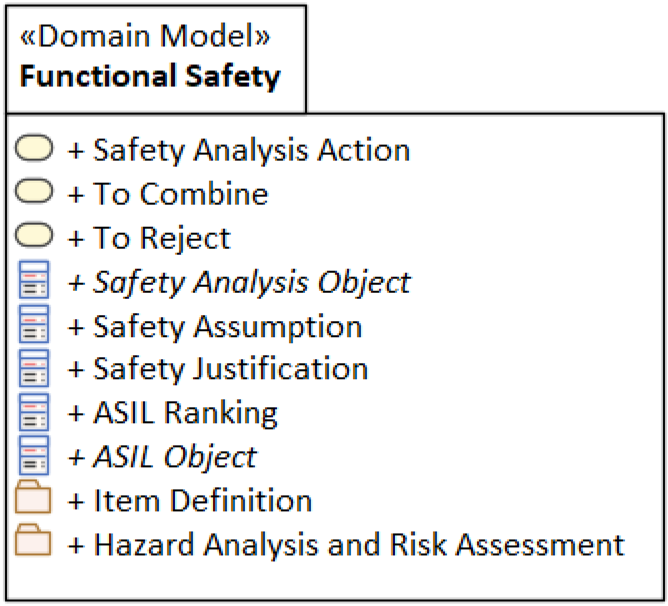
5.3.3 Actors view

- Root: context
- Derivation:
  - The actors in the specification declarations of the context.
  - Per actor the events it can react to or it can generate.

5.4 For domains

5.4.1 Declarations view

- Root: Context
- Derivation: The specification elements in the specification declarations of the domain.

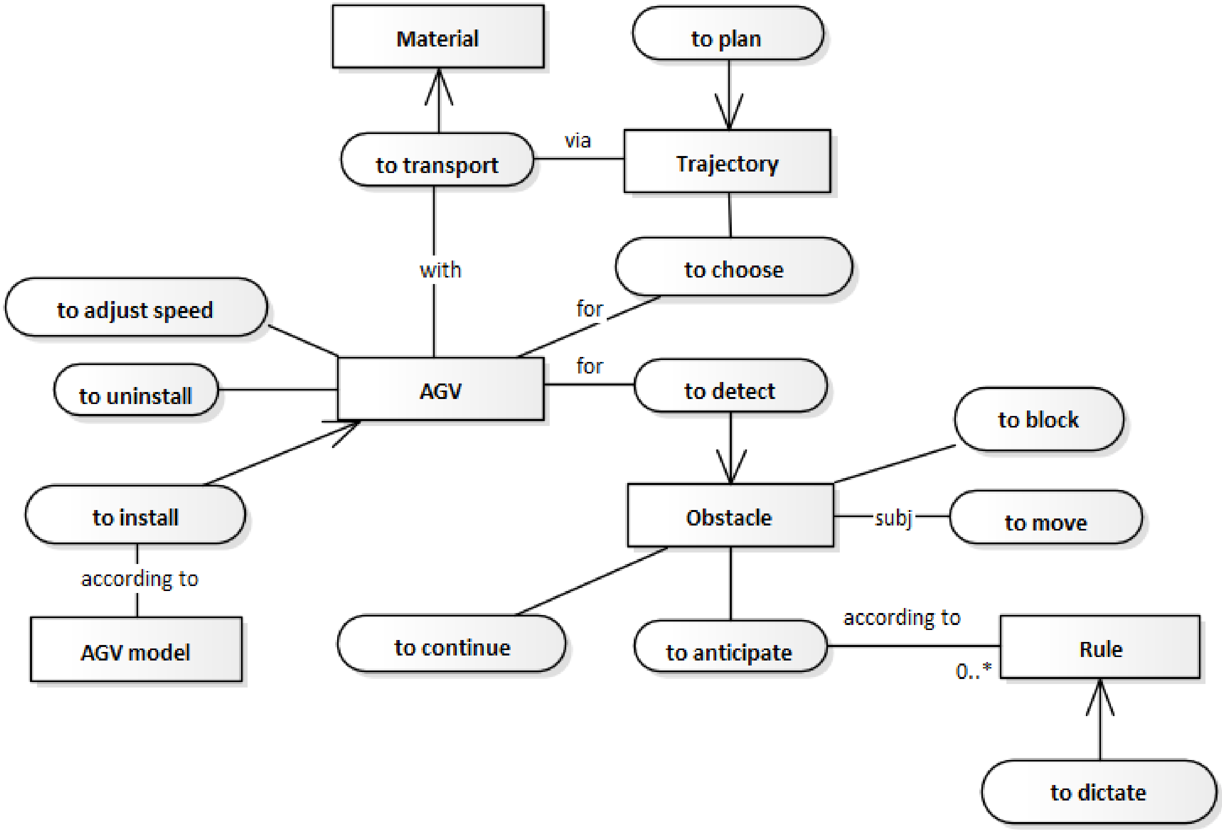


«Domain Model»
Functional Safety

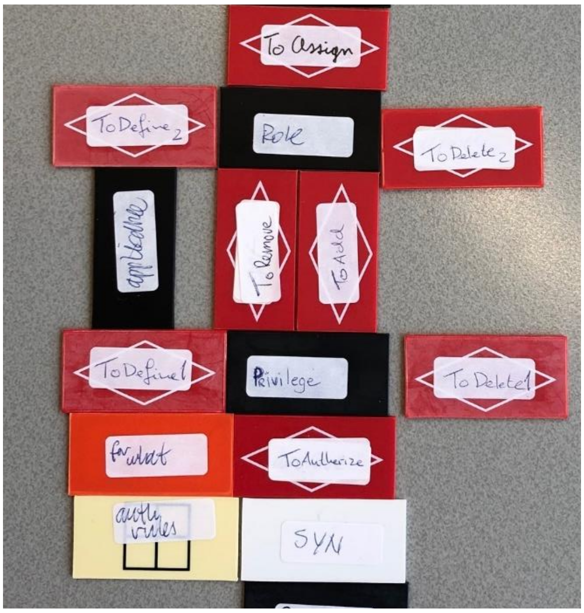
+ Safety Analysis Action
+ To Combine
+ To Reject
+ *Safety Analysis Object*
+ Safety Assumption
+ Safety Justification
+ ASIL Ranking
+ *ASIL Object*
+ Item Definition
+ Hazard Analysis and Risk Assessment

5.4.2 Interaction view

- Root: domain
- Derivation:
  - The domain activities of the domain, and the roles and involvements in their role structure.
  - The generalizations in the classifier structure of the domain activities.



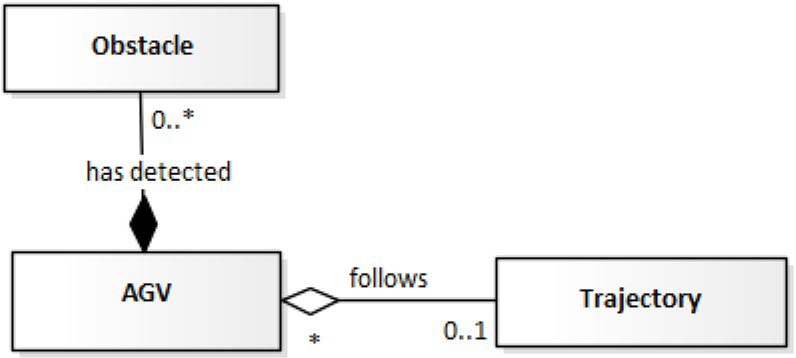
Example with KISS domino:



5.4.3 Class view

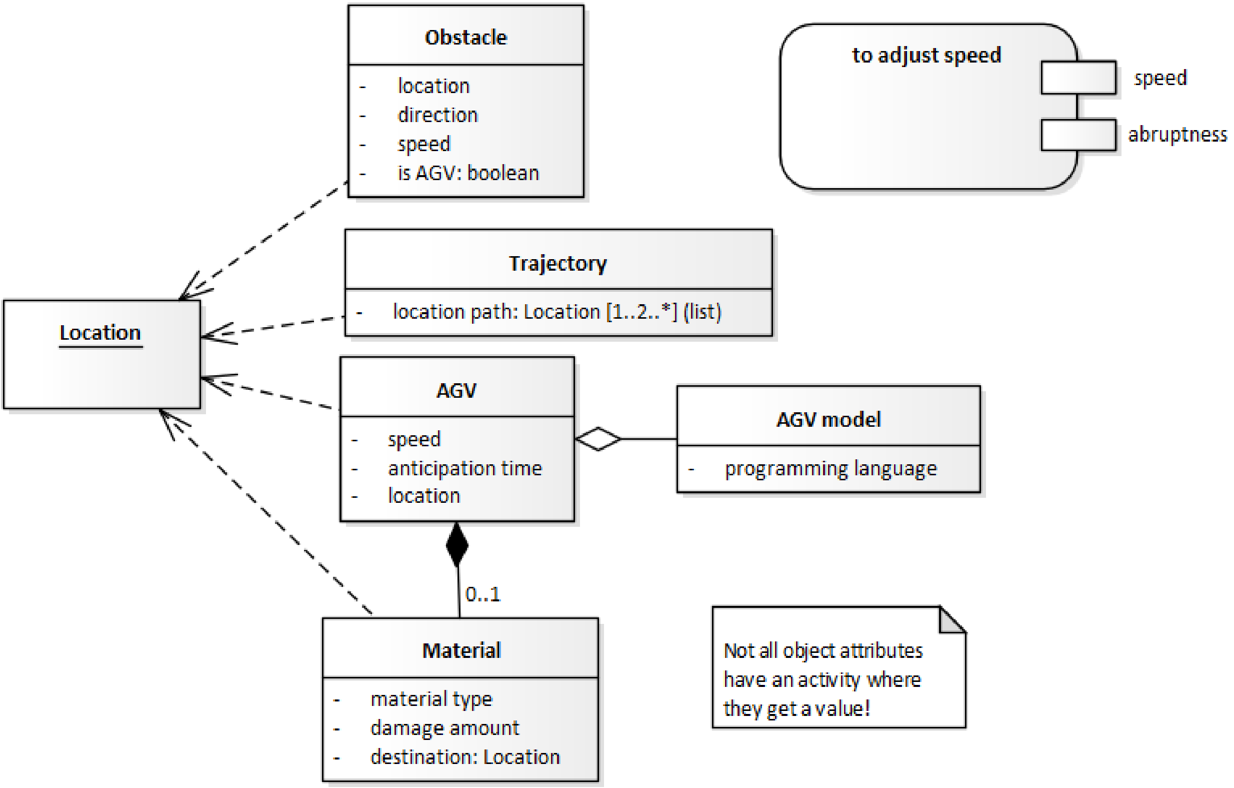
- ○ Root: domain
- ○ Derivation:
  - ▪ Domain classes in the domain.

- References in the Life dependencies of the domain classes.
- Domain class relations in the domain, and the class relation roles and their role connections, in the class relation structure of the domain class relations.



5.4.4 Attribute view

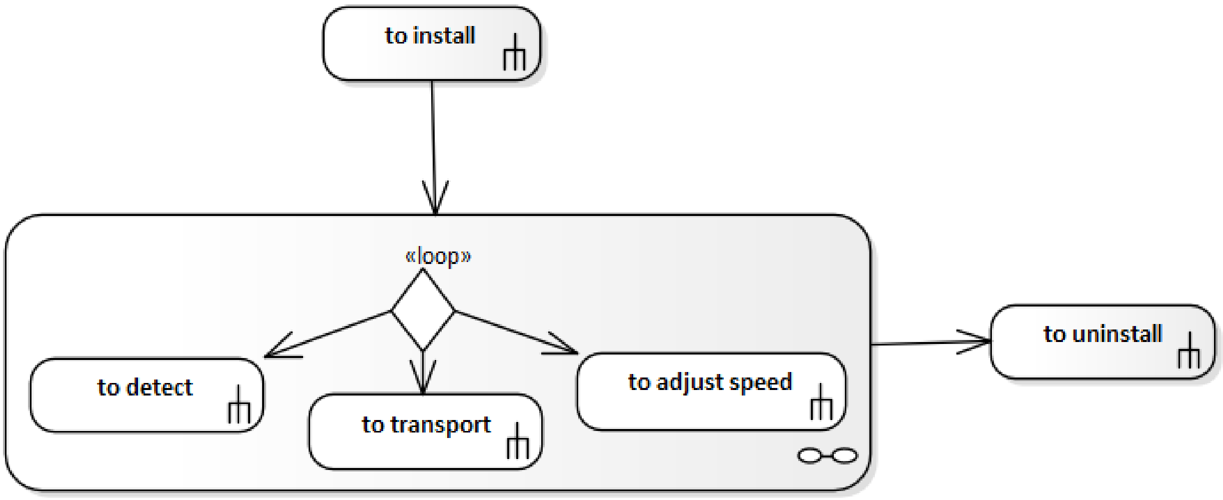
- Root: Domain
- Derivation: Attributes in the attribute structure of each of the:
  - domain classes,
  - domain activities,
  - the activity roles of those domain activities,
  - the involvements of those activity roles.



5.4.5 Object lifecycle view

- Root: domain class

- Derivation: the flows steps in the object lifecycle of the domain class.



5.4.6 Activity view

- Root: domain activity
- Derivation:
  - The activity operations and operation invocations in the activity model of the domain activity.
  - And the actual parameters in the actual parameter structure of the operation invocations, and the bound attributes of each actual parameter.

No example yet.

5.5 For features

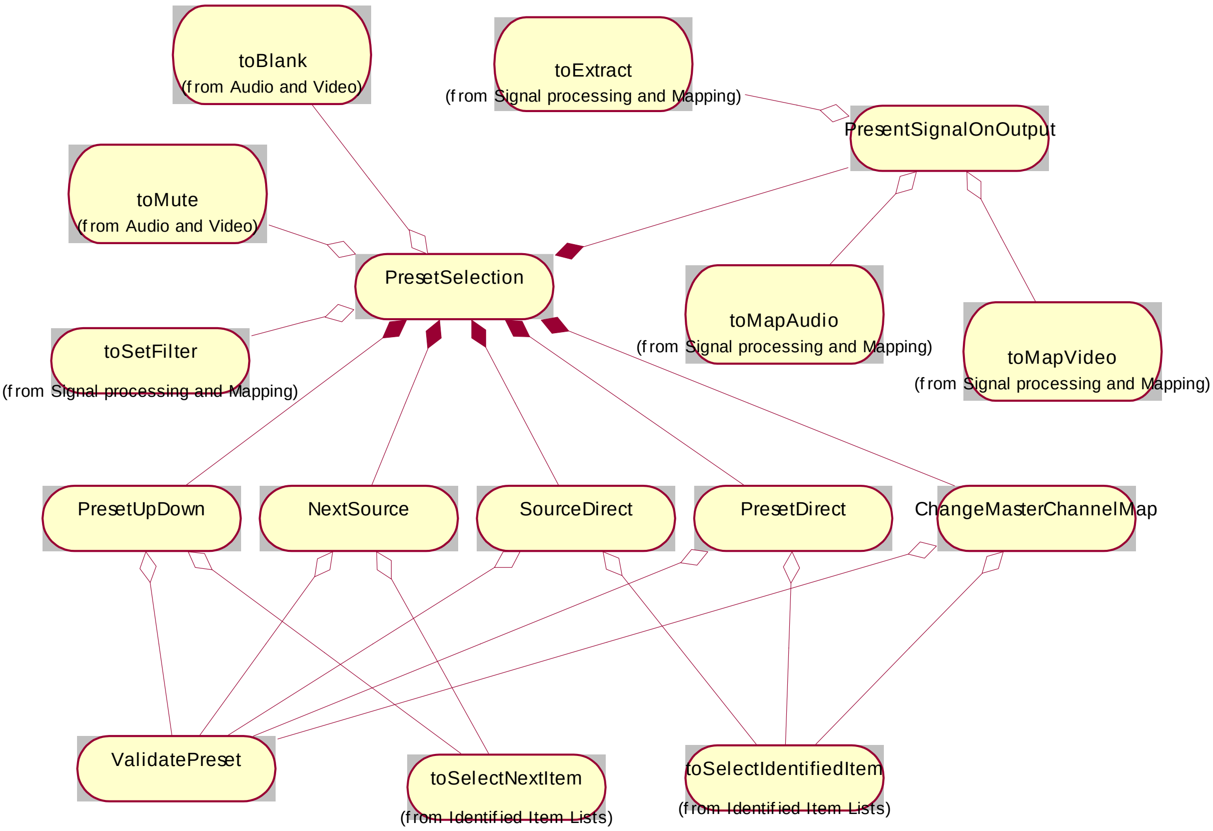
5.5.1 Declarations view

- Root: Feature
- Derivation: The specification elements in the specification declarations of the feature.

No example, but it looks the same as the other declaration views.

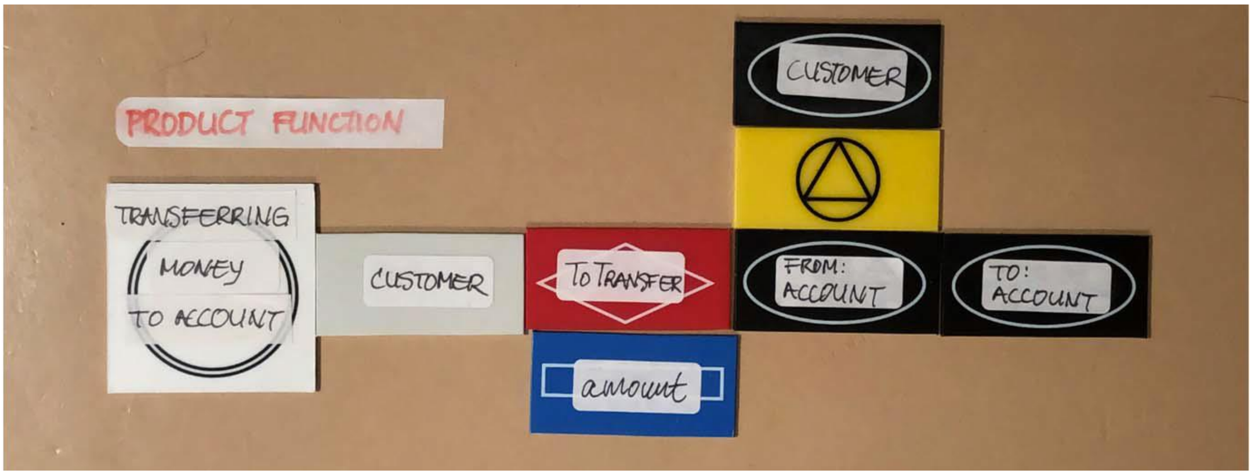
5.5.2 Feature structure

- Root: Feature
- Derivation: All the compositions in the functional composition structure of the functions of the feature (including the feature itself).



1.1.1 Function lifecycle

- Root: function
- Derivation: (similar to activity view)
  - The function steps in the function flow of the function.
  - The function step participants in the function step structure, and the bound Function attributes of each step participant.



Below an example that is a purely textual notation. The more logical choice is a notation based on the UML activity diagram.

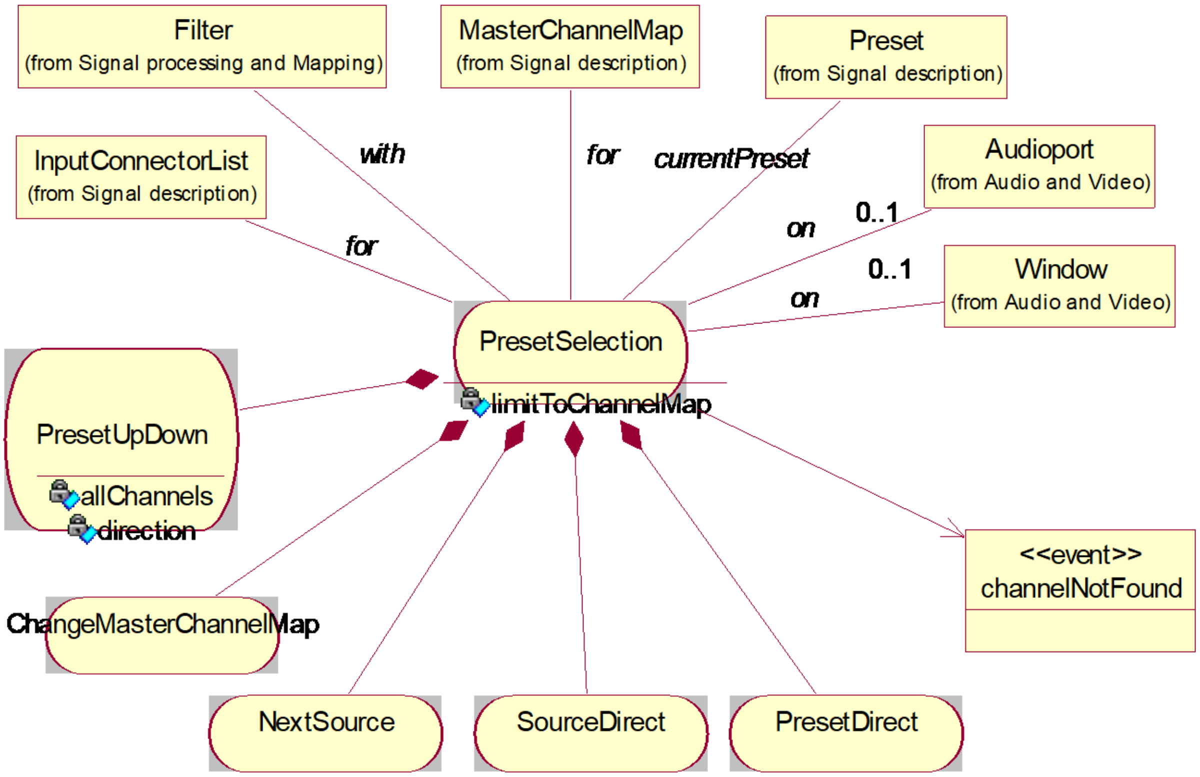
```
if      this event is equal to the n events before
        longPress = true;
        if      the table contains longPress for this event
                for all events in the selected column
                        generate the event as indicated in Table 1;
elif    this event is equal to the m events before
        extraLongPress = true;
        if      the table contains extraLongPress for this event
                for all events in the selected column
                        generate the event as indicated in Table 1;
elif    this event is not equal to the previous event
        generate the event as indicated in Table 1;
end if;
```

Table 1: KeyFiltering definition of code mapping

Code	Condition	if not HotelMode	if HotelMode
RC5 0,56 RC6 0,56		RCNextSource	RCNextSourceHM
RC5 0,32 RC6 0,32 RC6 0,30		RCChanStepUp	RCChanStepUp
RC5 0,3 RC6 0,3		RCDigit 3	RCDigit 3
RC5 0,4 RC6 0,4		RCDigit 4	RCDigit 4
RC5 0,5 RC6 0,5		RCDigit 5	RCDigit 5

5.5.3 Function signature

- Root: function
- Derivation:
    - All the attributes in the attribute structure of the function.
    - All the function event parameters of the function. *(This should be changed into an event structure where the static elements are events and the referred Items are behavior elements.)*



Other example:

Signature

Attribute	**Type**	**Scope**	**Description**
theWindow	Window	In, out	This instance of PresetSelection is applicable for this Window.
theAudioPort	Audioport	In, out	This instance of PresetSelection is applicable for this Audioport.
currentPreset	Preset	In, out	The preset that is shown.
masterChannelMap	MasterChannelMap	In	The map that is currently in use.
inputConnectorList	InputConnectorList	In	The list of Extensions in the TV.
theFilter	Filter	In, out	A reference to the filter (in most cases a tuner) that is used to extract the program signal from the Input connector.
limitToChannelMap	Boolean	In	If limitToChannelMap = true, then entering digits that identify a channel that is not in the masterChannelMap does not invoke a preset change. E.g. in case of authenticated POD.
when(theFilter.signalAvailable)	Event	In	
PresetUpDown (direction: UpDown, allChannels: Boolean)	Function	In	
NextSource()	Function	In	
PresetDirect(targetPresetID: IDString)	Function	In	
SourceDirect(targetPresetID: IDString)	Function	In	
ChangeMasterChannelMap(mcm: MasterChannelMap, limit: Boolean)	Function	In	
channelNotFound	Event	Out	
theAudio	AudioSignal	Local	

Attribute	Type	Scope	Description
theVideo	VideoSignal	Local	
newPreset	Preset	Local	
ChangePreset(newPreset: Preset)	Event	Local	
ValidatePreset(tmpPreset: Preset)	Function	Local	
PresentSignalOnOutput()	Function	Local	

6 Modeling constructs Package

This package contains the different modeling constructs that will be used throughout the definition of the MuDForM modeling concepts. The role of modeling constructs is to provide uniform model structures, and consequently enable a uniform modeling language, and a uniform way of parsing and interpreting models. There are currently two sets of constructs:

- the Structure patterns, which provide a uniform way to deal with structured relationships between a container modeling concepts and a contained modeling concept.
- the formalism pattern, which enables to embed existing formalisms into MuDForM models.
- Pattern based queries/constraints *(lack of a better name), which are not modeled yet. Examples:*
  The children of a instance.
  Sameparent(A,B,R): A and B should have the same parent C along the relation R.
  Samegrandparent(….).

6.1 Structures Package

This package introduces several generic patterns for constructing recursive relations between modeling concepts. We distinguish the following structure patterns:

- Structure: analogous to a tree, with one root (Structure), nodes (Elements and Substructures), and leaves (Elements). An Element may refer to another modeling concept (Referred Item), which makes such an Element a Reference.
- Static structure: a structure that forms a regular expression in which the composition is expressing the possible instance structures of a classifier.
- Flow structure: Ordered structures for behavior items. Like the static structure, this is a regular expression.
- A reference structure in which the structures of the references in a tree are connected to the nodes in the tree. So, a leaf of Tree A refers to a tree B and the other nodes of A are connected to the nodes of B. Example: an activity invocation in a function. The function parameters must be connected to the parameters of the invoked activity.

6.1.1 Structure

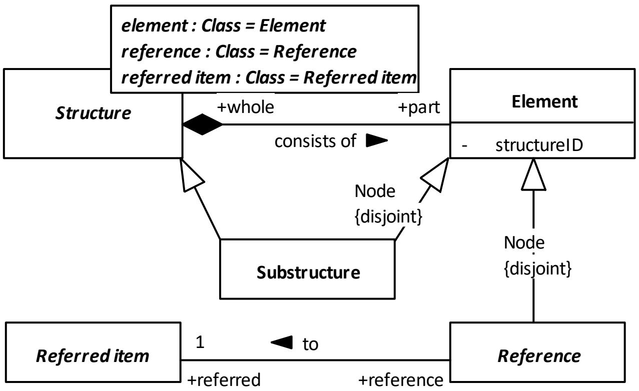
A Structure consists of Elements. An Element can be, but not necessarily, either a Substructure or a Reference to a Referred item. This way a tree-structure can be created and the leaves of the tree, i.e., elements, can point to another concept that might be defined outside the structure. in that case the Element is a Reference.
The intention is to reuse this as a pattern for more specific structures, like classes with attributes, and object life cycles. To achieve this, the class Structure is a template class. The parameters of the class correspond with the variable part of the pattern. The parameters must be bound when applying the Structure class:

- elements must be bound to a class. The default value is the class Element.
- references must be bound to a class. The default value is the class Reference.
- referred item must be bound to class. The default value is the class Referred item.

It is possible to add derived associations (not presented in the diagram). For example:

- For all the elements in the tree of a structure (recursively).
- "Structure.refersto.Referred item": the items that are referred to from a structure. (similar to a dependency).

Those derived associations may be useful in the definition constraints for modeling concepts to which the structure is applied.

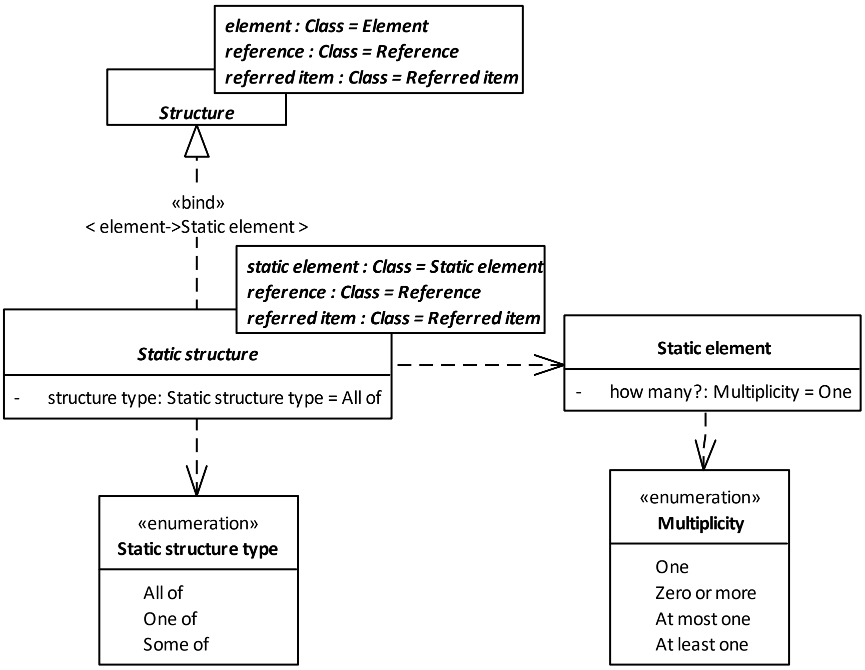


Structure

6.1.2 Static structure

A static structure is meant to constraint and prescribe the possible elements of a structure of a classifier. The structure type defines which parts in the structure must/can be present. For example:

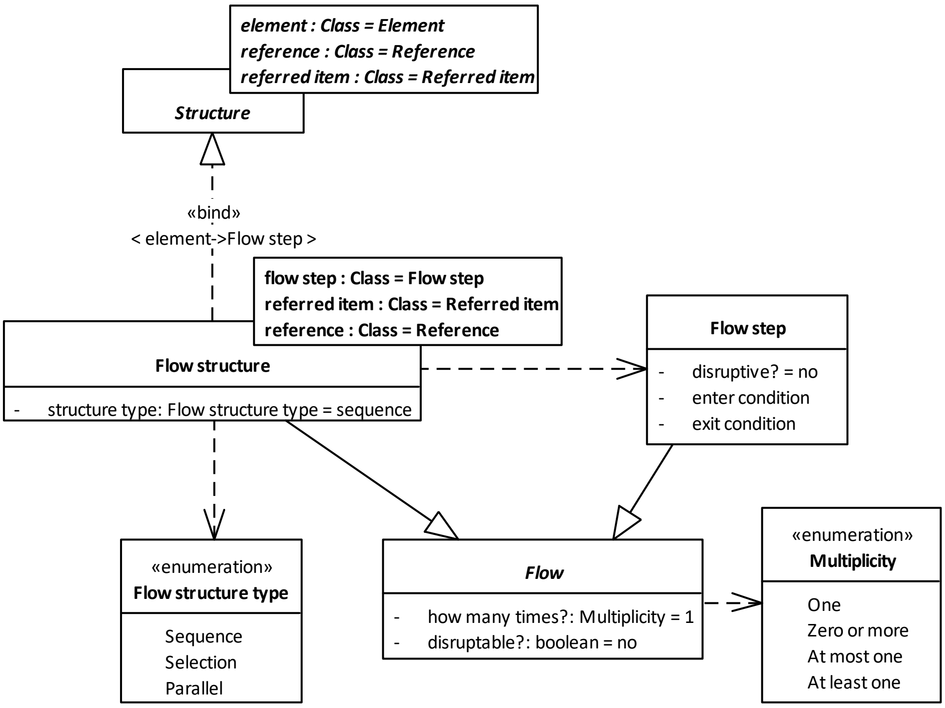
- A product has a name and a code (type=All).
- A product has a name and/or a code (type=Some).
- A product has either a name or a code (type=One).


element : Class = Element
reference : Class = Reference
referred item : Class = Referred item
Structure
«bind»
< element->Static element >
static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Static structure
- structure type: Static structure type = All of
Static element
- how many?: Multiplicity = One
«enumeration»
Static structure type
All of
One of
Some of
«enumeration»
Multiplicity
One
Zero or more
At most one
At least one

Static structure

6.1.3 Flow structure

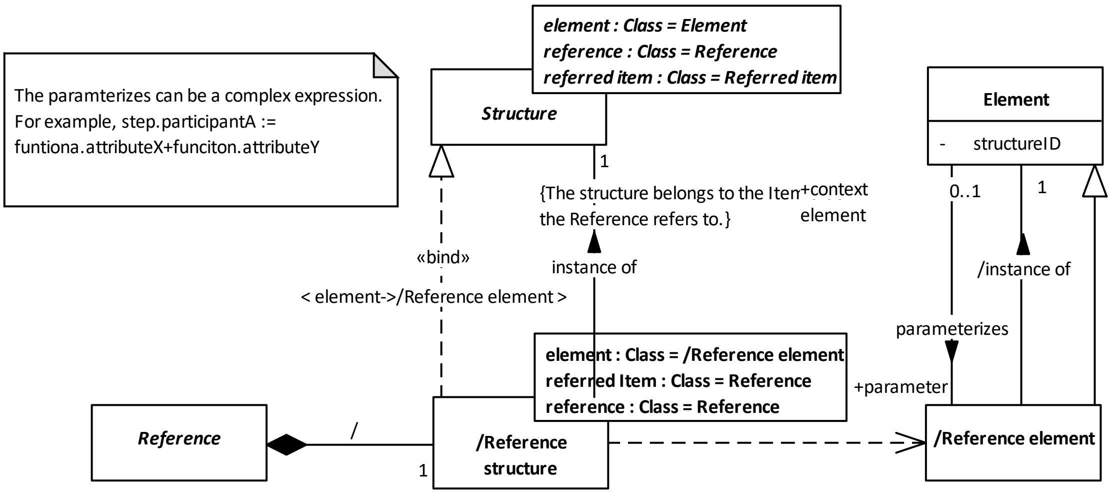
A Flow structure is used to constraint and prescribe an ordering of Referred items (Flow steps), i.e., a control flow. A Flow structure is a Structure where the element parameter of the Structure template class is bound to the class Flow step. The structure type defines the order relation between the Flow steps in a Flow structure. The Flow structure has three parameters: flow step, referred item, and reference.


element : Class = Element
reference : Class = Reference
referred item : Class = Referred item
Structure
«bind»
< element->Flow step >
flow step : Class = Flow step
referred item : Class = Referred item
reference : Class = Reference
Flow structure
- structure type: Flow structure type = sequence
Flow step
- disruptive? = no
- enter condition
- exit condition
«enumeration»
Flow structure type
Sequence
Selection
Parallel
Flow
- how many times?: Multiplicity = 1
- disruptable?: boolean = no
«enumeration»
Multiplicity
One
Zero or more
At most one
At least one

Flow structure

6.1.4 Reference structure

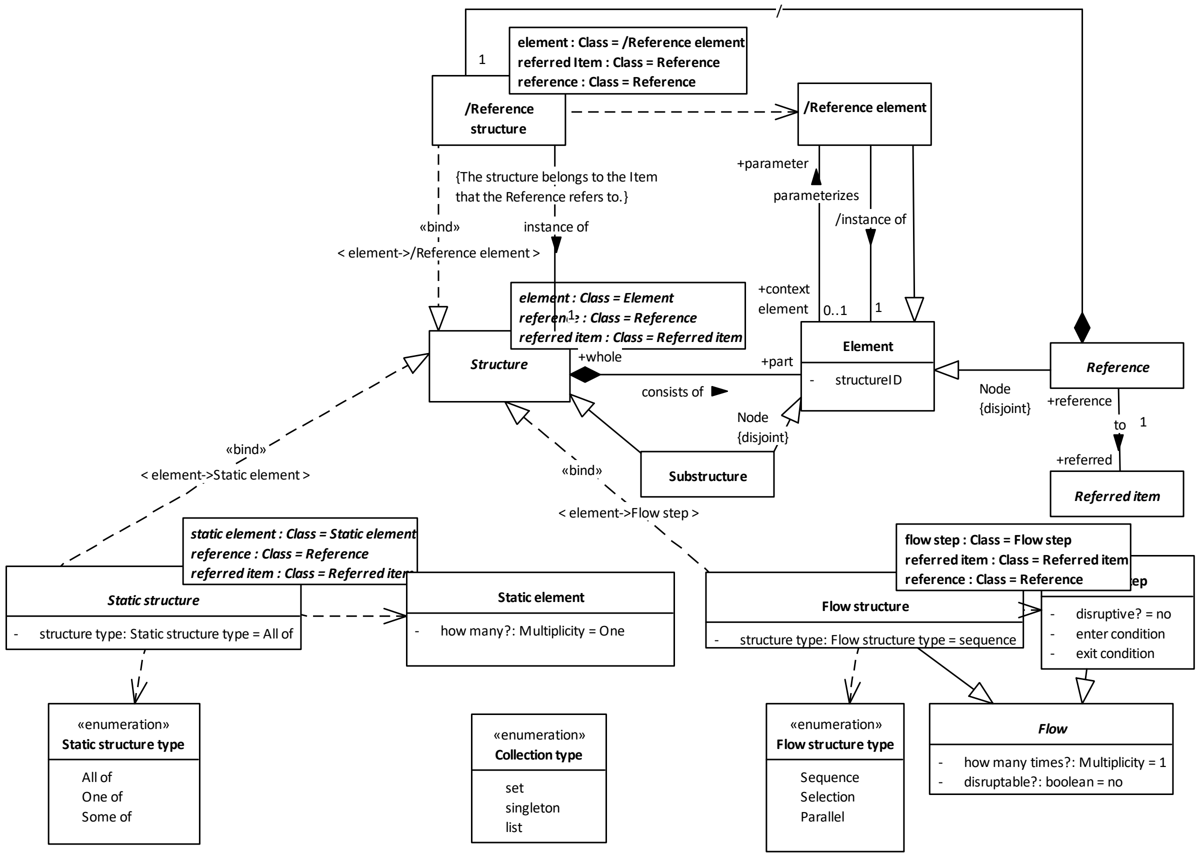
A Reference can have a derived Reference structure. A Reference structure is a structure in which the elements are Reference elements. The Reference structure is an instance (and as such a copy) of the structure of the item that the Reference refers to, and as such the Reference structure is derived. This structure is formed by the Reference elements. For each element of the structure of the Referred item, the Reference structure may have a Reference element. A Reference element will be parameterized by an Element of the main Structure. The main structure is the structure of the same Item that the structure of the reference belongs to.
Example: if a function F (=item) contains (=consists of) a function call G (=reference to Referred item G), then the function call structure (=reference structure) has active parameters (=reference element) for the formal parameters (=element) of function G (=structure of item). The actual parameters of that call of function G are then parameterized with the parameters (variables) of Function F. (Of course, the types (referred Items) of the two parameters must match. i.e., they are the same or the type of the Element is a subclass of the type of the Reference element.)


The paramterizes can be a complex expression. For example, step.participantA := funtiona.attributeX+funciton.attributeY
element : Class = Element
reference : Class = Reference
referred item : Class = Referred item
Structure
Element
- structureID
1
{The structure belongs to the Item the Reference refers to.}
+context element
0..1
1
«bind»
< element->/Reference element >
instance of
/instance of
parameterizes
element : Class = /Reference element
referred Item : Class = Reference
reference : Class = Reference
+parameter
Reference
/
1
/Reference structure
/Reference element

Reference structure

6.1.5 All structures

This diagram is the union of the other four diagrams related to the Structure pattern.


/
1
element : Class = /Reference element
referred Item : Class = Reference
reference : Class = Reference
/Reference structure
/Reference element
+parameter
parameterizes
/instance of
{The structure belongs to the Item that the Reference refers to.}
«bind»
< element->/Reference element >
instance of
element : Class = Element
reference : Class = Reference
referred item : Class = Referred item
1
+context element
0..1
1
Structure
+whole
+part
consists of
Element
- structureID
Reference
+reference
to
1
+referred
Referred item
Node {disjoint}
Node {disjoint}
Substructure
«bind»
< element->Static element >
«bind»
< element->Flow step >
static element : Class = Static element
reference : Class = Reference
referred item : Class = Referred item
Static structure
- structure type: Static structure type = All of
Static element
- how many?: Multiplicity = One
flow step : Class = Flow step
referred item : Class = Referred item
reference : Class = Reference
Flow structure
- structure type: Flow structure type = sequence
ep
- disruptive? = no
- enter condition
- exit condition
«enumeration» Static structure type
All of
One of
Some of
«enumeration» Collection type
set
singleton
list
«enumeration» Flow structure type
Sequence
Selection
Parallel
Flow
- how many times?: Multiplicity = 1
- disruptable?: boolean = no

All structures

Element	**Description**
Collection type	Possible collection types of a static element. **attribute:** set type: An reference to this type can have zero or more elements. **attribute:** singleton type: A reference to this type has exactly one element. **attribute:** list type: A reference to this type has zero or more elements that are ordered by StructureID.
Element	An identified part of a structure. The structureID identifies the element in the scope of the structure. An element can be a substructure or a reference, but never both (as expressed by the disjoint property of the specializations. **attribute:** structureID type: An identifier for this element, unique within its structure, e.g., a name or a sequence number.
Flow	Unit of behavior specification that is composed of other behavior. **attribute:** how many times? type: Multiplicity Specification of how many times the Flow can or must be executed. This specification might be a range, e.g., "0..N" or a condition that acts as a guard on the iteration of the Flow. **attribute:** disruptable? type: boolean

	Can the flow can be stopped, such that it exits without being fully executed,?
Flow step	Step in a Flow structure, possibly referring to a behavior item. **attribute:** disruptive? type: Can this step disrupt the previous step (if there is one), or does it just follow the previous step? When the previous step is disrupted, it exits without being fully executed. **attribute:** enter condition type: What must be valid to enter this step. **attribute:** exit condition type: What must be valid to exit this step.
Flow structure	A structure in which the elements are Flow steps and the referred items are Behavior items. A flow structure defines the control order of steps. **attribute:** structure type type: Flow structure type The temporal relation between the steps.
Flow structure type	Default set of possible types of a flow structure. **attribute:** Sequence type: All the Flow steps in the Flow are executed after each other. **attribute:** Selection type: Exactly one of the Flow steps in the Flow is executed. **attribute:** Parallel type: All the Flow steps in the Flow are executed concurrently (in arbitrary order).
Multiplicity	**attribute:** One type: **attribute:** Zero or more type: **attribute:** At most one type: **attribute:** At least one type:
Referred item	A concept that has an identifier and that can be referred to. It is a placeholder for all the things you want to refer to from an Element in a Structure, which makes such an Element a Reference.
Reference	An Element in a Structure that refers to a Referred item.

/Reference element	An element in a Reference structure. A Reference element is an Element and an instance of an element in a Structure of the Item that the Reference Item of the Reference structure refers to. See derivation in /Reference structure. A Reference element can be parameterized by an element in the containing structure.
/Reference structure	The structure of a reference. A reference structure is copied from the structure of the referred item. As such, the Reference structure is an instance of the structure of the Referred Item. Derivation: a Reference structure is derived from a Structure S of the Reference that it is a part of. It contains a /Reference element for each Elements E of S. The Reference element is instance of such E.
Static element	An Element of a Static structure. The attribute "how many?" specifies the constraints to the plurality of the Static element instances within an instance of the Static structure. **constraint:** note **constraint:** type=singleton impiles maximum multiplicity=1 **attribute:** how many? type: Multiplicity Specification that states how many values/instances the Static element may have. This specification might be a range, e.g., "0..N", or a condition that acts as a guard on the size of the set or list.
Static structure	A Structure in which the Elements are Static elements. The type defines which instances of the Static elements and instance of the Static structure may or must have. • All: all the elements must be present. If the structure is contained in a classifier, then it defines that all the elements must have an instance. • Some: at least one of the elements is present. • One: exactly one of the elements is present. **attribute:** structure type type: Static structure type
Static structure type	Default set of possible types of a static structure. **attribute:** All of type: Indicating that all elements in the static structure should have a value/instance. **attribute:** One of type: Indicating that exactly one of the elements in the structure should have a value/instance. **attribute:** Some of type: Indicating that 1 or more elements in the structure should have a value/instance.
Structure	A concept that consists of Elements. The Elements are identified by the structureID.

Substructure	An Element in a Structure that is itself also a Structure. Typically, used for the nodes in the structure that are not a leave.

6.2 Formalisms Package

The definition of the MuDForM modeling concepts is based on several theories.

- Set theory (Sets and operators). This is used in classifiers and allows to speak of instances of a class, or of subclasses. Of course, UML itself is also based on set theory.
- Process algebra (composition of behavior). We use process algebra to define the semantics of flow structures.
- Graph-theory, especially acyclic graphs and tree-structures. We use this in the definition of the different model construction patterns.

By using the theories, we enable a formal interpretation of a MuDForM-compliant model and we achieve uniformity across the different views on a model. Although we propose UML as a notation for MuDForM, UML itself does not assure, let say enforce, semantic consistency of views.

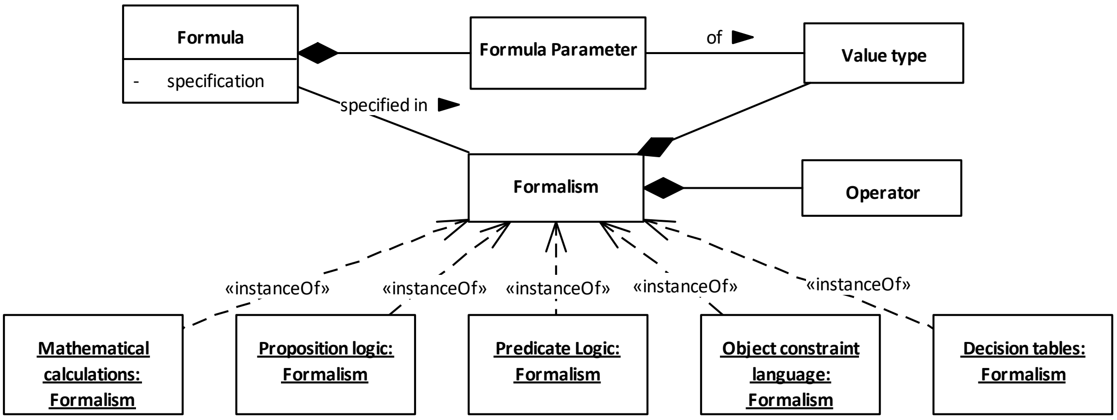
6.2.1 Formalisms

MuDForM offers the use of available (mathematics) formalisms for the detailed specification of some model elements. We will not make an explicit metamodel of these theories. We will use an abstraction of these theories that allows their usage in MuDForM specifications. The diagram shows examples of allowed formalisms:

- Mathematical calculations, like Natural numbers and the operators that are used in formulas.
- Proposition logic, a.k.a. Zeroth order logic (Boolean expressions), used in conditions and operations.
- First order predicate logic (statements about sets, like forall, exists,..), used in conditions.
- Object constraint Language (language for constraints in UML), used in conditions.
- Decision tables (ref), used in operations.

A Formula has parameters which get their value when the formula is used in a context i.e., is executed. A formula is specified in some formalism. The given instances are examples of typical formalisms that might be used. Formalisms must all have a formal syntax and semantics. Of course, the syntax may differ per formalism, from a string of characters, to a decision table structure. A formalism consists of operators and value types. For example, Proposition logic has Boolean operators, like AND and OR, and Boolean value (true or false) as value type.

Physical quantities. like speed, weight, and temperature, and their units can also be seen as formalisms.


Formula
- specification
Formula Parameter
of
Value type
specified in
Formalism
Operator
«instanceOf»
«instanceOf»
«instanceOf»
«instanceOf»
«instanceOf»
Mathematical calculations: Formalism
Proposition logic: Formalism
Predicate Logic: Formalism
Object constraint language: Formalism
Decision tables: Formalism

Formalisms

Element	**Description**

Formalism	Definition of a set of Operators, Value types, and rules for specifying formulas.
Formula	Definition of a calculation in some formalism. The specification defines the formula in terms of the operators and value types of the formalism. **attribute:** specification type:
Formula Parameter	Formal parameter of a formula, which gets a value when the formula is used. A formal parameter has a value type that must fit with the type of the actual parameters in the use of a formula.
Operator	A mapping that maps values of value types onto other values. Operators can be simple mathematical operators like "+" and "-", or Boolean operators like "OR" and "AND", but also more complex ones like a decision table.
Value type	Definition of a type that is used in a Formalism. A Value type defines the rules to which a formula parameter must adhere.
Decision tables	The formalism of decision tables as defined in [ref]. Decision tables can be used to direct decision regarding attribute values, or control flows.
Mathematical calculations	Mathematical calculations are typically used in operations on attributes.
Object constraint language	The OCL as defined in [ref]. OCL constrains can be used in invariants, preconditions, and postconditions throughout a model.

Predicate Logic	Predicate logic can be used in all kinds of conditions.
Proposition logic	Proposition logic can be used in simple conditions throughout a model.