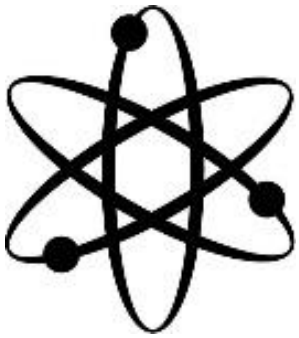


Atom Free IT
for true business agility

MuDForM Definition

Metamodel, method flow, guidelines, and viewpoints

Version:16-5-2024
Author: Robert Deckers

Distribution and usage limited to: Atom Free IT

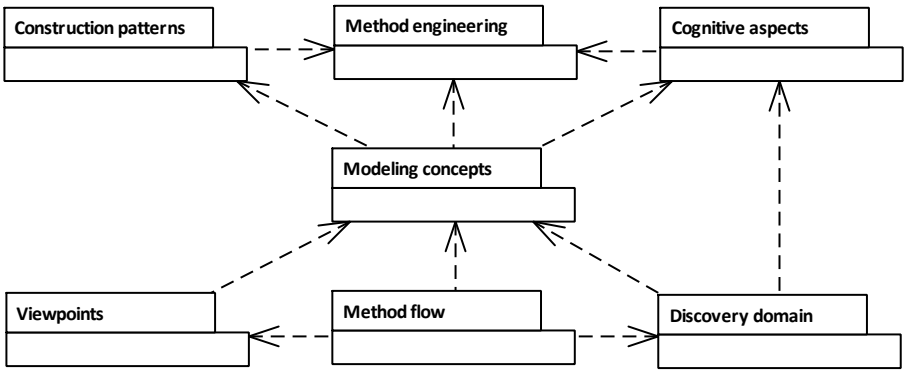
Table of Contents

1 MuDForM definition

This package contains the specification of MuDForM. It explains the metamodel, the method steps and guidelines, the concepts for the discovery phase, the possible applications of a MuDForM specification, and the principles behind the metamodel and method.

1.1 MuDForM definition structure diagram

The complete MuDForM specification structure is defined in several packages. The core is formed by four packages. The metamodel is in the MuDForM modeling concepts package, in which all the different modeling concepts are defined. The Discovery domain package defines all the concepts that are used in the phase where texts from expert is analyzed such that it can be transformed into a MuDForM model. The Viewpoints package explains the different viewpoints that are used in creating a MuDForM compliant model. The Method flow package explains the steps and guidelines for making MuDForM models.
The other packages explain how the core is created, except the model usage package, which defines how a MuDForM model can be used in different contexts.
The root of the whole definition is described in the MuDForM method engineering package, which explains how the core is defines using the elements from the Cognitive aspects package and the Modeling constructs package.
Although MuDForM is not tied to a specific notation, we have experimented with the use of the UML notation in the MuDForM Viewpoints. The Subset of UML packages explains which parts of the UML notation have been used in our experiences with MuDForM modeling.



MuDForM definition structure

2 Method engineering

This package explains the high-level structure of the MuDForM method definition and how the MuDForM modeling concepts are constructed from the modeling construction patterns and the cognitive aspects.

2.1 Method ingredients diagram

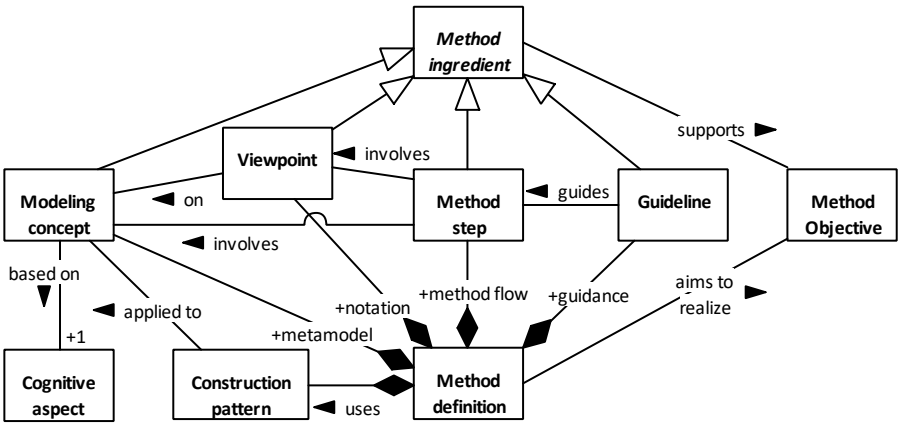


Figure 1: Method ingredients

2.2 Modeling concepts construction diagram

In the MuDForM language metamodel we separate three different types of "things":

- The cognitive aspects are the things that we want to express in models via modeling concepts. They form the foundation for the semantics of the modeling concepts.
- Method constructs are the things that are used to express the modeling language on an abstract level. The constructs are the basis for the abstract syntax of the MuDForM language. We try to use a minimal set of constructs for the sake of language simplicity and uniformity. Separating the structure from the notation allows the use of different (abstract) syntaxes. For example, we have chosen now to use process algebra, via the flow structure construct, to express a possible flow of events. But it is possible to replace is with a state transition diagram, or a Petri Net.
- Base modeling concepts are the language elements that can be declared autonomously. Every Base modeling concept has a set of Contained modeling concepts which follow a method construct.

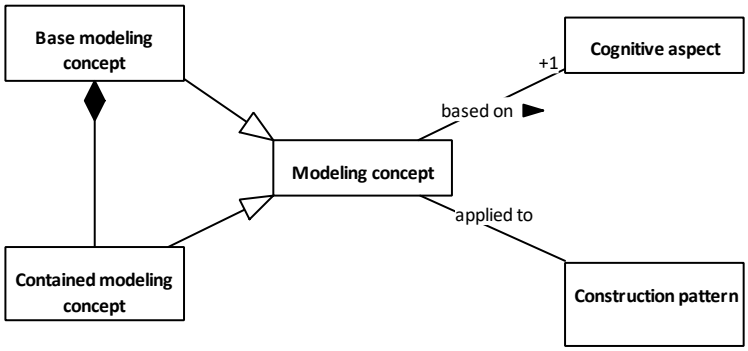


Figure 2: Modeling concepts construction

2.3 Classifier and Structures diagram

The models that we want to make mostly describe classes of possible instances. We use these main types of classifiers aspects:

- Action classifiers: Classifiers that describe changes. Think of Activities, Operations, and Functions.
- Object classifiers: Classifiers that describe things with a life, i.e., things that have a state. So, their instances can be involved in at least two changes. Think of Domain Classes and Functions.
- Actor classifiers: Classifiers that describes things that are capable of executing behavior and can react to and generate events. Think of actors, like people and systems.
- Space Classifiers: Classifiers that contain definitions of other classifiers, and serve as a namespace. Think of Domain, Features, and Contexts. Classes and Activities are a namespace for Attributes.

We use different types of structures to define these classifiers:

- Object structures: Classifiers that describes possible compositions of identities. Think of Class attributes (references from a property to a class), Action roles, Involvements of domain classes in Action roles, and Function parameters.
- Class structures: Structures that define classifications relations. Think of the subclass relation between two domain classes, or an abstract activity and its specializations.
- Life structures: A structure that describes possible changes on a life. Think of the Object life cycle of a domain class, or the control flow of a Function, or the flow of operations in an activity.
- Constraint structures: A structure that is a specification of allowed instances of a classifier. A constraint may involve anything that can be used to describe the instances for the classifier. Constraints can be used to describe instances of value and life classifiers, or to define the preconditions and postconditions of instance of action classifiers.

A MuDForM compliant specification will consists of a collection of models of contexts, domains, and features.

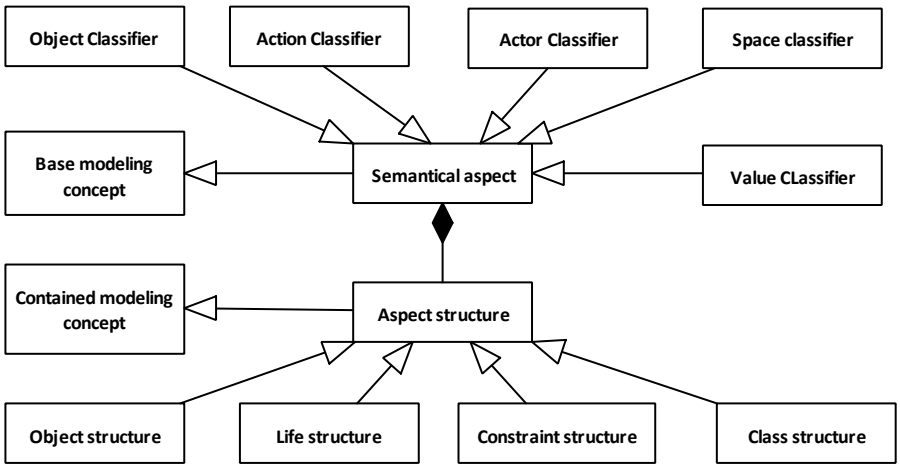


Figure 3: Classifier and Structures

2.4 Interpretable specification and terminology diagram

In general, a specification is defined via a metamodel. A metamodel forms the language (grammar and keywords) of such a specification. MuDForM separates specifications in three: domain specifications, domain-based specifications, and contexts. We say that a domain-based specification is written in terms of a domain specification. This idea is derived from the Paradigm of De Leeuw [ref], which that states the difference and relation between a control system and a controlled system. The control system specifies what things must happen and exist in the controlled system. To be able to specify what shall happen, we must know what can happen and exist in the controlled system. The specification of the control system is called a feature, and the specification of the controlled system is called a domain model. A feature is a prescriptive statement and a domain model is a descriptive statement (see Cognitive aspects : Statements). To specify features and domains, one might also need generic concepts that are neither part of the feature nor part of the domain or interest. The generic concepts are defined in a context model.
Three examples where this distinction between the three different types of specifications is useful:

- In the control process of a factory, the factory is represented by a domain model and the control process in a feature. The rules for safety and other aspects that need to be considered are the context.
- In an embedded system, the control software is defined by a feature and the physical capabilities of the device (hardware) in a domain model. The operators and the hardware itself are part of the context.
- In a warehouse there are processes for ordering products and picking products. The products and warehouse are defined in a domain model. The rules for ordering, picking, and shipping might vary over time and are modeled in a feature. Suppose the product catalog management is out of scope, but it is needed to define orders. The needed aspects of the product catalog are then defined in a context model.

MuDForM allows any MuDForM specification to act al all three types. This is possible because the modeling constructs are applied analogously on all three levels. So, a function in a feature specification can become an operation in a domain that considers that feature specification as a context model. Or, an operation in a context model can become an activity, when the context is later modeled in a domain model. This enables an incremental approach for the extension of domain-based specifications for a complete system or organization.

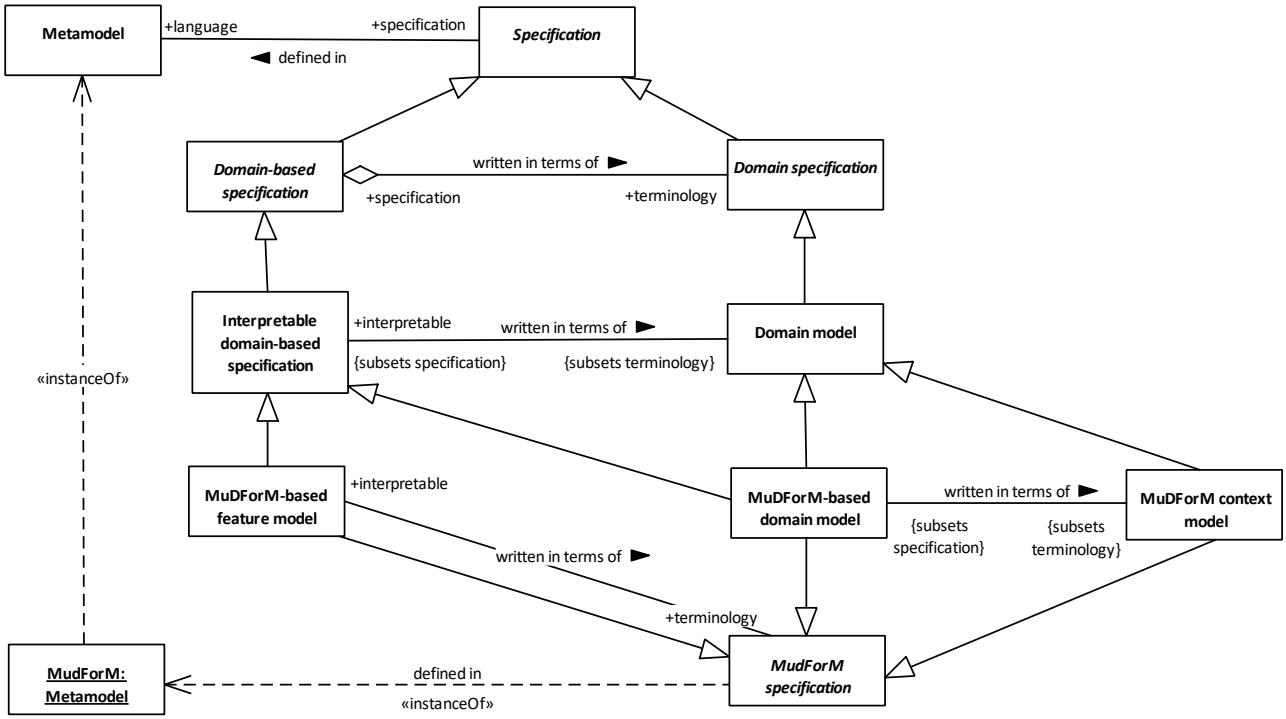


Figure 4: Interpretable specification and terminology

2.5 Specification structure of a MuDForM model diagram

There are three types of MuDForM specifications that we depict here with a UML package symbol:

- Domain models that define what can happen and exist in the considered domain (aka domain of interest). The core concepts are domain classes and domain activities.
- Feature specifications that define what shall happen and exists. The main concepts are function and actor (aka active object).
- Context models provide the basics operations and classes that are needed to define domain models and features.

(TBD: Later we will also consider Domain integration specifications. This text has to be changed due to the integration of consistency specifications between domains. For the moment this a special type of domain model.)

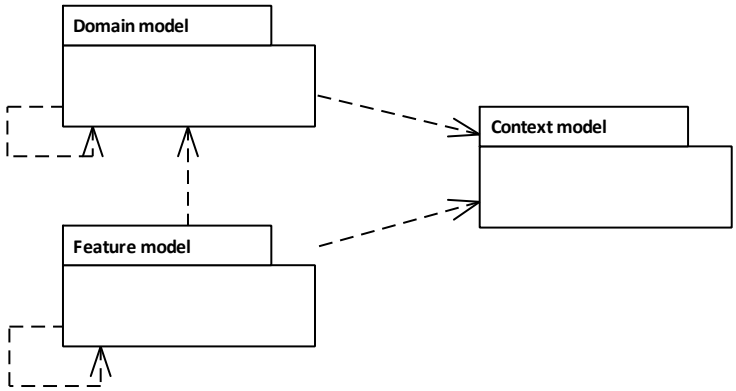


Figure 5: Specification structure of a MuDForM model

Name	Definition
Action Classifier	Classifier that expresses instances that are changes at a moment in time.

Actor Classifier	Classifier that expresses instances that can execute and react to changes.
Aspect structure	A structure via which the a cognitive aspect of a modeling concept is captured.
Base modeling concept	Concepts used to define MuDForM compliant models. A Base modeling concept consists of a set of Modeling concept aspects.
Class structure	Structure that expresses the relations between classes (sets), especially super class and sub class relations.
Cognitive aspect	A class or relation in the cognitive aspects package, or a combination of those.
Cognitive aspect	
Constraint structure	Structure that expresses the elements that are involved in a constraint on an element, and what the actual constraint is. The constraint itself is typically specified in some existing formalism like Predicate logic or the Object Constraint Language.
Construction pattern	A mechanism that is used to define method ingredients, e.g., a pattern from one of the diagrams in the modeling constructs package, defined via the corresponding parameterized class.
Contained modeling concept	The assignation of a cognitive aspect to a modeling concept via a modeling construct. We can say that the modeling concept expresses the cognitive aspect via the modeling construct.
Domain-based specification	A specification that is written in terms of a domain specification, which means that the specification uses the concepts of a domain specification.
Domain model	A domain specification written in an unambiguous language, i.e., the domain modeling language. That domain modeling language has a defined grammar and syntax.
Domain specification	Specification of the concepts in a domain. The simplest form is a list of concepts in a glossary.
Guideline	
Interpretable domain-based specification	Domain-based specification where all terms are defined in a domain specification, or they are keywords of the modeling language.
Life structure	Structure that expresses the order of possible changes of an instance. Think of the steps in an Object lifecycle, the flow of a function, or the order of operations in

	an activity model.
Metamodel	A set of concepts that can be seen as an (abstract) language. The concepts have a defined meaning. A metamodel does not have to be complete or restrictive in any way. *TBD: Maybe we need to distinguish language and metamodel.*
Method definition	
Method ingredient	
Method Objective	
Method step	
Modeling concept	
MuDForM-based domain model	A MuDForM-based domain model is a domain model specification that is defined in terms of the MuDForM metamodel.
MuDForM-based feature model	Unambiguous domain-based specification where all literals are defined by a MuDForM specification. A MuDForM-based feature model is a context-free specification.
MuDForM context model	A model of the context of the domain and feature that are under consideration. A context model defines the concepts that are needed to define other MuDForM specifications. A context model can be seen as a black-box view on a domain and feature.
MudForM specification	Specification made according to the MuDForM metamodel.
Object Classifier	Expresses classifiers with instances that have an identity. The identity represents a value.
Object structure	Structure used to express static relations between instances that have to comply with the structure. Think of attributes and compositions.
Semantical aspect	An aspect that is covered throughout multiple MuDForM modeling concepts. This is not necessarily the same as the cognitive aspects.
Space classifier	Classifier that contains definitions of other classifiers, and form a namespace. Think of Domain, Features, and Contexts. Classes and Activities form a namespace for Attributes.

Specification	An identifiable statement about something, defined in a metamodel. The metamodel forms the language of the specification. The language includes a definition of the grammar, semantics, and syntax, although the latter is not required.
Value CLassifier	
Viewpoint	
MudForM	The MuDForM metamodel is a Metamodel that specifies the language (concepts and rules) for MuDForM specifications.

2.6 Domain model

A specification space with classifiers that define what instances with state can exists and in which changes those instances may be involved.

2.7 Context model

A specification space that contains concepts that are supposed to have a clear meaning outside the domains of features that are modeled. A context contains different types of concepts. First, mathematical concepts such numbers (integers, real,...) and there operators (+,-,/,..): formal concepts, and logical concepts like Boolean operators (and, or, ...) and predicate logic operators (forall, exists,...). Secondly, physical concepts like length, time, power, speed, and their relations. Thirdly, concepts that are taken from a domain that you do not want to model, but want to use objects from. In this category you define classifiers and constraints for their identifying terms. Think of Name, Address, Phone number. You need them to represent concepts that you are interested in, but you are not interested in how they change over time. A context may also declare Actors that a modeled feature has to interact with. The events that an Actor can generate or react to, form the hooks for connecting a feature to its operational context.

2.8 Feature model

A specification space that depends on one or more domain models, contexts, and other features. A feature defines what instances shall exist and what changes shall take place in the domains that the feature depends on. A feature specification consists of a functional decomposition and per function how it internally works (the function flow) and how it exposes itself to the environment (signature).

3 Cognitive aspects

This package defines the cognitive aspects that should be covered by MuDForM. The cognitive aspect stands for anything that the human mind is familiar with on an elementary level. We are not going to give a more stringent definition than that (here). We have chosen the aspects based after studying literature in language philosophy and cognitive psychology. The aspects form the foundation for the things that we want to describe with models. So the cognitive aspects are not modeling concepts themselves, but represent perspectives that the human mind uses to think (and communicate). The MuDForM modeling concepts will be a combination of language primitives (terminology and grammar) and the aspects we want to describe with them.
We are explicitly looking for the cognitive aspects, because many modeling methods are defined in a metamodel that is language-centric, i.e., most metamodels offer concepts and relations to create a grammar and a notation. For example, the concept of Event or Action is not present most metamodels. So, when one wants to be able to model behavior, one first has to define a language element that captures the syntax of an Event description or Action description.

The goal for MuDForM is to offer modeling concepts for aspects that are common in the human mind. This package is not an attempt to be complete in the presented set of aspects. The intention is to present an initial set, such that it can be used to show how the modeling concepts are created from it and justify the chosen modeling concepts. For this purpose, the aspects should preferable not have any overlap in their meaning, i.e., they should be orthogonal. A metaphor is that the set of aspects make up a vector space with a dimension for each aspect, in which modeling concepts can be seen as a specific vector. (This does not completely hold, because some aspects are specializations of another aspect, and thus are a kind of sub-dimension.

3.1 Cognitive aspects diagram

Cognitive aspects are things that have their own meaning (semantics) for the human mind. The intention is to introduce elementary aspects that the human mind uses to understand and reason about the world.
The diagram introduces the notion of concept, and a number of derivatives. These aspects form the primitives from which the modeling concepts are derived. The subclasses of concept are non-disjoint. So, for example, something that is an Attribute can also be an Entity. If all possible combinations of subclasses are relevant is not important here.
The individual definitions of the classes in the diagram explain their intention in more detail.
We have used a UML class diagram to show the coherence between the cognitive aspects. However, the cognitive aspects are not intrinsically present in the human mind as classes, associations, and generalizations. So please, do not see them as such. These are some perspectives on the choice for UML to model the cognitive aspects:

- First of all, it is chosen for its familiarity.
- The aspects are depicted as classes, because they all can be seen as something that represents many instances. Furthermore, the generalization to the class concept is useful to depict that all aspects are a special kind of concept.
- The meaningful associations are used to keep the diagrams simple. The disadvantage of UML is here that UML does not allow for relations between association, which in some cases would be handy. For example, there is a relation between "order", "causes", and "performs". But it is not possible to model it graphically in UML (without introducing association classes).
- The role names on the associations allow to use the subsets qualifications. For example, to state that "identifies" is a special kind of "refers to".

Open issues:

- *Add the concept "constant"? A constant is an Entity that is never changed, (but can be involved in an event). E.g., the amount 5 (not the symbol "5") can be involved in Events, but always stays the amount 5 and doesn't change. But, if you want to keep track of how many animals have 5 toes on a foot, then fiveness is an object that undergoes changes. Less abstract: the name Robert always stays the name Robert, unless you are interested in how popular names are, i.e. a name has a popularity. Then naming a person Robert, changes the state of the name Robert. So, a Constant is an Entity that doesn't have a Life (= changing state). In programming languages a constant is a term that identifies an entity. For example, the constant "number-of-toes" may be set to the value "5", which means wherever number-of-toes is stated, the value 5 is meant. For now, a constant is an entity that is not an object.*
- *There is a relation between "property" and "involves", i.e., an "Event involves an Entity" can be seen as the "involvement is a property of the Event".*
  Example: paint the wall blue
  (Event) paint (involves Entity) the wall (and involves Entity) blue (Entity)
  can be seen as
  (Property) blue (of Event) paint (and Property) the wall (of Event) paint.
  Maybe this can be solved by seeing the role "involved" as a subset of the role "property"

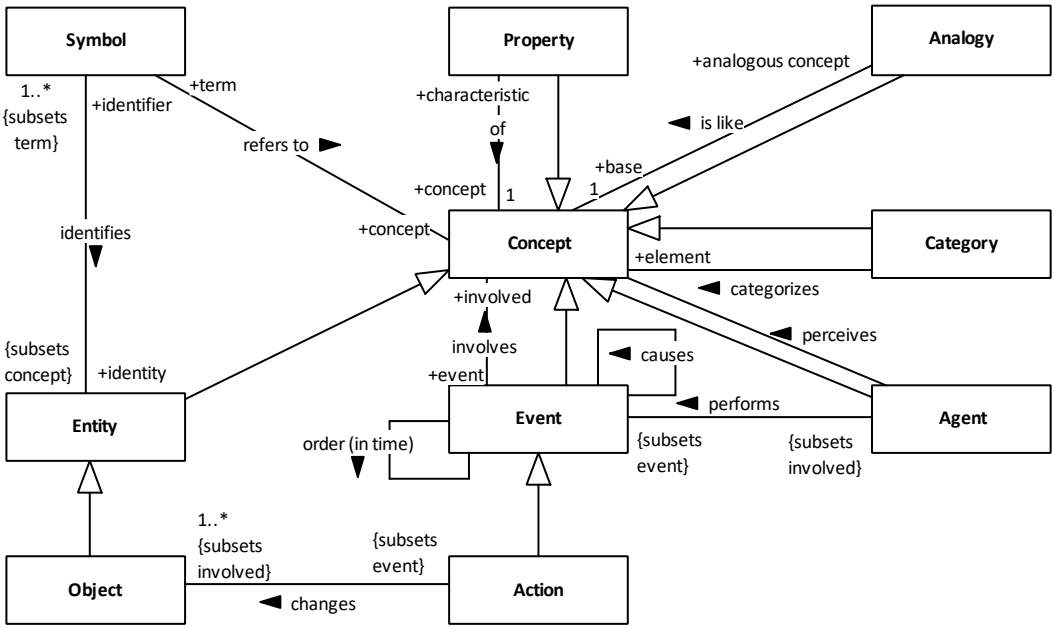


Figure 6: Cognitive aspects

3.2 Statements diagram

A statement is a concept that says something about other concepts. A statement contains Symbols which refer to the concepts that the statement is about in the statement's context. The statements are written in a language, which is a property of the statement's context. The grammar of the language in combination with the symbols, makes a statement interpretable.
A descriptive statement describes concepts as they are (perceived). Descriptive statements are used for analysis, i.e., understanding something. A prescriptive statement describes what should be true for the concepts it is about. They are used for design, i.e., describing something that does not exist yet. This distinction is less driven by cognition, but more by the need for a specification language for developing systems, which involves understanding a system's context through an analysis and describing what you want to create through a design.
For example, the descriptive statement "my car is blue" says something about me, my car, blue, and about being. An example of a prescriptive statement is "my car must be blue", or "I must choose between red and blue for the color of the car that I want to order".
We distinguish descriptions from prescriptions, because we both want to make an analysis of the world we need to operate in, as well as make a design of how we want the world to be.

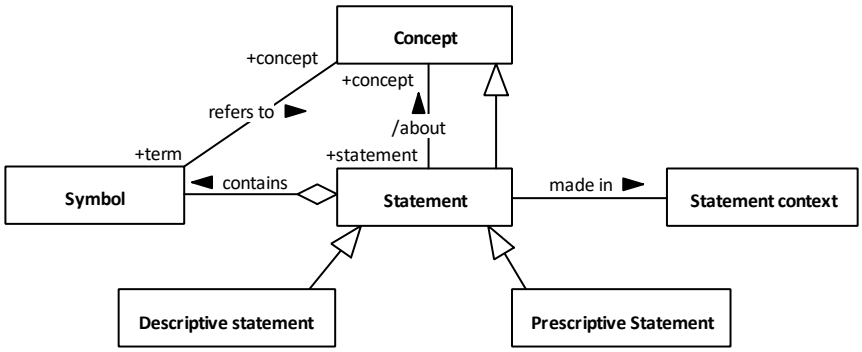


Figure 7: Statements

Name	Definition
Analogy	A concept that is like another concept, i.e., the base concept. In practice, the analogous concept is a structure in which all the attributes are like the attributes of the base concept. For example, a tree with branches is like an animal with

	limbs. The tree is the analogous concept and the animal is the base concept. Branch is the analogous attribute and limb is the base attribute.
Statement context	A statement context stands for the things one should be aware of to understand a statement, for example: • The one who made the statement. • The document, or some other knowledge container, in which the statement is made, • Other statements in the same context. • The language in which a statement is made. This implies that each statement has a unique statement context.
Action	An event that changes one ore more objects. Besides that, the action may still involve entities that are not an object. So, actions are creating (=identifying) and linking concepts, and as such entail behavior that is known to some extent.
Agent	A concept that can cause events and can perceive concepts. If the cause event is an action, then we can say that the actor performs the action.
Category	A concept classifies other concepts, which are called elements of the category. Categories are ubiquitous in all specification languages, e.g., simple operations, activities, functions, attributes, classes, and data types. Typically, nouns and verbs are symbols in natural languages, which refer to categories. For example, "chair" refers to a set of things that we consider to be a chair. But using and recognizing categories is not limited to things that can be referred to by one word. For example, the category of things on my refrigerator.
Concept	Something that the human mind can think of. Concept is the most abstract and fundamental class in the metamodel. No things that you want to model can only be a concept, because a model requires at least symbol to indicate the concept. (Refer to Mentalese and to Chomski.) To communicate about a concept, one might use multiple Symbols that refer to the concept. Concept is the root of all aspects. It corresponds with concept in Chomsky's triangle of human language [ref]. This triangle states that the mind thinks in concepts, communicates using terms to refer to those concepts, and the concepts, and terms reflect, but are not limited to, things in the world outside the mind (often called reality). MuDForM is intended to be close to the different aspects that the human mind inherently uses in thinking and communication. What those aspects are is not a scientific fact, but much literature gives hints (ref). Furthermore, natural language is evolved by humans over time, and is used to communicate for many things that the human mind wants to communicate. We use this observation to look for concepts in natural language. But we are not limited to it, because people can reason without language. Some other approaches speak of "modeling reality". But MuDForM doesn't follow this perspective, because is requires an assumption about what reality is. MuDForM focuses on what the human mind perceives. Accordingly, if you like, you may call something that most people perceive as a fact, reality.
Descriptive statement	A statement that describes (existing or known) concepts, which is useful to understand such concepts (analysis).
Entity	A concept with at least one unique identifying symbol (ID). So, an entity is an identifiable concept.

	An Entity is a recognizable concept identified by at least one symbol. Entities are not dependent on any category. So, entities exist without having them assigned to an explicit category. The concept of Entity brings the possibility that we can distinguish things in our perception as separate things and that we can refer to those things via their identifying symbol (name, number, picture of it, pointing to it, ...). With identification also comes also equality. Two entities are (semantically) equal if their identifying symbols are equal.
Event	A concept that expresses something that happens (in time). Events can have a duration of zero, i.e., they are atomic in time, or they can have a start and stop moment. An Event may involve zero or more entities. Events may have some order (in time) to each other, like sequentially, or simultaneously, or one event happens during another event. One event can cause another event. The tendency to look for causal relations is very common in people (and other evolved animals). An event can be performed by an actor, but that is not always the case. For example, the event "I knock the glass from the table" causes the event "the glass falling on the floor" (under the circumstance of gravity). Who is the actor of the second event? It is weird to say that the first event is the actor of the second event. But apparently there is some relation between "causes" and "performs". There is also a relation between "causes" and order, because an event cannot be caused by an event that happens later in time. *TBD: validate if the current model regarding order, causes, and performs suffices, because they are not orthogonal and independent aspects.*
Object	An Entity that is changed by Actions, i.e., has follow-up Actions. An Object exist over a period of time and therefore, we can speak of its state. One can say that an Object is not alive (a.k.a. is dead) when it cannot be changed anymore by any Action. However, a dead object can still be involved in Events, but such an Event is not changing the Object.
Prescriptive Statement	A statement that prescribes a concept, i.e., it states what should be true about the concept, which useful to convey information about what should be realized (designed).
Property	A concept that belongs to another concept, i.e., it is a property of that concept. Typically, we use attributes to denote characteristics, and determination of time, location, possession, or composition. Property can also be seen as the aspect that reflects coherency of matter. For example, if you step into your car and you get out after a while, then you expect that the wheels are still on the same location and that the color has not changed. The car wheels and car color are both properties of the car. In this case, one could also say that a car wheel is an element of the category wheel, and a car color is an element of the category color.
Statement	A Statement is a concept that contains symbols. A statement is about concepts that the Symbols in the statement refer to. A statement can be ambiguous when a symbol in the statement is not an ID for a concept, but just refers to the concept. The notion of ambiguity of statements is relevant, because human communication can be ambiguous, and removing ambiguity is an important aspect of capturing knowledge with MuDForM.
Symbol	Something that can be perceived and uttered by the human mind, and refers to a concept. Symbols can have different forms like a word, an icon, a sign, a photo, a sound, etc.. In modeling, a word or phrase is the most common symbol form,

	often combined with a modeling concept like a class, activity, or attribute. Notice that a Symbol is not a concept, because we only use it to point to a concept. The symbol itself does not have meaning. A symbol is called an identifier when it identifies and entity.

4 Construction patterns

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

- Names in natural language, which enable model elements to have an identifying term in natural language, such that people can talk and communicate about a model element.
- 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.
- Metamodel constraint patterns, which can be used throughout the metamodel to put restrictions on allowed instances. For example, model elements must have the same parent or grandparent, if relation R exists between A and B, then also a relation Q must exist between A and B, or every child A of B, must also have a C in under B. *TBD: these patterns are not in the metamodel yet. But can be added later when more consistency rules are identified.*
- Notation conventions, which explain how the used notation must interpreted.

4.1 Names

Models should usable by people, which implies that it is easier if model elements have a recognizable symbol to identify them and a description to elaborate on that. This is strictly seen not necessary for formal specifications, because there is no formal semantics in a name or a description. But it is handy for practical use of MuDForM models and can be used in situations where it is not the goal to make a complete MuDForM specification of all model elements.

4.1.1 Named elements diagram

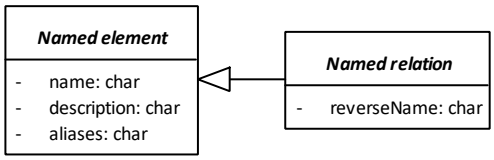


Figure 8: Named elements

Name	Definition
Named element	Anything that can have a name in a model. Typically, a name should be unique within the scope of the named element. **attribute:** name A term (word or compound word), which is used to identify the model element. **attribute:** description A description can be used to clarify a model element to people that do not want to see the full model around some elements or in case there is relevant information that is not captured in the model. **attribute:** aliases A list of other terms that can be used to identify the model element.

Named relation	Relations can also have a reverse name. **attribute:** reverseName How the relation would be identified if read in opposite direction.

4.2 Structure patterns

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.
- Coordinated structure: A Structure in which the structure type defines which instances of the Elements and instance of the Structure may or must have.
- 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 can be seen as 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.

4.2.1 Structure diagram

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.
For example:

- person *has attributes (structure)*
- person.identifier isa attribute/element.
- Person.identifier isa substructure
- person.identifier.kind = one of
- person.identifier.name is an attribute
- person.identifier.name is a reference to Name
- person.identifier.socialnr is an attribute
- person.identifier.socialnr is a reference to SocialNr.

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 in a structure (not presented in the diagram). For example:

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

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

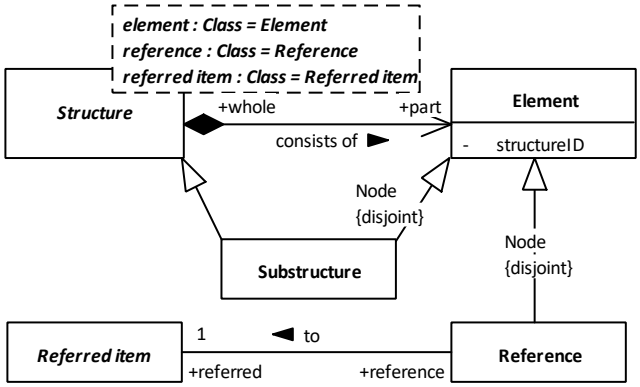


Figure 9: Structure

4.2.2 Coordinated Structure diagram

A coordinated structure is meant to constraint and prescribe the possible elements of a structure of a classifier. The Coordinated 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).

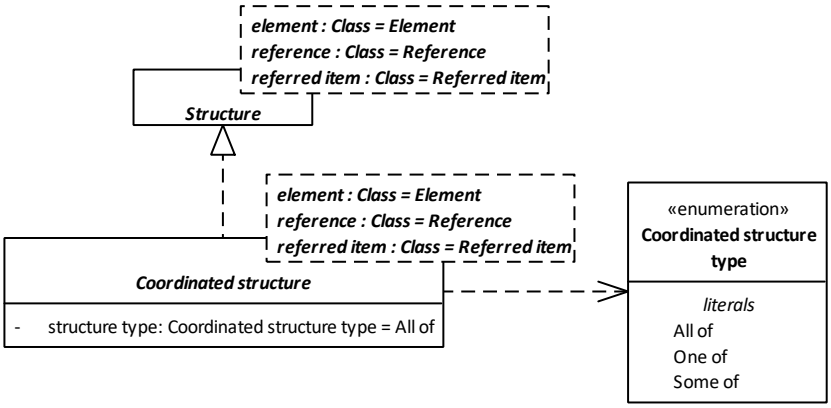


Figure 10: Coordinated Structure

4.2.3 Static structure diagram

A static structure is a coordinated structure, in which also the multiplicity of each (sub)element is specified. This can be used when the elements are also typed by a classifier. The attribute "how many?" specifies the possible amount of instances for each (static) element:

- One, meaning exactly one. This is the default value.
- Zero or more. So, no restrictions.
- At most one. So, zero or one.
- At least one.

Any other restrictions can be specified with a constraint (precondition, postcondition, invariant).

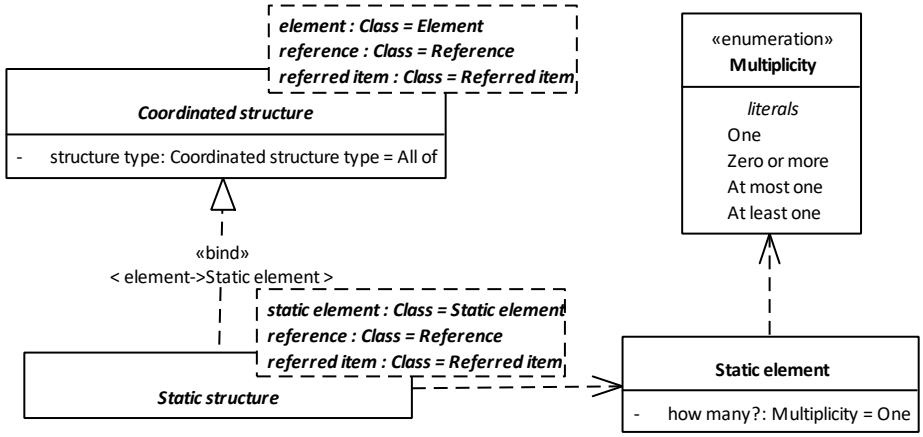


Figure 11: Static structure

4.2.4 Flow structure diagram

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.

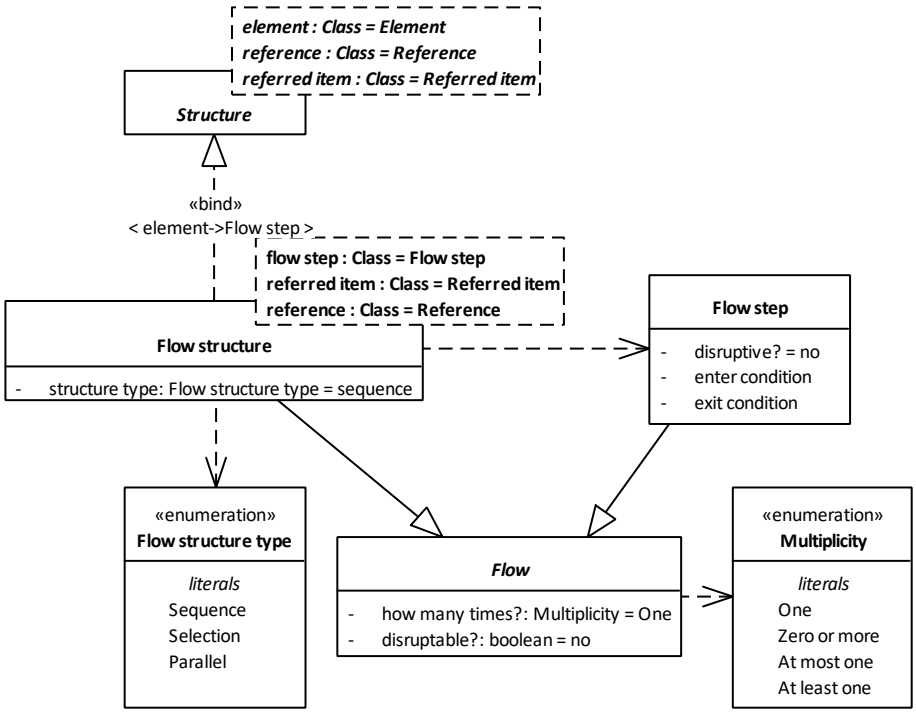


Figure 12: Flow structure

4.2.5 Reference structure diagram

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 Reference 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) has a structure which 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 of F and the function call of G must match. i.e., they are the same or the type of the Element is a subclass of the type of the Reference element.) *TBD: The parameterizes relation can be an expression in some formalism. For example, step.participantA := funtion.attributeX + function.attributeY. This is not covered yet.*

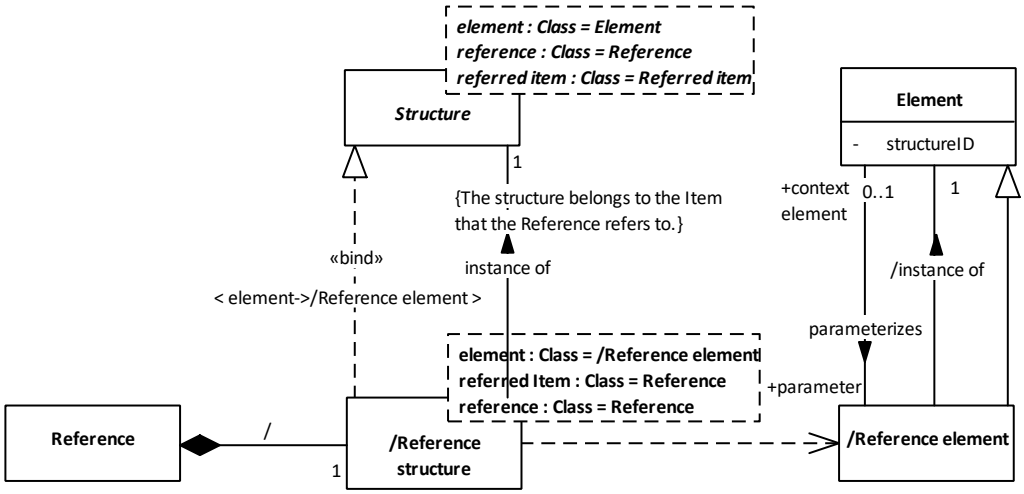


Figure 13: Reference structure

4.2.6 All structures diagram

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

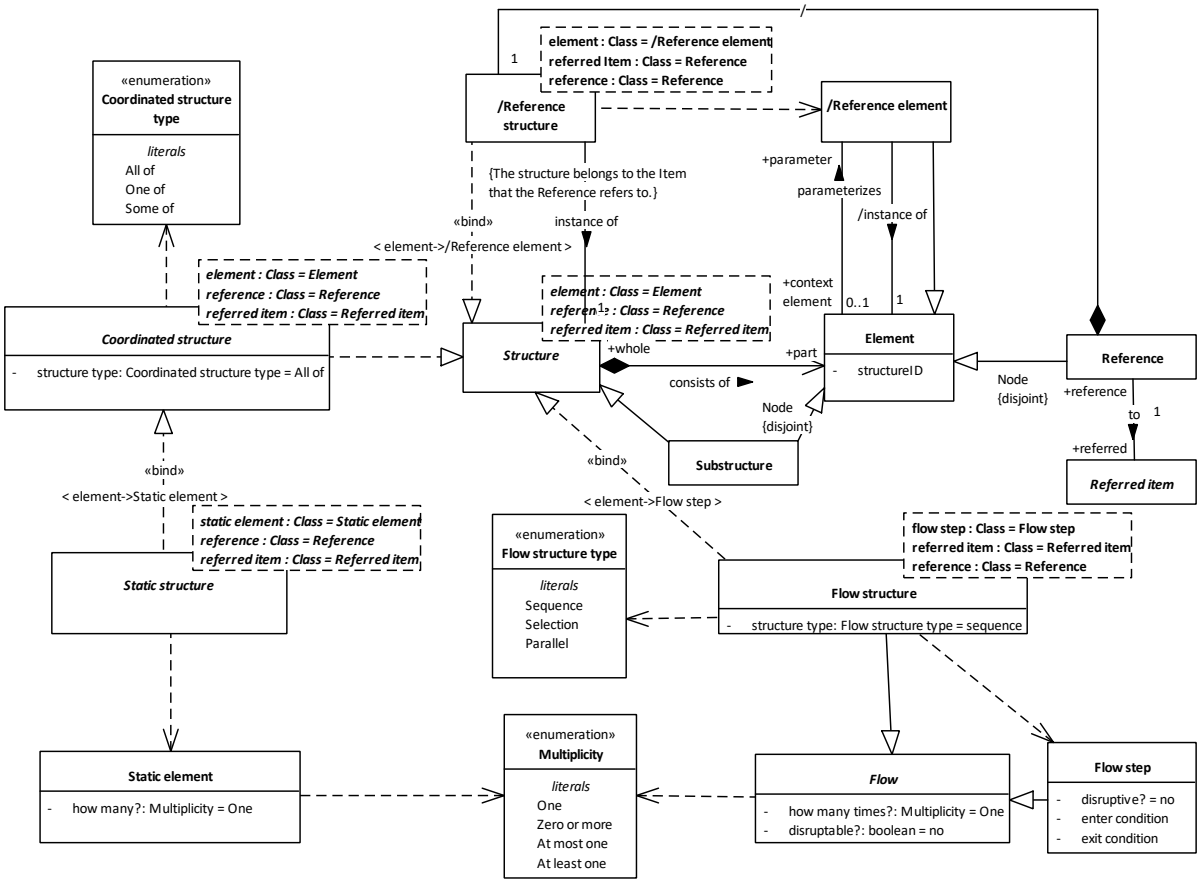


Figure 14: All structures

4.2.7 The other structures diagram

This diagram is made for the method engineering paper.

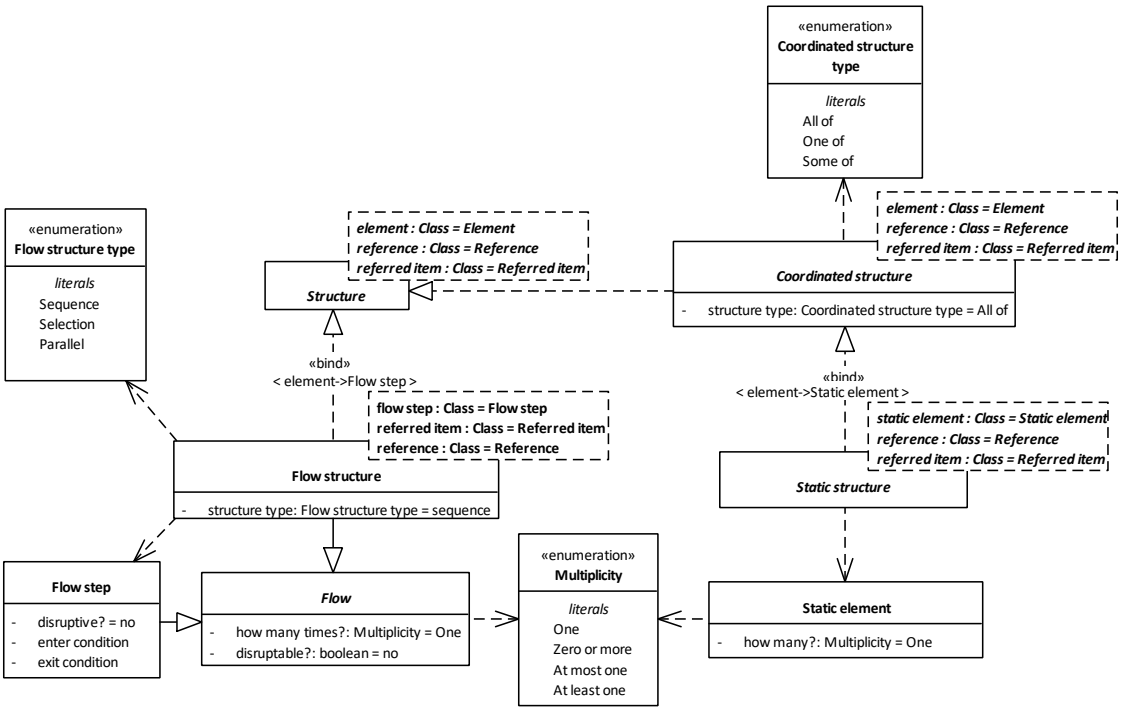


Figure 15: The other structures

Name	Definition
Coordinated structure	A Structure in which the structure type defines which instances of the Elements and instance of the 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
Element	An identified part of a structure. The structureID identifies the element within 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 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? Specification of how many times the Flow can or must be executed. The Multiplicity enumeration offers the most common ones. The specification might be a condition that acts as a guard on the iteration of the Flow. **attribute:** disruptable? 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 Referred item. **attribute:** disruptive? 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 What must be valid to enter this step. **attribute:** exit condition 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 The temporal relation between the steps.
Referred item	A concept that has an identifier and that can be referred to. It is a placeholder for all the things you can refer to from an Element in a Structure, which would make 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. **attribute:** how many? Specification that states how many values/instances the Static element may have. The Multiplicity enumeration offers the most common ones. The specification might also be a condition that acts as a guard on the size of the set or list.
Static structure	A Coordinated structure in which the Elements are Static elements.
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, it is used for the nodes in the structure that are not a leave, but have other nodes, possibly leaves under them.
Flow structure type	Default set of possible types of a flow structure. **attribute:** Sequence All the Flow steps in the Flow are executed after each other. **attribute:** Selection Exactly one of the Flow steps in the Flow is executed. **attribute:** Parallel All the Flow steps in the Flow are executed concurrently (in arbitrary order).
Multiplicity	 **attribute:** One Exactly one instance must be present. **attribute:** Zero or more Any number of instances is allowed. So, no restrictions. **attribute:** At most one Zero or one instance is allowed. **attribute:** At least one At least one instance must be present.
Coordinated structure type	Default set of possible types of a Coordinated structure. **attribute:** All of Indicating that all elements in the static structure should have a value/instance.

	attribute: One of Indicating that exactly one of the elements in the structure should have a value/instance. **attribute:** Some of Indicating that one or more elements in the structure should have a value/instance.

4.3 Formalisms

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. UML itself is also based on set theory. But MuDForM does not have the same semantics everywhere the same term is used.
- 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.

4.3.1 Formalisms diagram

MuDForM offers the use of available formalisms for the detailed specification of 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 well-known and 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.

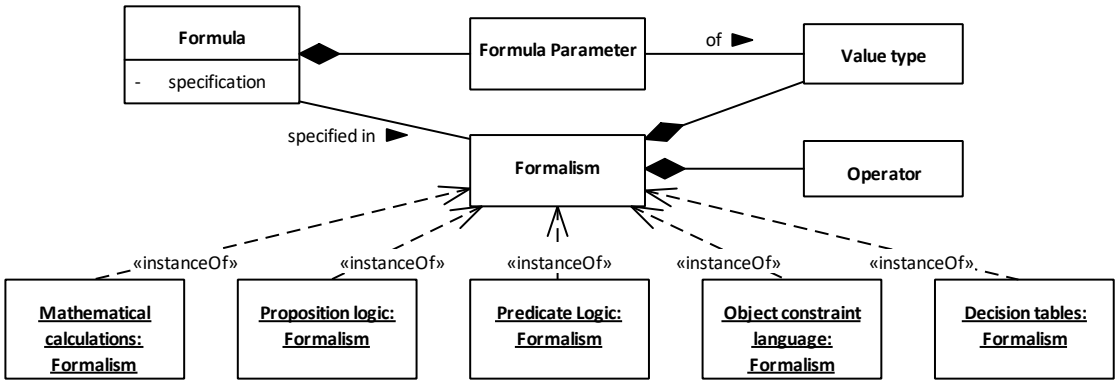


Figure 16: Formalisms

Name	Definition
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
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 for example be used to formalize decisions 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.

4.4 Notation conventions

Modeling conventions throughout the MuDForM metamodel:

- Associations and attributes can be derived. But there is no option, in EA or UML, to say that a class is derived. We indicate a derived class the same as a derived association: its name is prefixed with a "/". A derived class must have a constraint (called the derivation) that specifies which objects are element of the class.
- We use a dependency relation between a class that binds a another class, and the classes that are used as actual parameters of the binding. For example, a Object lifecycle structure depends on an Involvement, because in the definition of Object lifecycle the definition of Involvement is used. Namely, in the binding between Object lifecycle and Flow structure, Involvement replaces referred item.
- 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.
- Unless specified otherwise, generalizations to the same parent class are non-disjoint.

5 Modeling concepts

This package defines all the MuDForM modeling concepts. Modeling concepts are the types of elements that make up a MuDForM compliant model, i.e., model elements are an instance of a modeling concept. 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).

5.1 MuDForM modeling concepts diagram

The modeling concepts are captured in four packages:

- The generic concepts, which are used as a basis for the concepts defined in 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.

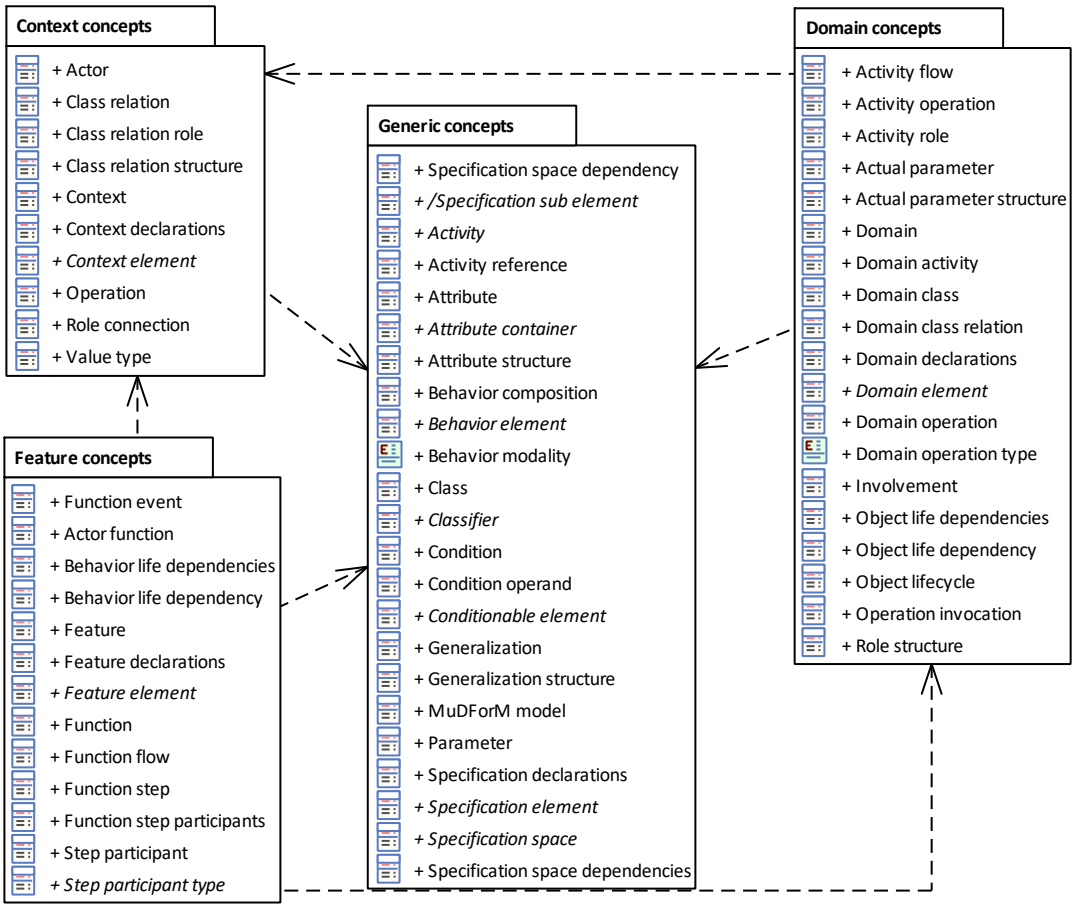


Figure 17: MuDForM modeling concepts

5.2 Sub packages of the package Modeling concepts diagram

This diagram presents the sub packages of the package Modeling concepts.

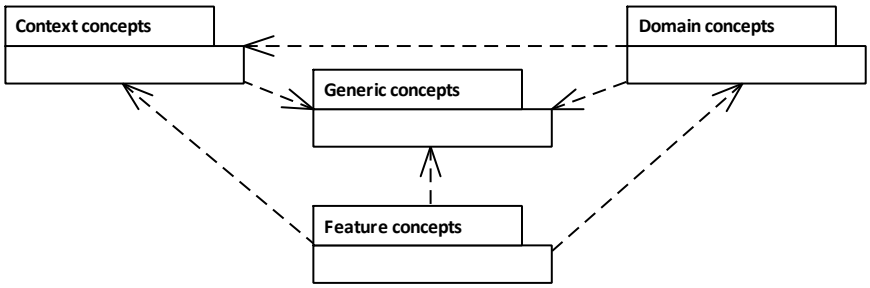


Figure 18: Sub packages of the package Modeling concepts

5.3 Generic concepts

This package defines the concepts that can be used all three different types of specification spaces, i.e., contexts, domains, and features, which modeling concepts are defined in the corresponding packages. This package also defines the concepts that form the basis for the definition of concepts defined in those other packages.

5.3.1 Specification Spaces diagram

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 objects with states, called domain classes, can exist, and in which changes (actions), called domain activities, those lives may be involved. Domain models may depend on context models and other domain models.
- A feature model defines what instances shall exist and what changes shall take place in the domains that the feature depends on when the feature is active. Feature models may depend on one or more domain models, other features, or contexts.

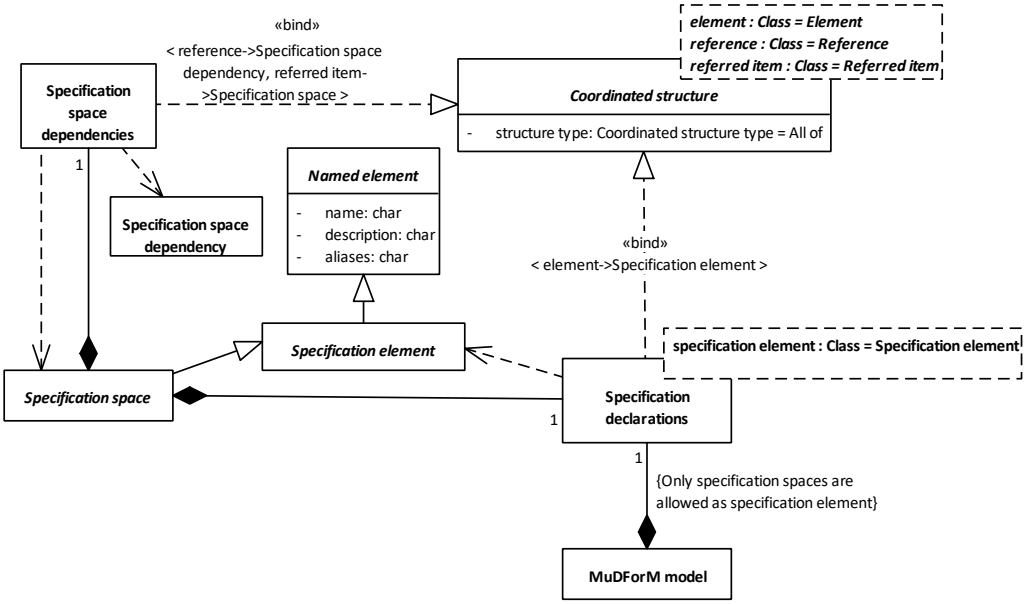


Figure 19: Specification Spaces

5.3.2 Attributes diagram

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.

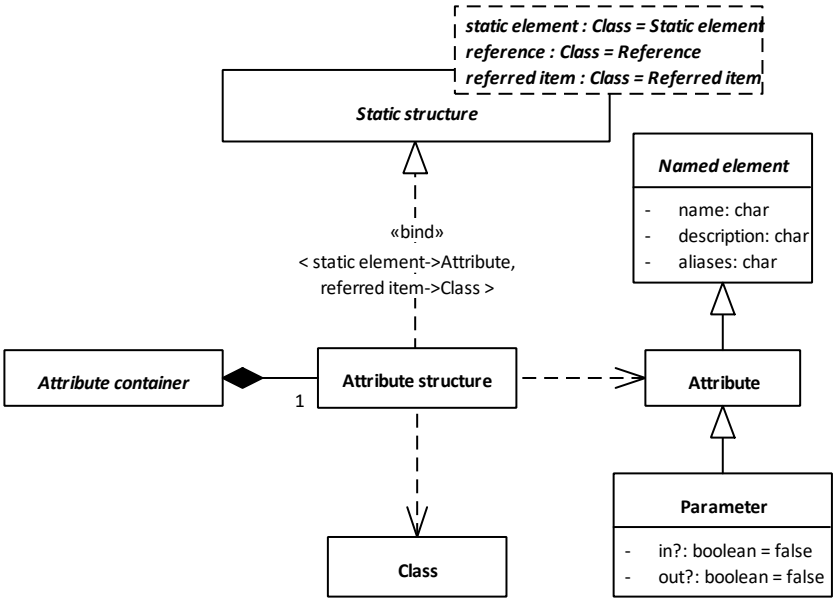


Figure 20: Attributes

5.3.3 Classifiers diagram

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's coordinated 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.

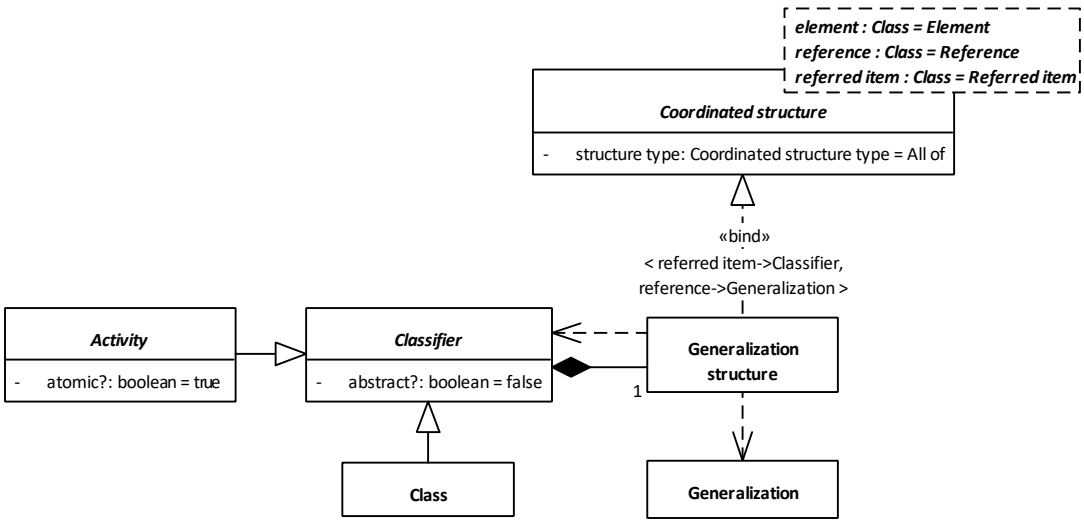


Figure 21: Classifiers

5.3.4 Behavior composition diagram

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.

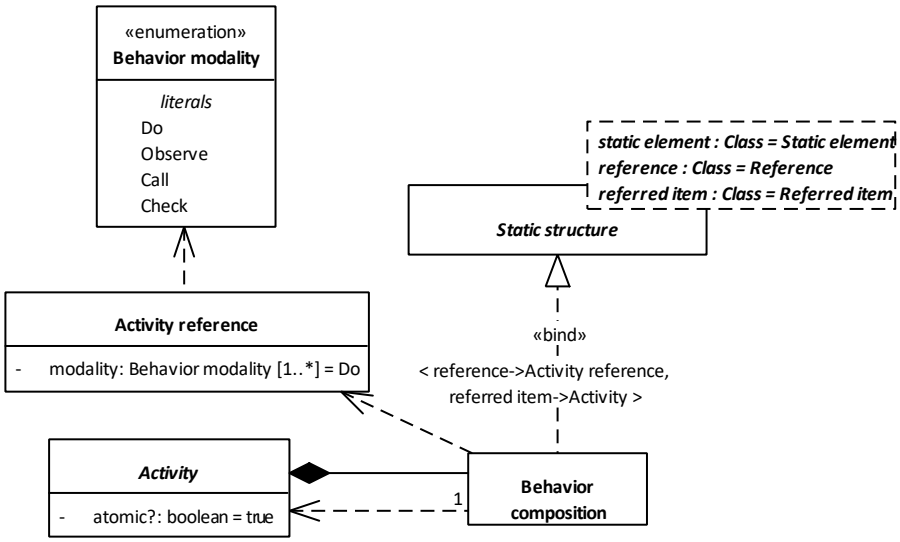


Figure 22: Behavior composition

5.3.5 Conditions diagram

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. (Conditionable) behavior elements can also have a precondition and a postcondition.

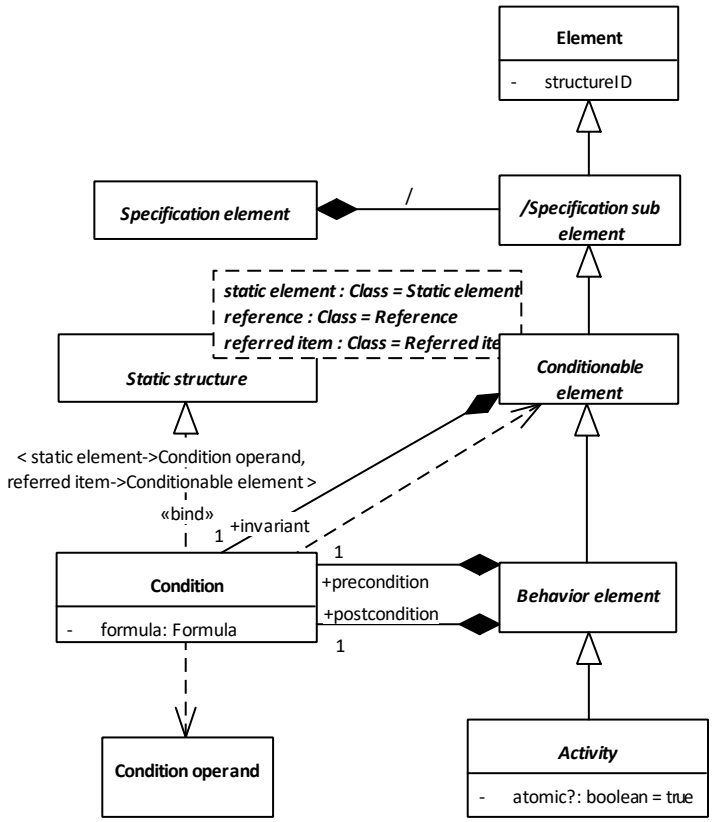


Figure 23: Conditions

5.3.6 Different specification spaces diagram

This diagram is not a main diagram. It just shows how the abstract concept specification space is used.

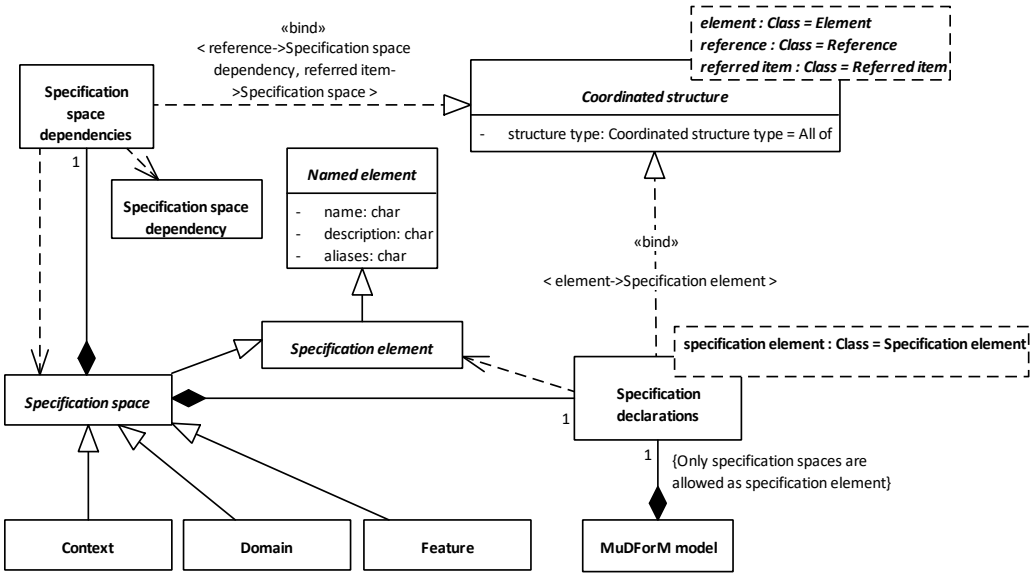


Figure 24: Different specification spaces

Name	Definition
/Specification sub element	Any Element of a Structure of a Specification element can be a conditionable element. As such, the Specification sub elements of a Specification element are

	derived, and defined by the union of all the elements in any structure that is contained by the specification element. Note that all specification elements are also specification sub elements, because they are an element in the declarations of a specification space.
Activity	Any behavioral element that is executable, which means it can be a referred Item in an activity flow 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? 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	A reference to an activity that another 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 some languages that use the event concept, in which events are global and can be defined without any restriction to their scope). Activity references can be used to represent things like: • 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 an activity. *(Disclaimer: this kind of activity reference might need some extra constraints to be modeled.)* **attribute:** 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.
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.
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? An abstract classifier does not have elements (instances) 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 the classifier structure of the abstract classifier, or by a classifier structure in which the abstract classifier is referred to by a generalization.
Condition	A Static structure in which the static elements are Condition operands, and the referred items are Conditionable elements. A Condition states what must be true for its container. All Conditionable elements can have invariants. Moreover, 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. **attribute:** formula specification of the 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 specification sub element. Example: Invariant of person: person.age > 0. The Condition operand person.age in this condition is bound to the specification sub element person.age.
Conditionable element	Any element that is part of a structure, i.e., is a specification sub element, can be identified, and as such can have invariants. An invariant states what always must be true for the element.
Generalization	Reference to a classifier in a classifier structure of a classifier, meaning that the containing classifier is a subclass of the referred classifier.
Generalization structure	A static structure of a classifier, in which the references are generalizations, and the referred items are classifiers.
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?*
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? Does the parameter get a value from the environment of the behavior element? **attribute:** out? Does the behavior element give a value to the parameter, which can be seen from the environment of the behavior element?
Specification declarations	Coordinated structure belonging to a specification space, in which the elements are specification elements. The declarations contains all the specification elements that are declared in its specification space.
Specification element	Specification elements are things in a specification space, which have their 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. • Contexts 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.
Specification space dependency	
Behavior modality	Enumeration of the possible modalities for an Activity reference. **attribute:** Do Execute the Activity. So, the actor that executes the activity also executes the activity referred by the activity reference. **attribute:** Observe React to the execution of this activity in the environment of the containing activity. **attribute:** Call Tell the environment to execute the activity referred to by the activity reference. **attribute:** Check Check if the activity referred to by the activity reference can happen, according to all constraints that apply to this action. *(Disclaimer: the usefulness of this one is doubtful)*

5.4 Context concepts

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

5.4.1 Context definition diagram

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

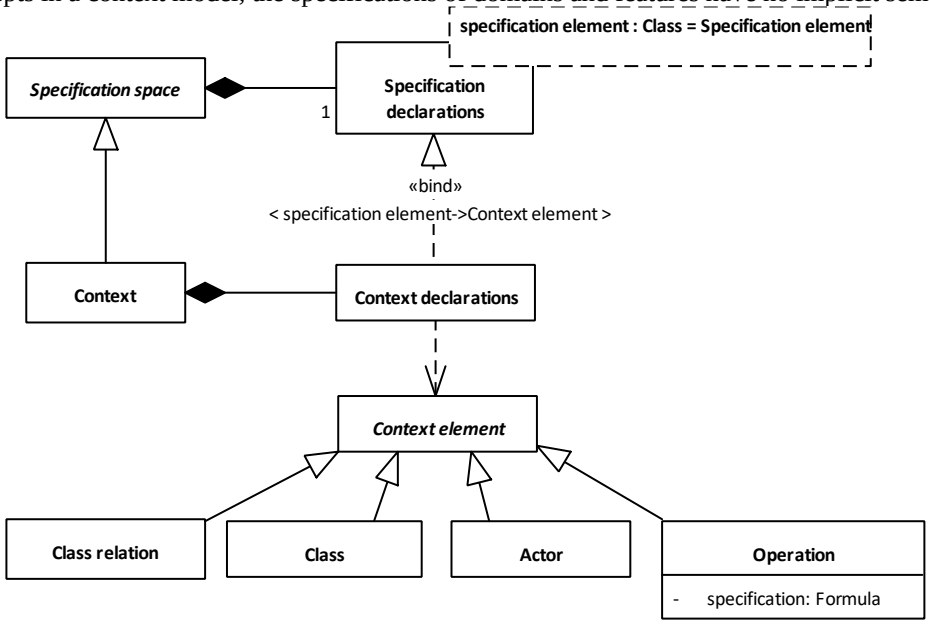


Figure 25: Context definition

5.4.2 Context-specific concepts definition diagram

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 whose 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.

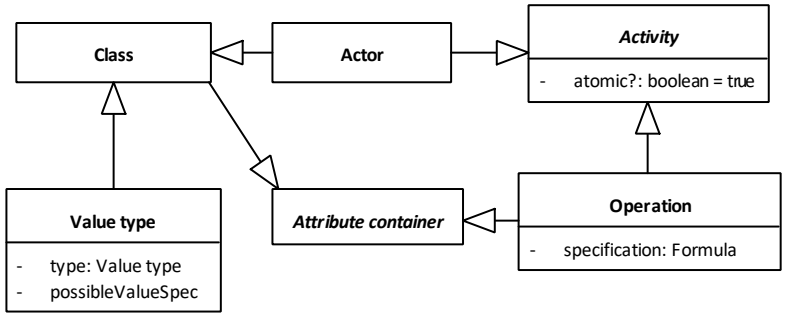


Figure 26: Context-specific concepts definition

5.4.3 Class relations diagram

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.
To be clear, MudForM distinguishes different kinds of relationships between classes:

- A subset of the class relations are the domain class relations, which are instantiated when two or more objects participate in the same action.
- lifecycle dependencies: 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.

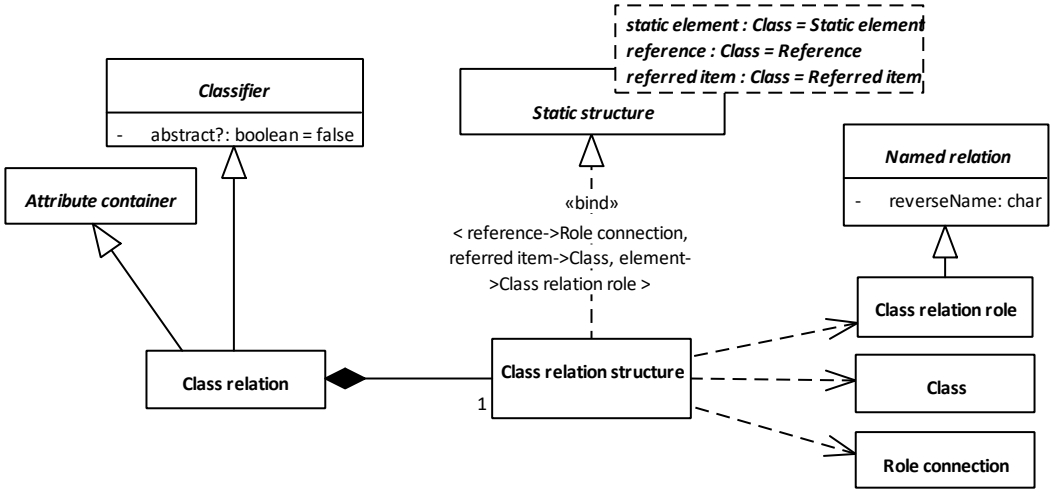


Figure 27: Class relations

Name	Definition
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	Class relations are used when one wants to express and communicate a relation between two or more classes, and none of the classes can be seen as the container of that relation. (Namely, in such a case the relation is an attribute in the attribute structure of the container class.) A Class relation is an Attribute container and a Classifier. 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.
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. **attribute:** specification 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 The type of the values of this value type. That type is typically from one of the used formalisms. **attribute:** possibleValueSpec A specification of the possible values of this Value type, e.g., a range of values, or an enumeration.

5.5 Domain concepts

This package defines the modeling concepts that can be used in a domain model.

5.5.1 Domain definition diagram

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.

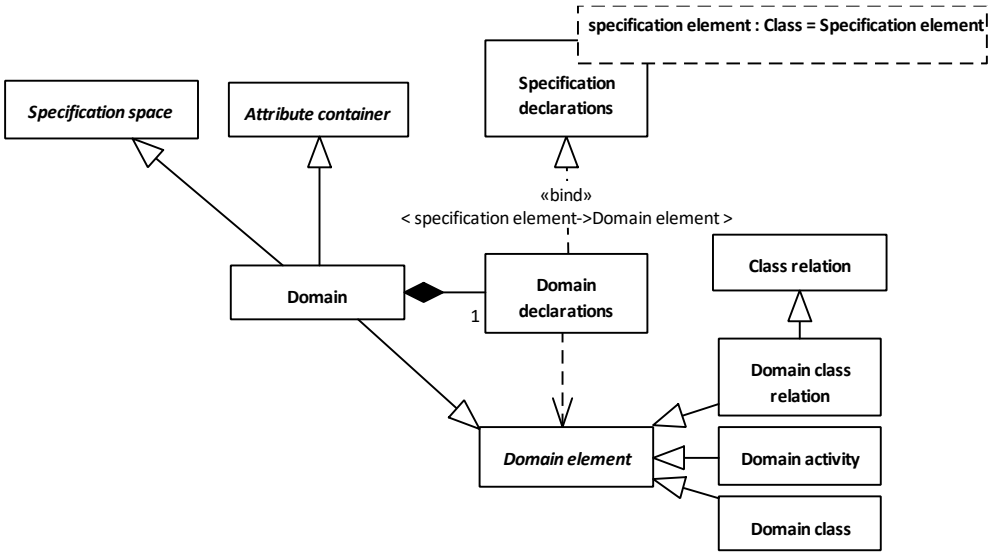


Figure 28: Domain definition

5.5.2 Domain class definition diagram

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 cannot 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 and must be alive when it is a domain object.

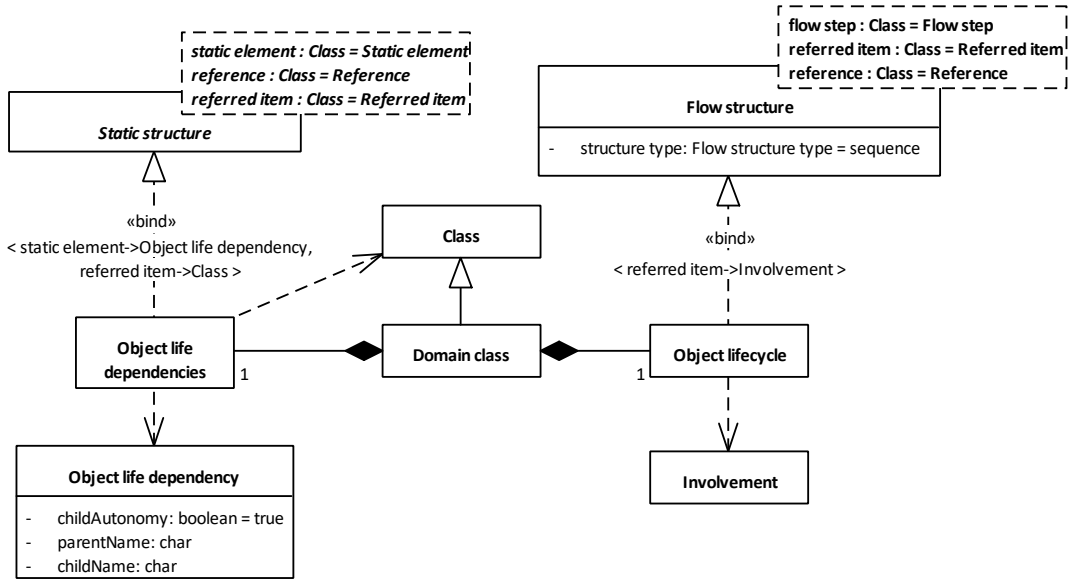


Figure 29: Domain class definition

5.5.3 Domain activity definition diagram

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 involvements are Partial behavior elements, which means they can have their own separate pre- and postcondition.

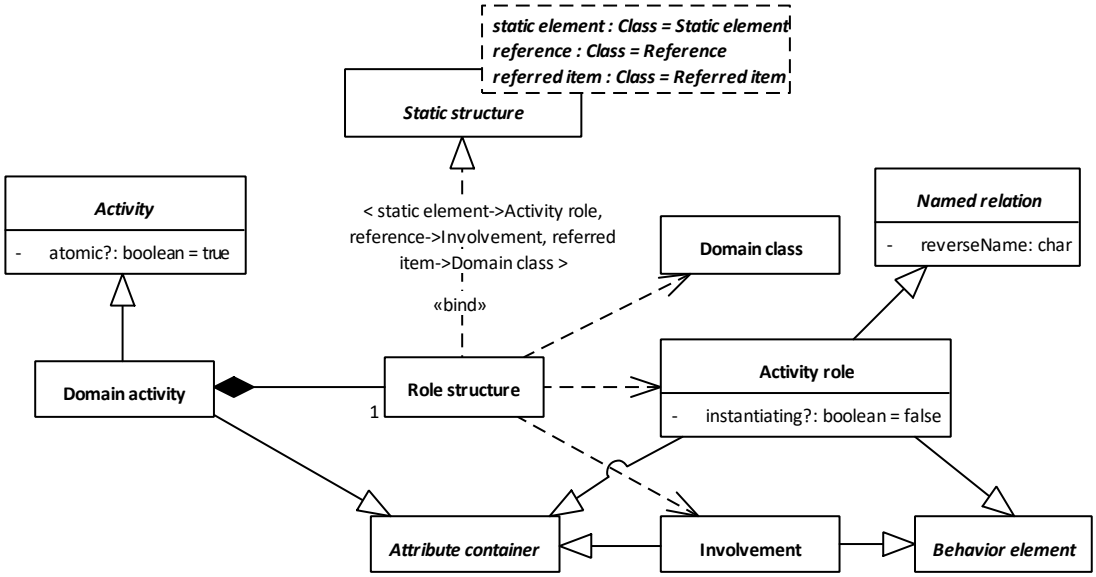


Figure 30: Domain activity definition

5.5.4 Activity flow diagram

A Domain activity may have an activity flow, which expresses the behavior of the activity. The activity flow 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.

Disclaimer: the part of the metamodel in this diagram is the least tested. But It is also the least interesting, because it resembles normal programming languages. The main difference is the availability of the "Domain operation types", which captures the kind of changes that result from the notion of domain in MuDForM.

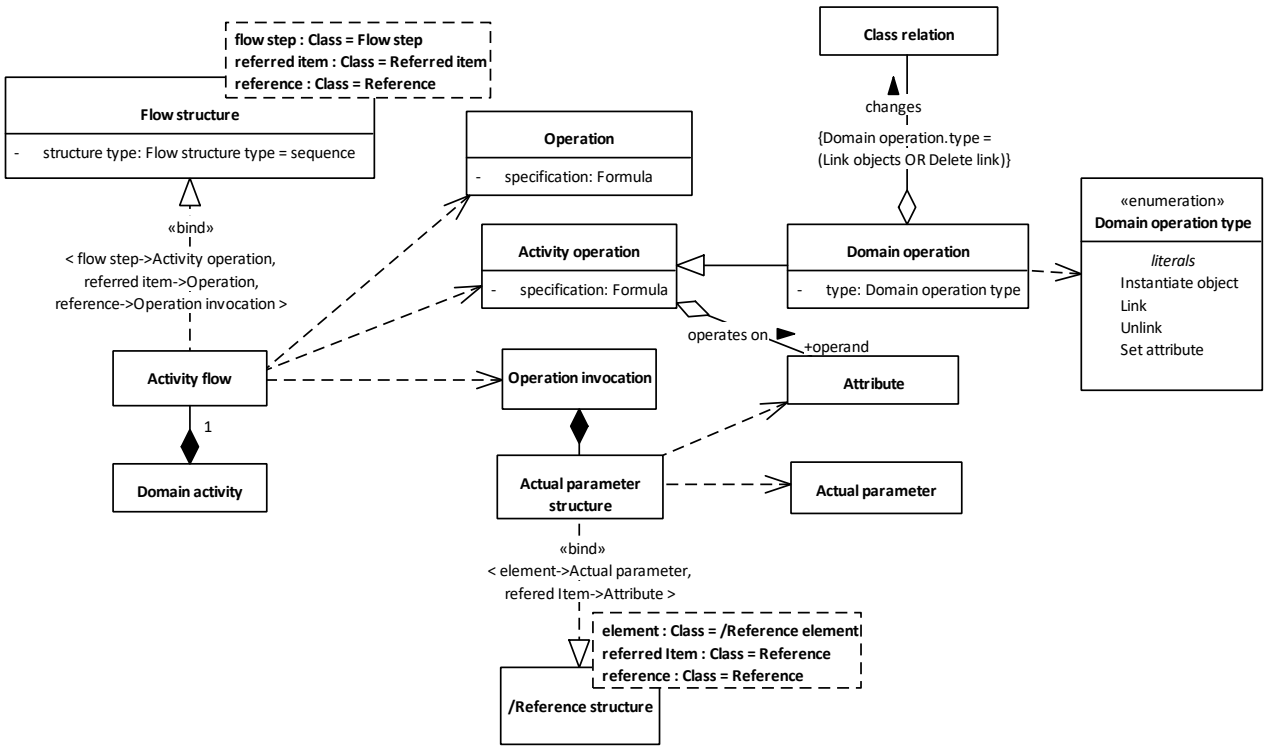


Figure 31: Activity flow

Name	Definition
Activity flow	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 expresses the internal behavior of an activity, i.e., what happens when the domain activity is executed.
Activity operation	An operation that is defined in the scope of an activity flow. The behavior of the Activity operation is specified in terms of the supported formalisms. **attribute:** specification
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? 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.
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 a Specification space and an Attribute container. A domain is an area of activity or knowledge, which 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 default. 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). A domain class relation instance can be broken by participation in a domain activity instance. The relation instantiation or break is specified in the activity flow via a Domain operation of type Link or Unlink respectively. There is a strong similarity between non-atomic activity and domain class relation. Difference: a 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
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 static elements are life dependencies and 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 from a domain that the containing domain is depending on, or from the containing domain itself.
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. In other words, the child object does not have meaning without the parent object. When the childAutonomy is true, then the parents' 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 then also the child cannot participate in an action anymore, i.e., the child object's life is also ended. *TBD:* *What to do with objects from another domain that participate in the instantiating action, but that do not have a life in the domain? Answer suggestion: if they have a life in another domain, then the action should also be present in the OLC of the parent. But then the questions is, in which domain should the activity be placed? Answer reasoning: apparently there is some larger domain that contains that action. If a domain object has a "foreign key" then there are two options. the foreign object can be deleted as thus is nothing more that a referring attribute. Or the foreign object may not be deleted, which is a problem.* *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.* **attribute:** childAutonomy 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 The role name of the parent, i.e., the containing class, in the dependency relation. **attribute:** childName 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 domain 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 to 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..*
Operation invocation	Invocation of an operation in an activity flow. The operation must be defined in a context of the domain of the activity.
Role structure	Static structure belonging to a 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).
Domain operation type	The possible types of Activity operations in an Activity flow. **attribute:** Instantiate object The operation creates an object of the given type and sets its attributes to a default value, and sets the initial relations for the object. **attribute:** Link Create a link between two or more objects given the Class relation. Also life dependencies can be linked (with compliance to the rules for life dependencies). **attribute:** Unlink Delete an existing link between two or more objects. Also life dependencies can be deleted (with compliance to the rules for life dependencies). **attribute:** Set attribute Give one or more attributes a value.

5.6 Feature concepts

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

5.6.1 Feature definition diagram

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.

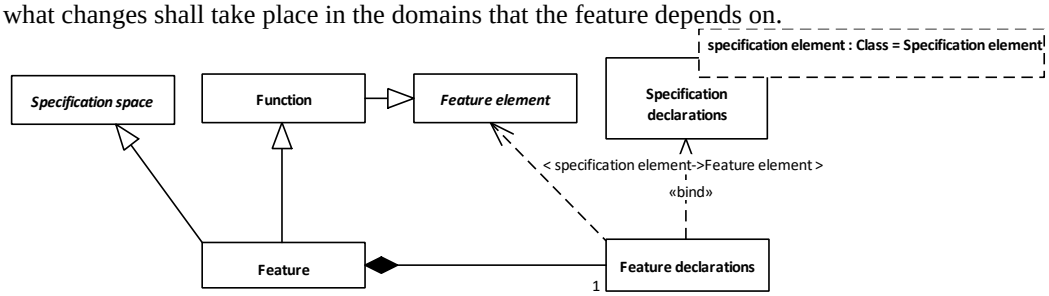


Figure 32: Feature definition

5.6.2 Function definition diagram

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.

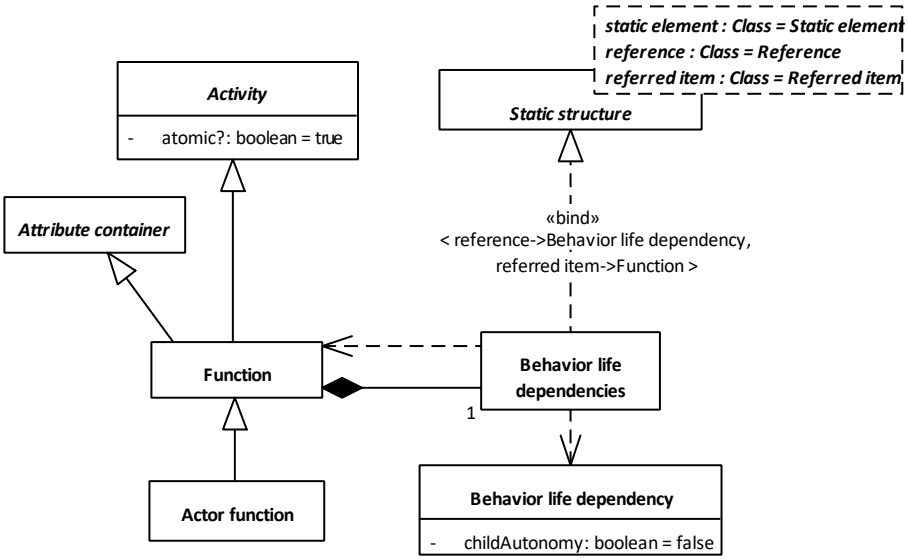


Figure 33: Function definition

5.6.3 Function flow diagram

A Function has a function flow, which expresses what shall happen when the function is instantiated. The flow steps refer to Activity references, which themselves 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.

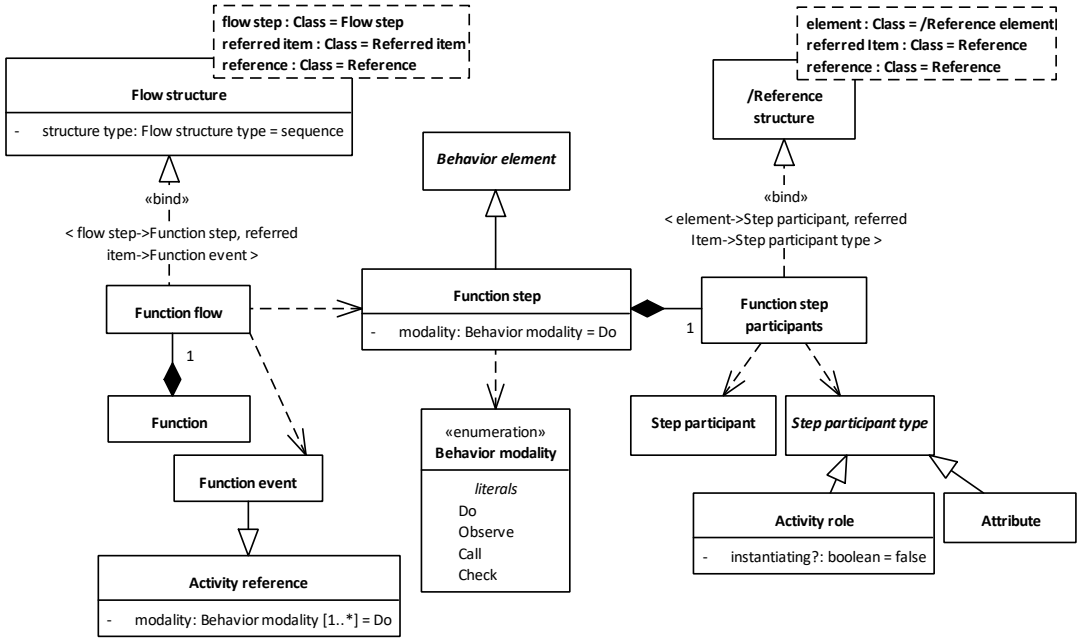


Figure 34: Function flow

Name	Definition
Actor function	A function that can be assigned to an actor and that can be invoked, canceled (devoked?), suspended, and resumed. *TBD: 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 executable. So those are the functions in which life, the life of the dependent function is contained. A dependency also means that the function can access the attributes of the referred function(s).
Behavior life dependency	A Behavior life dependency states within which referred Function the dependent Function must be executed. This means that the contained function can access the attributes of the containing Function. When the child is not autonomous (childAutonomy=false), the termination of the containing (referred) function execution also terminates the contained (referring) function. When the contained Function is autonomous (childAutonomy=true), then the containing function may not terminate its execution when the contained function is not finished yet. **attribute:** childAutonomy
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.
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 and an Activity. A function flow expresses what shall happen if the function is activated, i.e., instantiated.
Function event	An Activity reference within a Function. Each Function event must be referred to by at least one Function Step. *TBD: investigate if this class can be skipped, so that only Activity reference is needed.*
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. The Function steps may only refer to Function events of the same Function.
Function step	Step in the Function flow of a Function. A Function step is a Reference to an Activity. **attribute:** 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 Activity 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.

6 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 analysis, and Text-to-model transformation of the MuDForM method flow. It also contains concepts to record analysis decisions.

6.1 Knowledge containers (static view) diagram

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

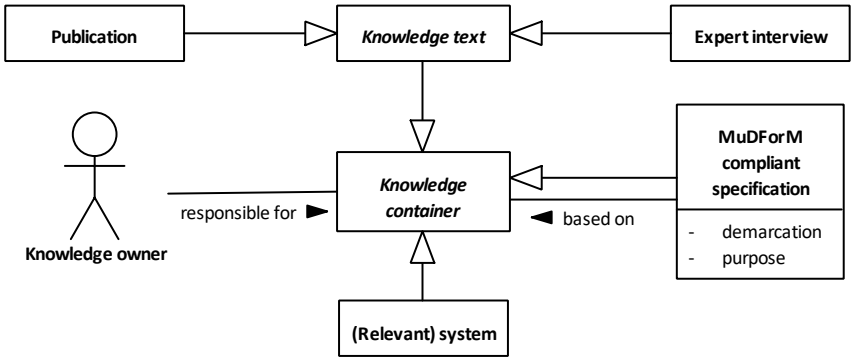


Figure 35: Knowledge containers (static view)

6.2 From text-to-model (interaction view) diagram

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 container. Phrases are Extracted from other Phrases, which can be Source sentences. Phrases can be Extracted or Rewritten according to one of the Phrase types. An alternative approach is not to have a source 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.
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 container. Later, due to new insights, a Phrase element can be (re)Typed with another Term. Terms can also be Renamed, often because of a detected homonym or synonym. 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, i.e., in case of a detected homonym. During analysis, it is also possible to add new Analysis items, which can be a Term or Phrase. One Analysis Item 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).
TBD: Maybe we need to introduce model candidate as the reification of a term with a model term type. Then the candidate can be located in max one MuDForM specification. Now "to Allocate" needs one of classified model term types.

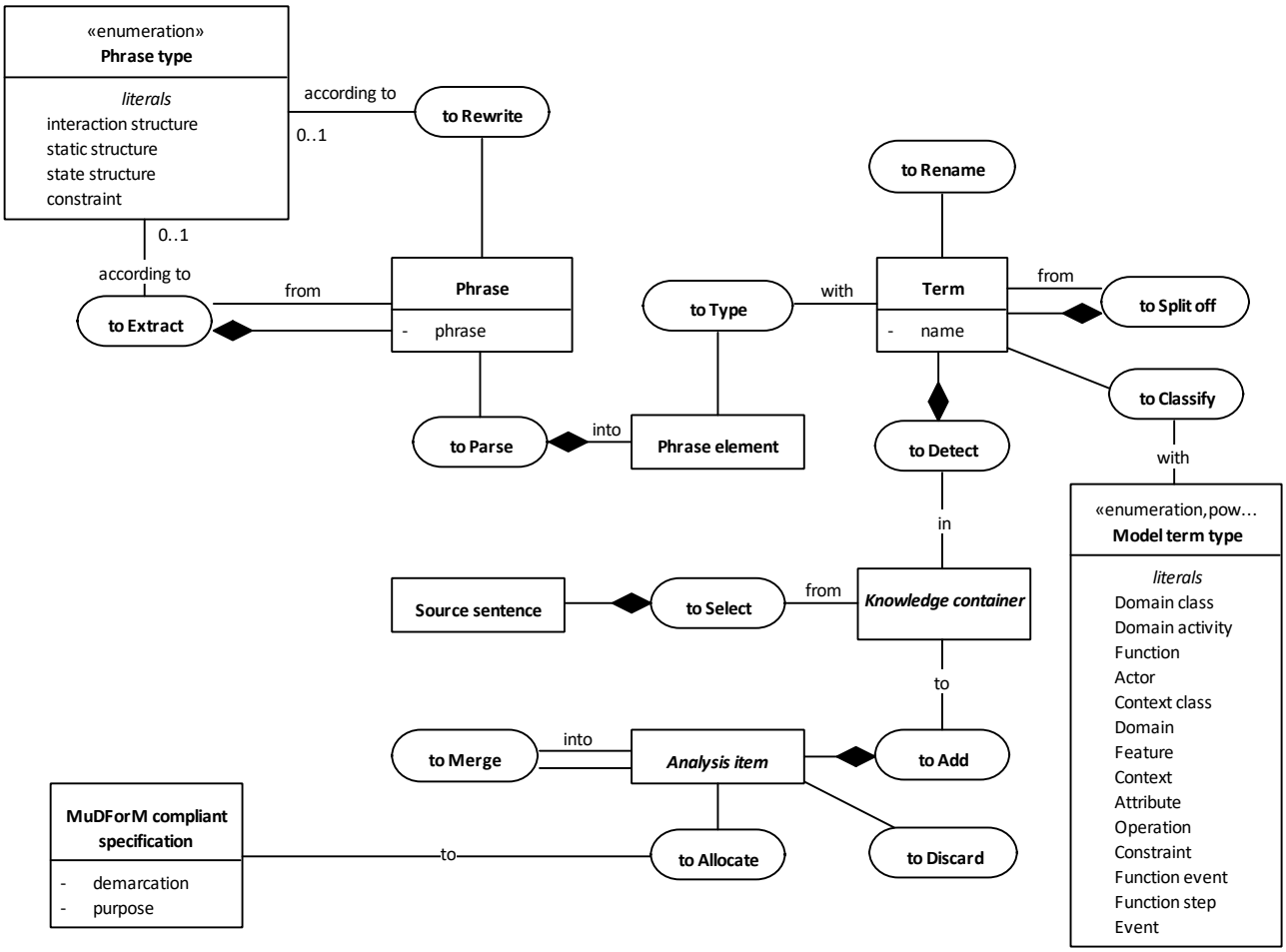


Figure 36: From text-to-model (interaction view)

6.3 From text to model (static view) diagram

Source sentences are selected from a Knowledge container. After that, phrases are extracted from the Source sentences, and contained in the same Knowledge container as the Source sentence. Phrases have a Phrase type and contain at least two Phrase elements. A Phrase element represents the occurrence of a Term in a Phrase. Terms that are a candidate for the model are classified with a Model term type. At last, Analysis Items, i.e., Phrases and Terms, are allocated to a MuDForM compliant specification, typically one of the identified domains, contexts, or features.

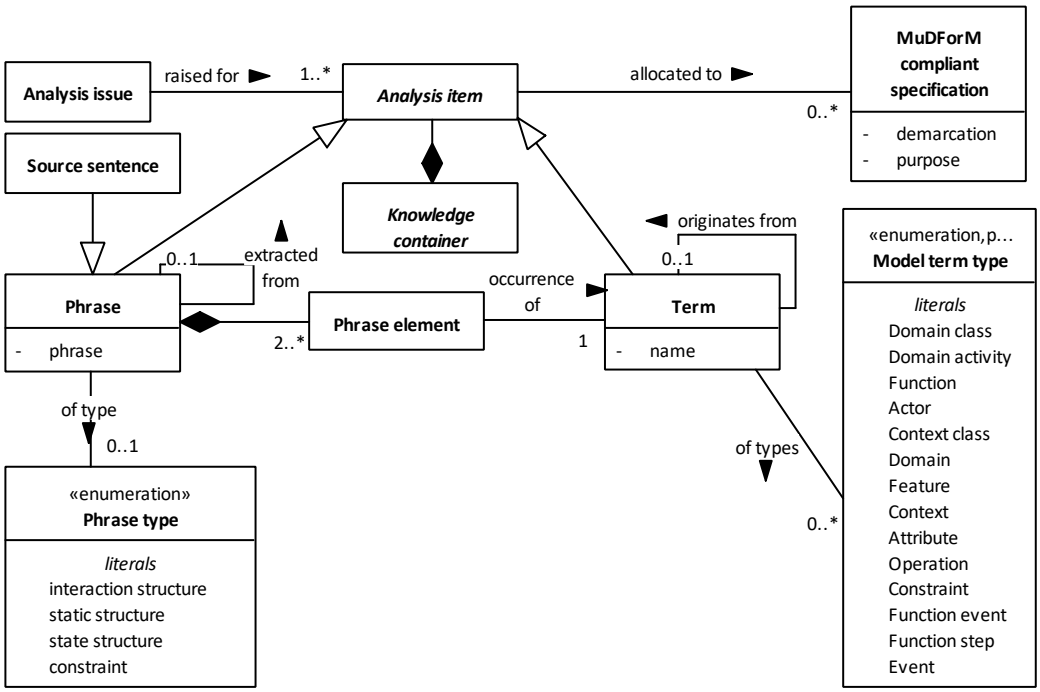


Figure 37: From text to model (static view)

6.4 Analysis issues (interaction view) diagram

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.
TBD: There is nothing specifically analysis-like in this diagram. it can be generalized to modeling or any other method-based action.

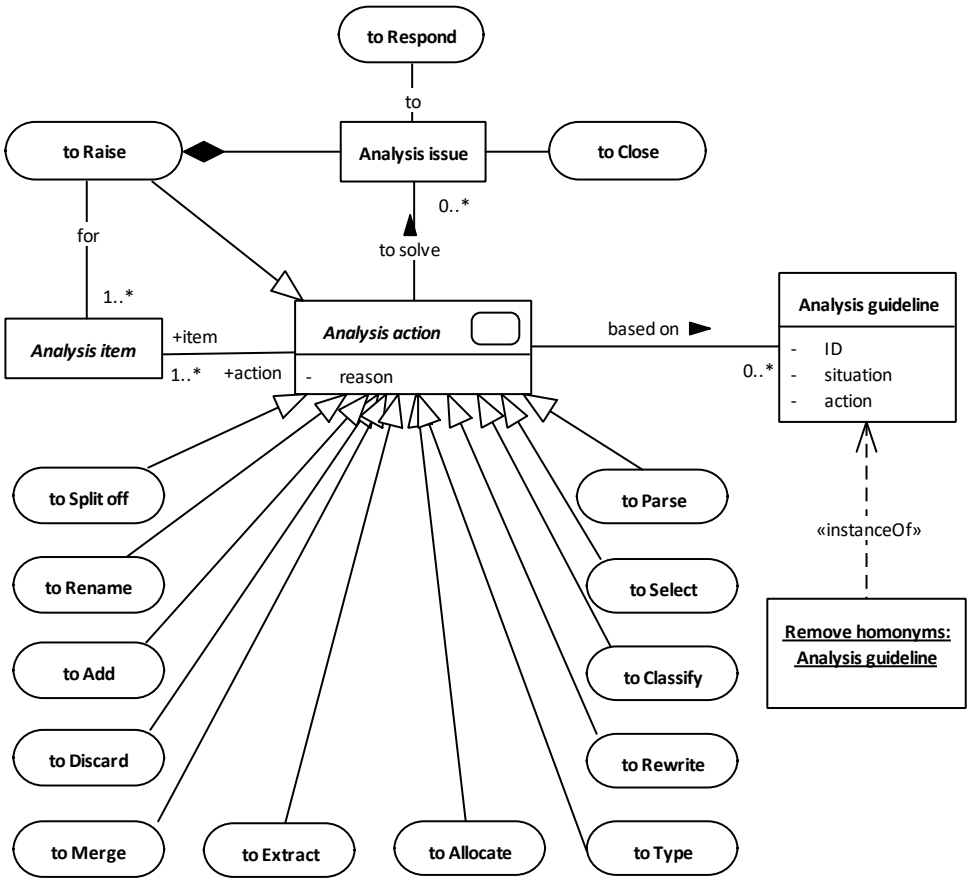


Figure 38: Analysis issues (interaction view)

Name	Definition
Knowledge owner	Person or organizational unit that is responsible for the contents of one or more knowledge containers.
(Relevant) system	(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.
Analysis guideline	Guideline for analysis actions. **attribute:** ID **attribute:** situation A description of the situation in which the guideline is applicable. **attribute:** action A description of the analysis (or design) action that should be done on the involved items.
Analysis issue	Issue raised for one or more Analysis 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 or a Model term type.

Expert interview	Documentation of the interview with an expert. Typically, the interviewed expert is the Knowledge owner of the interview.
Knowledge container	A source with relevant knowledge. A knowledge container can have one or more actors (organization or person) that are responsible for the knowledge in the container.
Knowledge text	A piece of text with relevant knowledge, which can be used for grammatical analysis.
MuDForM compliant specification	A complete MuDForM model or a specific specification space of such a model. 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 container. **attribute:** demarcation 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 specification should be usable for. **attribute:** purpose What is the purpose of the specification? What is it used for? In other words, what do you want to do with the specification?
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
Phrase element	The occurrence of a Term in a Phrase.
Publication	A (part of) a document, or document set, that contains relevant 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. The latter can be the case when the document origins from outside the involved organization.
Source 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.
Term	A single word or compound word that is used in the analyzed text, i.e., the knowledge container that is the source. This can be a noun (phrase), verb (phrase), 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 Classified with Model term types, which makes the term a candidate for the targeted model.

	attribute: name
Remove homonyms	When one Term has different meanings in different phrases, a new Term should be Split off and get a different name. (This is just an example of an Analysis guideline.)
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 is put. Only Terms that are Classified with at least one Model term type can be Allocated. Only Phrases that are formatted according to a Phrase type can be Allocated.
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, or not relevant anymore due to some other reason.
to Detect	To find a new Term in the analyzed Knowledge container.
to Discard	To state that an Analysis item is not of interest anymore.
to Extract	The create a new Phrase based on an existing Phrase that has potentially several other Phrases in it. The Extracted Phrase is part of the same Knowledge container 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, of which each is an occurrence of a Term.
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 Rewrite	To rewrite a phrase such that it matches one of the Phrase types.
to Select	To indicate that a Source sentence from a Knowledge container 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.
to Type	To replace the existing term of a phrase element with another term.
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 A justification for the action.
Model term type	Enumeration of the modeling concepts that are used in a MuDForM specification, and that are commonly identified during grammatical analysis. (Potentially, all modeling concepts could occur during grammatical analysis. But Model term type captures the most common ones). 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 contain the following concepts: • **Functions** are Activities. They specify what must happen when the function is active. • A function can use other activities, which can be other functions, domain activities, and operations. The usage of an activity in a functional 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.

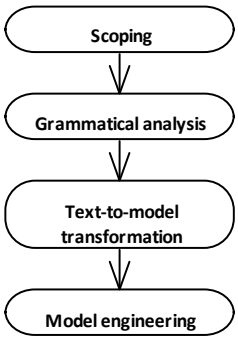
	It has the feature as the root and is called the feature structure. • Function lifecycles describe the control flow of the function's behavior in a process algebra style, i.e., in terms of sequence, selection, parallelism, and iterations of **Function steps**, which are typed by a function event. **Context** 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. Context models contain the following concepts: • **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 **attribute:** Domain activity **attribute:** Function **attribute:** Actor **attribute:** Context class **attribute:** Domain **attribute:** Feature **attribute:** Context **attribute:** Attribute **attribute:** Operation **attribute:** Constraint **attribute:** Function event **attribute:** Function step **attribute:** Event
Phrase type	Enumeration of predefined formats for Phrases which can be transformed into a piece of model. **attribute:** interaction structure 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 classes and activities, i.e., domain activities, operations, functions. 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

	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 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 Phrase that expresses some condition, typically written with operators of proposition or predicate logic, like a "if A then B", or a "for all A: B". Also temporal constraints are possible like "B after- A" or "B before X seconds after A".

7 Method flow

This package describes the steps and guidelines of MuDForM. It builds upon the concepts defined in the Modeling concepts package, the Discovery package and the Viewpoints 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 ordering of the guidelines of a step does not have an explicit meaning.)
Generic guideline: Follow the method flow
An experienced modeler may be inclined to skip steps or to follow guidelines of a step further in the process, e.g., 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 process as a 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.

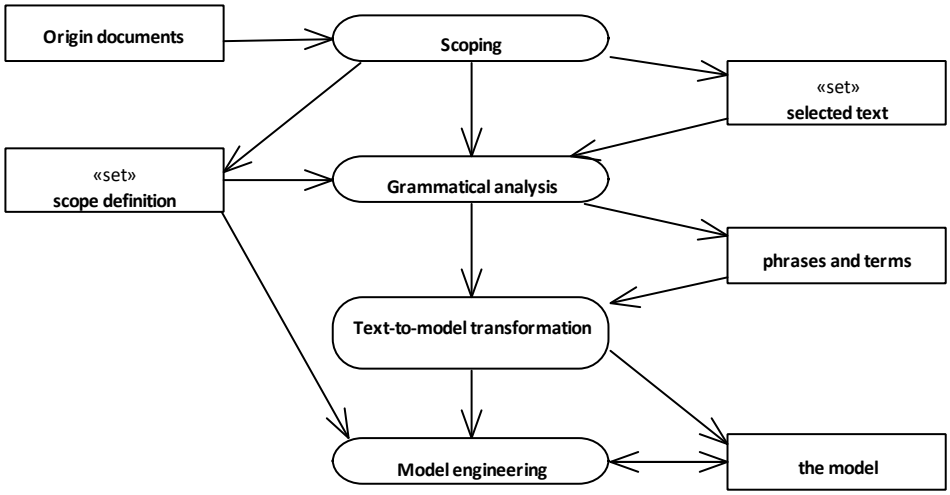
7.1 Main flow diagram



Main flow

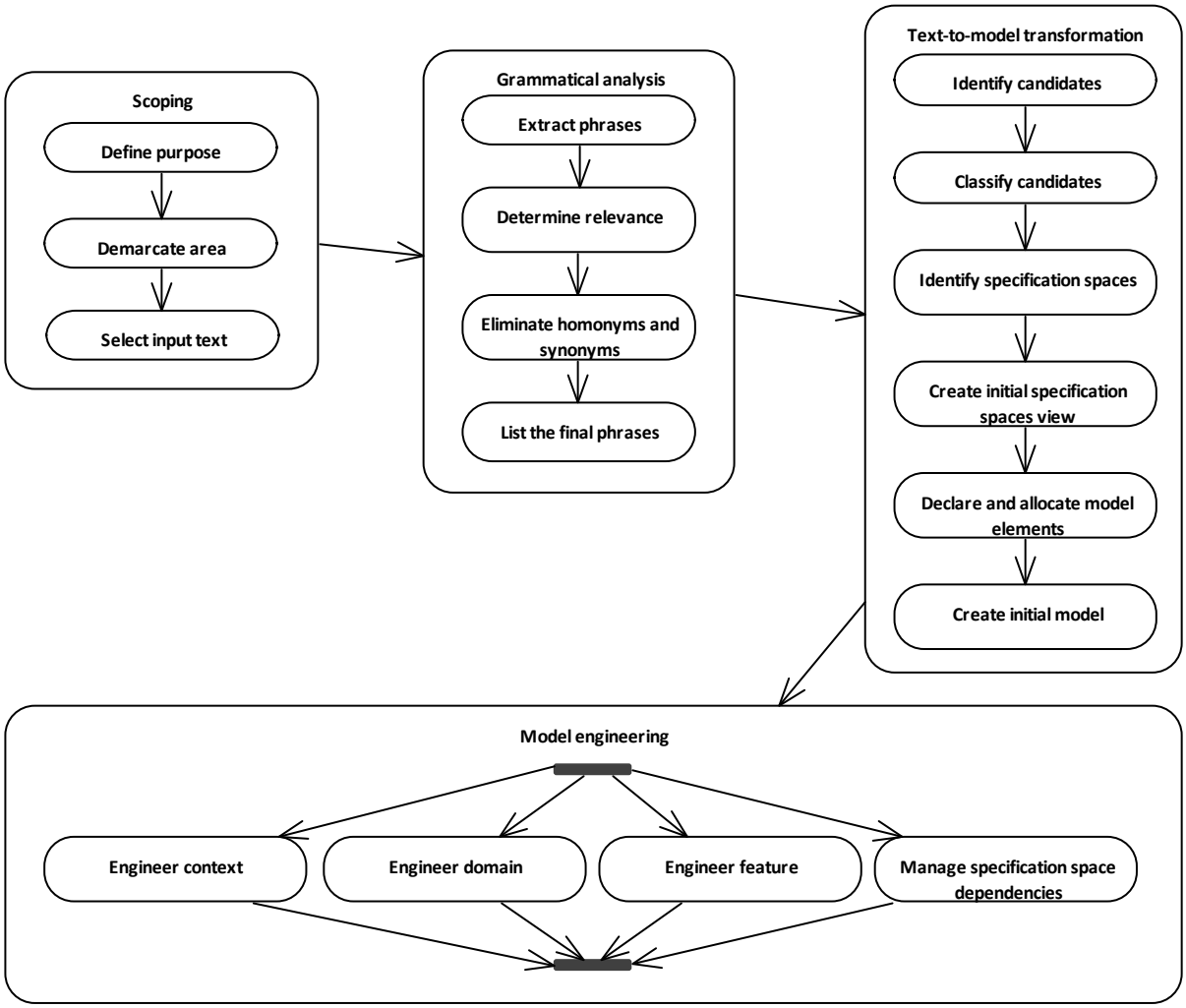
7.2 Main flow with objects diagram

MuDForM consists of four main activities that typically happen sequentially, but also may overlap in practice. It starts with setting the scope where the input text is selected and people are involved. Then, knowledge from these people and selected documents 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.



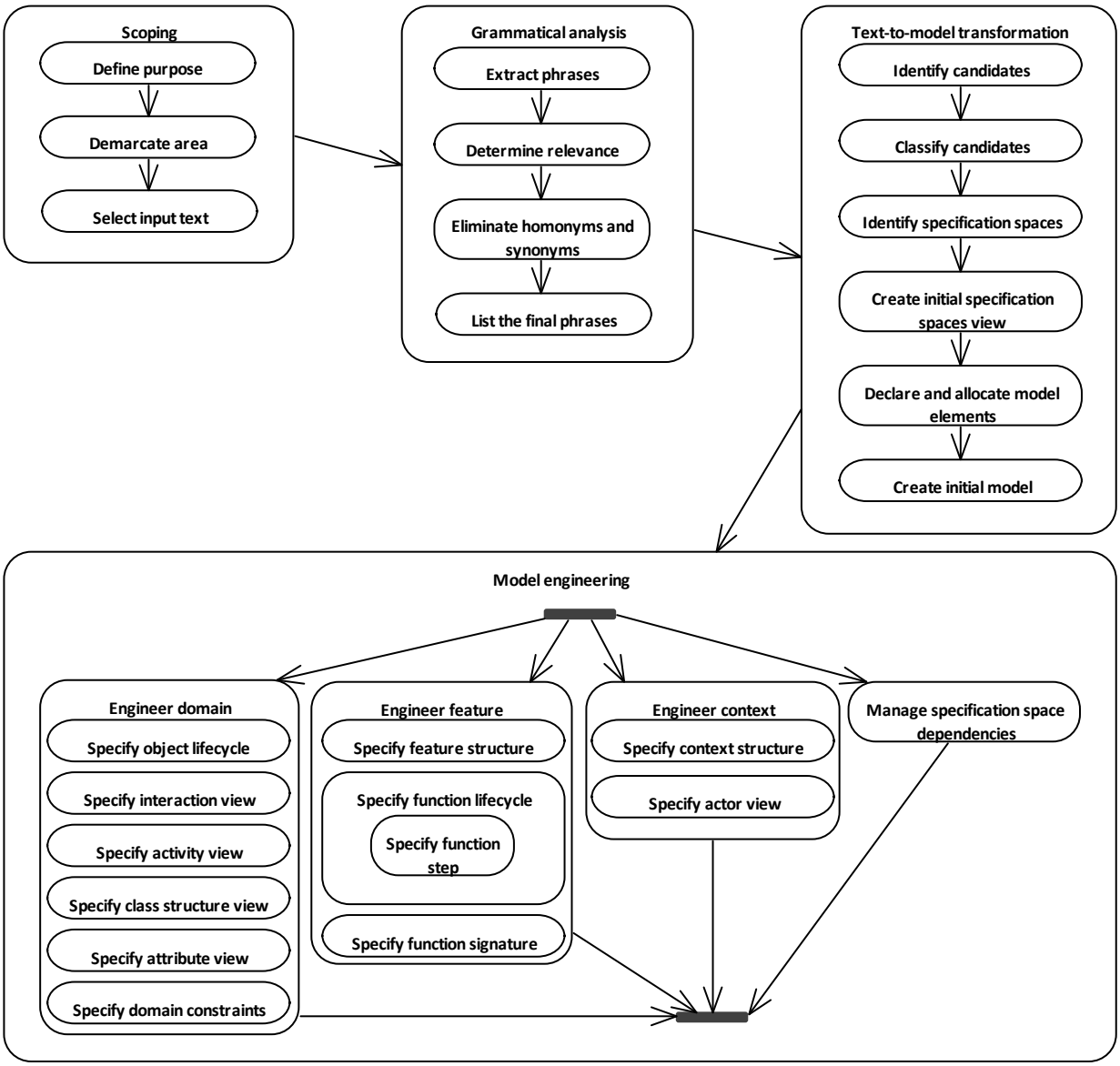
Main flow with objects

7.3 Main flow and sub flows diagram



Main flow and sub flows

7.4 All method steps diagram



All method steps

7.5 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 sources.
When there is no existing input text, the identification of possible model elements can be covered by letting (domain) experts coming up with sentences about the targeted scope.
The goal of scoping is 1) to prevent unnecessary modeling work, and too little modeling work, 2) to have relevant input, and 3) to keep the model and the modeling process manageable.

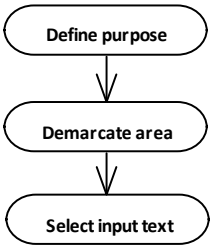


Figure: Scoping

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

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, features, or contexts), which can be the same person. 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.

7.5.1 Define purpose

Specify who the users/customers of the targeted model (or parts thereof) are, and what they want to do with it. 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: Modeling should replace or complement other activities
It does not make sense to start a modeling activity when a project already made designs in another way, or when there is already source code developed before the specification is made that the source code should comply with.
Modeling should start whenever relevant (domain) knowledge is entering the project, either through some document or through some expert. When domain models already exist and specifications, e.g., requirements, must be made, then the domain models should form the terminology for those specifications.

In other words, modeling is not something you do in parallel with things you already do or did.

Guideline: Different specification spaces have a different purpose
Write a purpose specification for each (expected) specification space. A domain model has typically a wider applicability than a feature model, which implies that it has a different purpose.

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.

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 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: 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.

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 model is for that activity.

7.5.2 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 for features. Otherwise, you can just name nouns, verbs, or phrases. It is also possible to describe a rule that demarcates 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. So, make the demarcation a regular discussion point during the rest of the modeling process. Furthermore, do not spend much time on it in the beginning of the modeling process. 15 minutes is a maximum.

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 for determining to 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: 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 with delivering goods.

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, to come up with an initial list of concepts.

7.5.3 Select input text

State which pieces of text from a knowledge source are the starting point. The knowledge source is often an existing document, or a document that is created from 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.
For each piece of input text, a domain expert is appointed to provide missing information and assist with inconsistencies during analysis and modeling. When there is no existing input text, the identification of possible model elements can be covered by letting (domain) experts coming up with sentences about the targeted scope.

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 analyzing 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 terms listed in 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 add more sentences first.

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 can happen (what is possible) vs. asking what must happen (what is desirable). Only apply this guideline if you are an experienced modeler. If not, then just stick to plain natural language.

Guideline: Ask for examples if generic knowledge is hard to phrase
When extracting knowledge from involved (domain) experts, ask questions about the targeted domain or system at a general level. So, questions about things that are throughout the whole system or domain. When these are hard to answer, ask questions about specific events that happen in their domain.

The first type of questions will have answers that are already on domain level or feature level. The second type of questions leads to answers on instance level, which can be generalized to domain level or feature level to get usable input for the model.

7.6 Grammatical analysis

The input text is analyzed and transformed into a set of phrases with terms that are candidate elements for the model. The goal of this step is to maximize the knowledge elicitation from the source, and to make the resulting model traceable to the input.
During grammatical analysis, all decisions regarding the phrases or terms must to be made with the approval of the involved (domain) experts.

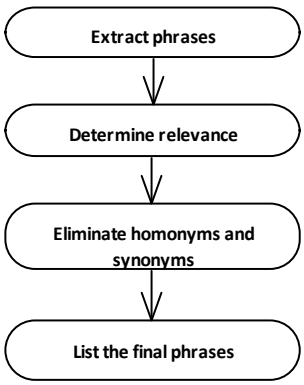


Figure: Grammatical analysis

Guideline: Combine steps for efficiency
Experienced modelers may combine or skip method steps (of grammatical analysis) for efficiency of the process. For example, do not extract irrelevant phrases, or replace already identified synonyms, which you identified before, when

you extract a phrase. This prevents unnecessary work. Make sure such replacements are done in agreement with the involved experts.
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 when 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 this is not possible, then either the constraint phrase is unclear or incorrect, or the domain model must be adapted.

7.6.1 Extract phrases

Extract phrases from the selected input text and format them according to one of the phrase types: interaction structure phrase, static structure phrase, state structure phrase, and conditional phrase. Typically, each sentence from the input text leads to one or more extracted phrases. As such, the extracted phrases form a decomposition of the original sentence. The next method steps might lead to changes in terminology or structure of the phrases. Based on those changes, it is always possible to rewrite an original sentence in the new terminology or structure, to check if the model still expresses the initially intended meaning. When there is no input text selected, sentences can be elicited directly from domain experts. This should be a very mechanical activity (which potentially can be automated). The meaning of the phrases or validity of the phrases is not important in this step. 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 which phrase type to choose for a part of a sentence, then just choose one and continue. It is easier to understand the relation between terms in a phrase when having multiple base sentences.

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, "They uses the key to open the door." can be replaced by "They opens the door with the key.", under the assumption that the text is about opening doors and not about different usages of keys. 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: Keep subjects of interaction phrases if it is 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".

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 indicates 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".

7.6.2 Determine relevance

Determine the relevance of each extracted phrase from the perspective of the defined scope. Discard phrases that do not fit the scope definition, or that are (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 modeling 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 about 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 throw an item into the trash bin".

7.6.3 Eliminate homonyms and synonyms

This step is all about reducing ambiguity. Check all phrases for homonyms and synonyms and eliminate them in consultation with the involved (domain) experts to assure that all terms (words) have exactly one meaning, and that all relevant meanings are covered by exactly one term.
Replace the terms that are a homonym or synonym in phrases, with the chosen term. The chosen terms may also come from already existing models.

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: 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: 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 verbs will lead to different steps in the object lifecycle. If needed, then making the activity views for 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: 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: 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: 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: 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".

7.6.4 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 identified homonyms and synonyms in the phrases with the chosen terms.
- Combine phrases with the same verb but with different nouns into one phrase.

7.7 Text-to-model transformation

This is the transition from working with text to working with models. The specification spaces, which form the top-level structure of a model, are identified, and the phrases are transformed into pieces of model.

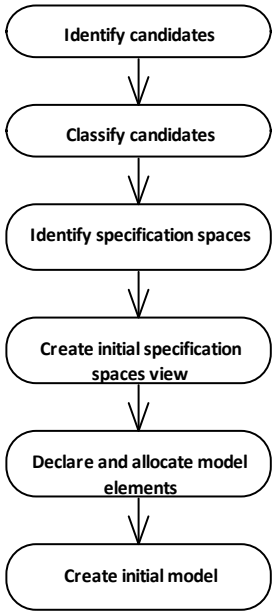


Figure: Transform text to model

7.7.1 Identify candidates

Determine the terms in the phrases, i.e., nouns, verbs, adjectives, and adverbs, that are a potential element of the model. Some phrases might (as a whole) be a candidate themselves, in the case of a condition phrase.

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 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.

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

7.7.2 Classify candidates

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

Guideline: Classify changes by default as activities
If it is not clear if a verb must be classified as a domain 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. This decision flow can be used:
Ask for each verb in an interaction phrase if it expresses an atomic change to one or more domain objects?

- If yes, then it is an activity or an operation. Then ask if it is controlled separately, i.e., invoked from the control system. If yes, then it is a domain activity. If not, then it is an operation, or a (sub)function.
- If not, then it is a function, or a domain activity with a separate start and end activity, i.e., it is a domain change, but not atomic. In this case it should be split in two domain activities, if start and end are controlled separately.

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: Identify functions from use case interactions
If use cases, user stories, system (interaction) scenarios, are used as input text, then the steps that describe system behavior are often calls of system functions.

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 within that activity.

Guideline: Focus on the core concepts
Focus on 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: 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: 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, 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 and indexicals indicate an (activity) attribute
A definite article, i.e., "the", might point to a role an object plays in a function or in a domain activity. Classify it as a activity attribute with a (domain) class as a type.
The same holds for so called indexicals. The standard list of indexicals includes pronouns such as “I”, “you”, “he”, “she”, “it”, “this”, “that”. They indicate that there is an object that participates in several steps in a lifecycle. Such an object must probably be represented by an activity attribute. In the case of an domain class, the indexical can also refer to the domain class itself, which is the container of the object lifecycle.

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.

7.7.3 Identify specification spaces

Identify the specification spaces and classify each of them as context, domains, or features. Choose spaces for specification elements that are coherent. Of course, the existing specification spaces must be taken into account. Each specification space should have an owner who is responsible for its content.

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. Locations 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.

7.7.4 Create initial specification spaces view

Create a view (e.g., a diagram) with all the specification spaces. Create relations (dependencies or sub declarations) between spaces if they are expected or already known, in compliance with the specified rules under the activity Manage specification space dependencies.
For example, the taxi driving domain depends on the navigation domain, and the context of Payments.

7.7.5 Declare and allocate model elements

Place each specification element in the most logical specification space. An element can be reallocated during model engineering.

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.

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.

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.

7.7.6 Create initial model

Create a first version of the views in the specification spaces from the list of final phrases, based on the candidates' types and the positioning of the candidates in the specification spaces. All interaction phrases become a relation between an Activity (Domain activity, Operation, Function) and one or more classes, 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.
In the case that there is no input text, and hence no grammatical analysis, this is the starting point of the modeling process.

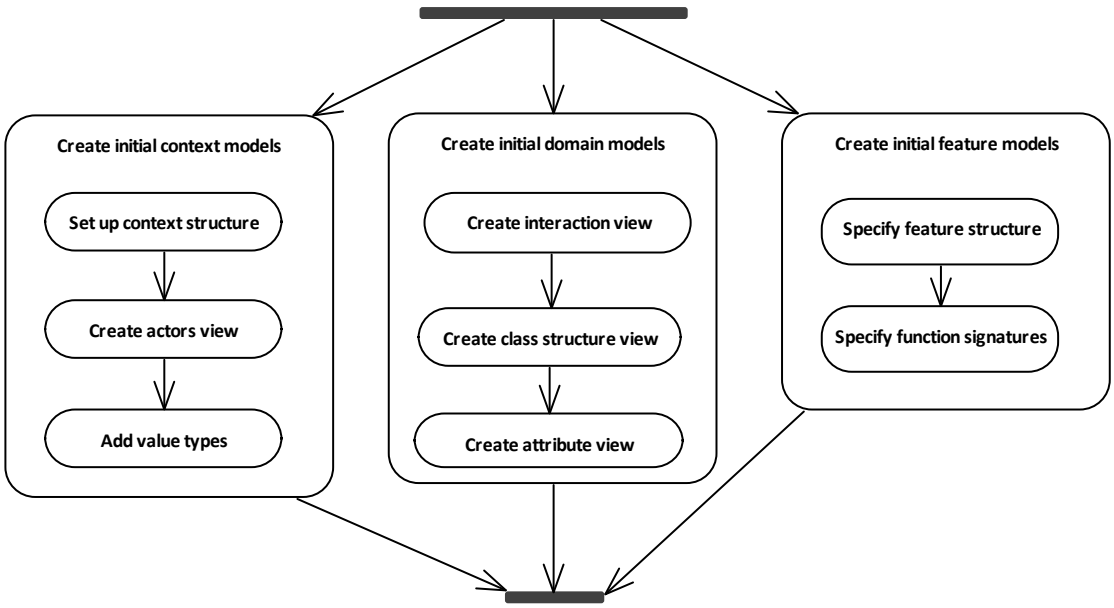


Figure: Create initial models

7.7.6.1 Create initial context models

Make the initial views per identified context.

Set up context structure

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

Create actors view

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

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.

7.7.6.2 Create initial domain models

Make the initial views per identified domain.

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: Use the default relation names
Let the prepositions in the phrase be the relation names between the corresponding domain activities and domain classes. When a domain class represents the direct object of an interaction phrase, leave the name empty or use "d.o.". When a domain class represents the grammatical subject of a phrase, use "subj" as the relation name.

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".

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. When later, during model engineering, this static view might become too big, it can be split into a separate attribute view and structure view.

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.

7.7.6.3 Create initial feature models

Make the initial views per identified feature.

Specify feature structure

For each function, define a function event for all the activities that can happen during an execution of that function. These include not only the activities that the function executes, but also the activities than are called from the function or that the function should react to.

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.

7.8 Model engineering

The model is completed. Inconsistencies are solved by following the method steps and guidelines, and iterating over the different views. Domain experts are involved to answer questions, decide about missing concepts and model conflicts. Model engineering consists of a step to **manage the dependencies between the specification spaces**, and three steps for **engineering** the different types of specification spaces, *i.e.*, **contexts, domains, and features** There is no restriction to the order of the main steps. But typically, the focus is on engineering a domain or feature, and capture in a context the

needed concepts that are not in the domain or feature of interest. When the focus is on engineering a feature then existing domain models and feature models might be used to specify the behavior of the functions in the feature, which also validates the used models. Often during feature engineering, new domain model elements and context model elements are identified. They have to be added to the domain or context models, to assure that all the concepts used in the feature model are properly defined.
The main principles that guide the model engineering phase are 1) keep the views consistent, and 2) acquire information from experts or documents to achieve a consistent and complete specification.

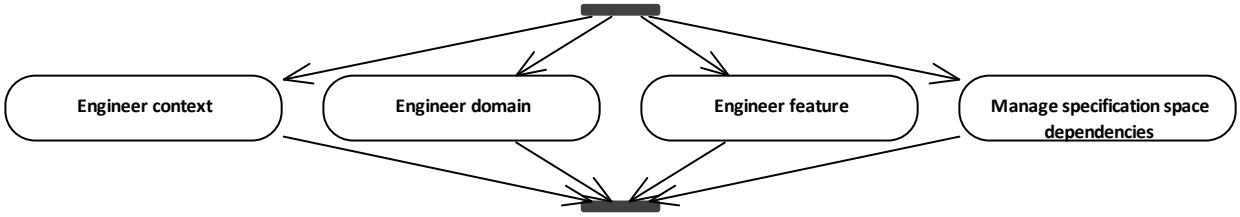


Figure: Main steps in model engineering

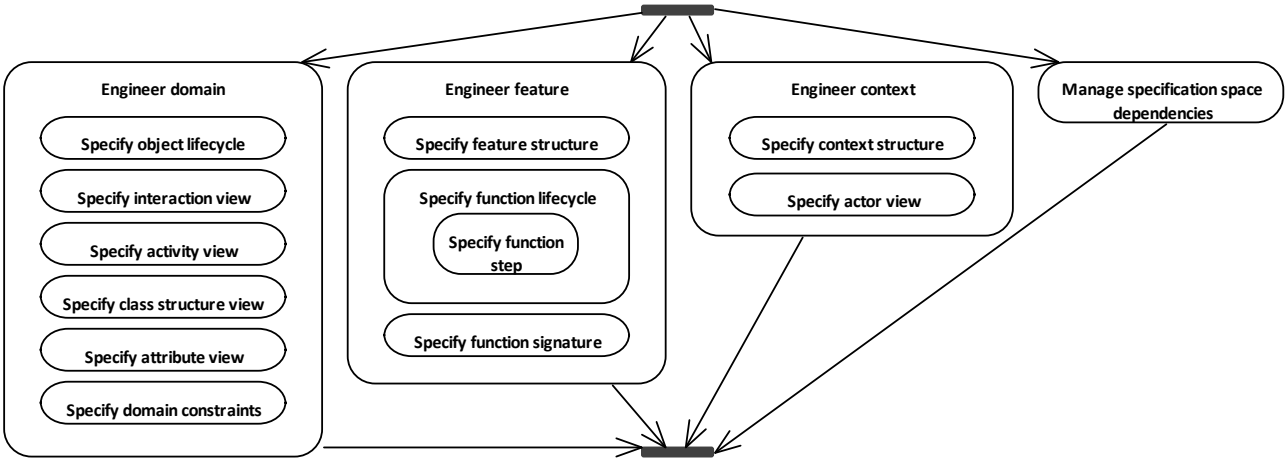


Figure: Model engineering steps

Guideline: Involved experts choose names
When making models, you sometimes need to come up with a name for a model element that has no existing term for it. This can be the case, because normalization forces you to introduce a model element which experts do not perceive as a distinct autonomous object (yet). In such a case, a name must be invented. Such a name should be supported, preferably chosen, by the involved (domain) experts.
Sometimes a collection of objects that is used throughout a feature must get a name, because they meet some constraint as in the example "people that you need to notify when you are going into the hospital for three weeks.". The involved experts should support the chosen name of, in this case, the function attribute.

Guideline: Start with the domains
Although the sub activities of model engineering can be done in parallel, it is probably efficient to start with the 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: Ensure consistency between classifiers and known instances
If you know an instance of a Classifier, e.g., Domain class or Domain activity, then it must have the specified properties for that Classifier. If it does not, then either the Classifier's definition is incorrect, the classification is incorrect, or your knowledge about the instance is incorrect (or incomplete). Correct the model or the instance such that the consistency between them is ensured.

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.

Guideline: Use nouns for class names
In case a class name has to be invented, i.e., when the involved experts cannot provide a good candidate, use a noun. When one noun does not suffice, then prefix the main noun with other nouns, or with adjectives.

Guideline: Use verbs for activity names
In case an activity name has to be invented, i.e., when the involved experts cannot provide a good candidate, use a verb that expresses activity. When one verb does not suffice, then postfix the verb with adjectives or the main object in the activity.

Guideline: Only use constraints when other modeling concepts do not suffice
If something can de defined through model elements that are not constraints, then that should be done. Only use a constrain for other cases. For example, do not use a constraint to specify the order of steps in a lifecycle, when the relations in a lifecycle can be used to specify the order. Let's say a lifecycle has to two types of steps: A and B, and it holds that an A should always be followed by a B, then the lifecycle should have a sequence of A followed by B, instead of a parallel flow structure A and B and a precondition on B that it should be preceded by an A.

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 and activity, and their subclasses.) So:

- A Domain activity may have a generalization to a Domain activity in the same domain.
- A Domain class may have a generalization to a Domain class in the same domain.
- A Class may have a generalization to a Class in the same context.
- An Operation may have a generalization to an Operation in the same context.
- A Function may have a generalization to a Function in the same feature.

If they are not declared in the same specification space, then:

- A Domain class may have a generalization to a Class in a context.
- A Domain activity may have a generalization to an Operation in a context.
- A Function may have a generalization to a Domain activity.

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 use a coordinated structure, with a Coordinated structure type (All, Some, One of) over zero or one elements.

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 has a generalization to Q, then the referring item 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 or generalization. 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.

7.8.1 Engineer context

Specify the operations, classes, and actors, and cross check their use in domains and features. Use the formalisms for value types and specifications of operations.

7.8.1.1 Specify context structure

Specify the operations and classes, and their attributes, and the class relations, the class relation roles, and the role connections. Specify the generalizations between classes, between class relations, and between operations. Express the value types and formulas of operations in terms of the used formalisms.

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 in such case 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.

7.8.1.2 Specify actor view

Specify the actors, their attributes, and their generalizations. Specify in the behavior composition structure which activities the actor can execute, call, and react to.

7.8.2 Engineer domain

Make a complete domain model by iterating over the sub steps, 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 system'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 causes.
- 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. On the other hand, 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. Only freeze and release the stable part, and keep the immature part open for change.

Guideline: Analyze 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 design and 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 some of 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 is involved in different domain activities it then another class, .i.e., you do something different with it.
- It has "significant" different attributes (without formalizing what significant exactly means).
- It has an interesting life in the considered scope, i.e., its instances (objects) undergo multiple actions 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 domain class and every action is a domain activity.

Guideline: Criterion for generalizations
Only use a generalization between two classes if:

- Two or more classes share the same relation to another class or action.
- Two or 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 the type of an attribute) 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 model 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: 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: 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: 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.

7.8.2.1 Specify interaction view

Define the domain activities, their roles, and which domain classes are involved in each role. Define the generalizations between domain activities.

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

Guideline: Describe a domain class
A domain class description 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: 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 that contains the activity. This ensures than the OLCs of those involved classes have the activity in their scope.

Postcondition: Objects enter a role via an action
An role class, i.e., an abstract class that has generalizations to one or more parent classes, must be associated to at least one domain activity that transforms an object into that role. Such an activity is called a transformation activity of that role class. This can 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: Unique role names
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.

7.8.2.2 Specify object lifecycle

For each domain class, define the order in which an instance of the domain class can participate in instances of the related domain activities, i.e., the involvements of the domain class in activity role are placed in the object lifecycle as a step, and connected to the other steps. As the object lifecycle is a flow structure, it is possible to use sub flows for the coordination of steps, i.e., to select between sequences of steps or to state that several sequences of steps are executed in parallel. While making an object lifecycle, it is possible that new domain activities are discovered, which impacts the next steps.

Guideline: Split up a non-atomic domain activity
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 introduce an explicit begin-activity, and end-activity. Ask if there are specific verbs that indicate the beginning and ending of such an activity. if not, then just use "begin/start" <activity> and "end/stop" <activity> as names.
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 step is called normalization.)

Guideline: An object can leave a role
A role class, i.e., an abstract domain class that has a generalization to at least one concrete domain 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.

Postcondition: An OLC has steps for each involvement
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 generalizations to 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.
TBD: to figure out how to do this efficiently. For example, by putting (the OLC of) the role class as a parallel branch or as one sequential step in the OLC of the concrete classes.

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 generalization class of the domain class.

7.8.2.3 Specify class structure view

Define the domain class relations, the generalizations of each domain class, and the object life dependencies of each domain class.

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.

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 of the child. In the case that role is "direct object", the default life dependency name is "of".

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 domain activity in which those classes are involved or must be part of a composition and the composition has an instantiating activity. 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.

Postcondition: Life dependency starts at instantiation
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. The question to ask is: does the child object mean anything if the parent's life is ended.
Note that not every class that participates in the instantiating activity of a child class must become a parent.

7.8.2.4 Specify attribute view

Define the attributes of domain activities and domain classes.

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.

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 mean a state change to the object.

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: 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.

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.

7.8.2.5 Specify activity view

Create a view for the activity flow of each domain activity, which expresses the order of the operations that make up the activity. Operations can be a local operation defined in a formula, invocations of operations defined in a context, operations that set object attributes, or operations that instantiate, link, or unlink objects. Possibly specify preconditions for the domain activity. Operations and preconditions may use the activity's attributes and the attributes of involved domain classes. Instead of making an activity flow, it is also possible to specify a postcondition of the domain activity.

Guideline: Ensure the realizability of an activity
Sometimes activity views are skipped. Often this is not a problem for the rest of the specification, because the activity view is a structure of a single activity. But when implementing the activity in software, one might discover that it cannot be implemented with the activity attributes, or available object attributes. In this case, making the activity view will clarify 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 (=parameter) 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, which can be used outside the activity. In other words, a domain activity can only cause changes to involved objects. There are no side-effects.

Postcondition: An activity view may only refer to related objects
An activity view may only change attributes of involved objects, and it may only refer to attributes or inspect attributes of the activity 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. Of course, it cannot change the past state of an object.

7.8.2.6 Specify domain constraints

Add invariants to the domain, domain classes, domain class relations, and attributes. This step is typically done at the end of domain engineering, and is needed when there are domain properties that cannot be expressed via the other modeling concepts.

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.

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.

7.8.3 Engineer feature

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

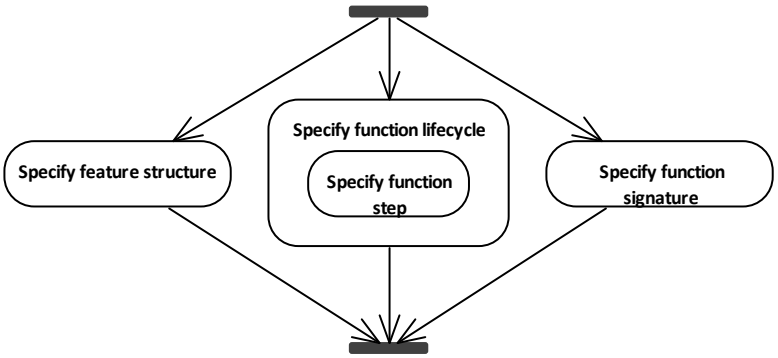


Figure: Engineer feature

Guideline: Define a function for coherent behavior that will be assigned to an actor
Define functions for units of behavior, often called tasks, that will be assigned to and performed by one actor.

Guideline: Let function names reflect the duration of the function
If a function has a clear start and stop moment, i.e., it is invoked and ends at some point, then the function name should start with a verb that expresses the change of the function, possibly the name of the main domain activity in the functions, followed by terms that express the main object in the function and the operational context of the function. For example: Order books online.
If a function is continuously active within a feature and manages some state, then the function name should include a term that reflects the state that the function manages, or the main object of the function, and a term that express the activity that function performs on that state or object. For example: Manage shopping cart of user, or Guard maximum amount of shopping cart.

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: An atomic object manipulation might indicate a domain activity
During feature specification, one may find functions that manipulate a single object. Take into consideration if such a function should be defined as a domain activity. If so, it should be allocated to the domain model. If the behavior is about what can happen and not about what must happen or how does it happen, and if the behavior is independent from the feature, and any actor that performs the function, then it might be a domain activity.

Guideline: Start with specifying functions for domain classes that are not 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 those strong objects.

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.

7.8.3.1 Specify feature structure

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

Guideline: Cover the relevant domain 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.

7.8.3.2 Specify function lifecycle

The control flow of each function is specified, i.e., describing the order in which the function steps must be executed. The function steps refer to sub-behaviors, i.e., function events, 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. All the sub behaviors of the functions must occur at least once as a function step.
There are different types of ordering the in the function flow, derived from the flow structure pattern: sequences, selections, concurrency, or iterations.
There are typically two ways to reason about the lifecycle. The first way is to start with the main input attributes of the function and decide which activities should be performed on them going forwards in time. The second way is to start with the end of the function in mind, which can be a postcondition or the execution of an essential domain activity, and then reason backwards about the order of the steps.

A graphical depiction of the lifecycle view typically uses 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 function attributes that are allocated to step participants.

Guideline: Temporal words in the input text indicate temporal order
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: All function events must occur as a step in a function lifecycle
All the function events must occur at least once as the type of a step in the function lifecycle. Otherwise, it would not be a function event. In other words, if a function is dependent on other activities, then those activities should also be used in the behavior specification of the function, i.e., in the function lifecycle.

Figure 39: Specify function step

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

- Function attributes are allocated to the (actual parameters of) the step. The function attributes must belong to the same function as the step, or they belong to a function that contains the function. Typically, a feature, which itself is also a function, has attributes that will be used in many steps of the functions of the feature.
- The preconditions for this step are defined, *i.e.*, the constraints on the step participants. A condition is typically expressed in a logical language and may only use terms that are elements within the scope of the function. (Step preconditions are also called enter conditions.)
- The postconditions for this step, that is, 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 1) The objects allocated to the action have a type that corresponds with a domain class that is involved in the domain activity. 2) Function attributes are allocated to each input attribute of the action and their types match. 3) The function lifecycle does not violate the object lifecycle of an involved object (which can only be fully controlled at execution time and not during modeling).
TBD:

selecting an object should not be a separate function step. It is just a constraint/precondition on the relation between the step participant and the function attribute that parameterizes the participant.
What to do when the step cannot be executed? Main idea: do not pollute the specification with this, but decide something in the transformation to a working system. A kind of default semantics must be defined for these cases. For example, "abort the function". What do you have to check to be able to execute a function step?

- *Does the function lifecycle allow it? In other words, is this a possible next step in the function lifecycle?*
- *Does the previous function step allow it, in case this step wants to disrupt the previous step?*
- *Are the preconditions of the function step met?*
- *Are the preconditions of the type of the step met? (The type can be an operation, domain activity, or function)*
- *Does the object lifecycle of the participating objects allow it.*
- *Is the actor ready?*

Guideline: Default allocation of function attributes to step participants
Often one domain class will only occur just once as the type of an function attribute within 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 suggests that those step participants don't need to be allocated manually, but could get a default allocation automatically.

Guideline: Ensure the model consistency between constraints that involve the same domain class
Find all the constraints that are relevant for a function step for each of the involved (domain) classes. 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: Make sure that used domain activities comply 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, i.e., the same activity roles)?
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.

Postcondition: Function steps are consistent wit ht the 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 lifecycle does not violate the object lifecycle of an involved object (which can only be fully controlled at execution time and not during modeling). So each function step (that is an action) must be performed on objects that can undergo that action according to their object lifecycle.

7.8.3.3 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 an Activity.
All function signatures can be put in a single view, or a separate view 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, they are put in the same view.
The function signatures also specify the preconditions and invariants that hold for the function attributes, that is, things that must be true in order to guarantee the proper outcome of the function. It is possible to specify postconditions, but this is not necessary, because MuDForM follows a whitebox perspective on function specifications. Although, a postcondition could help to guide the design of the function lifecycle.

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.

7.8.4 Manage specification space dependencies

Define the dependencies between the different specification spaces. Define a dependency from space P to space Q (the dependency structure of P has a reference to Q) if P has elements that refer to elements of Q. This step exists to guard the correct use of model elements across specification spaces.

Guideline: Define only used dependencies
Only define a dependency from specification space P to specification space Q, if somewhere in P a reference to an element of Q is set.

Postcondition: Allowed Dependencies between different types of Specification spaces
A specification space (child) may depend on other specification spaces (parent) with the following constraints:

- Features may depend on features, domains, and contexts.
- Domains may depend on domains, and contexts.
- Contexts 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.

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.

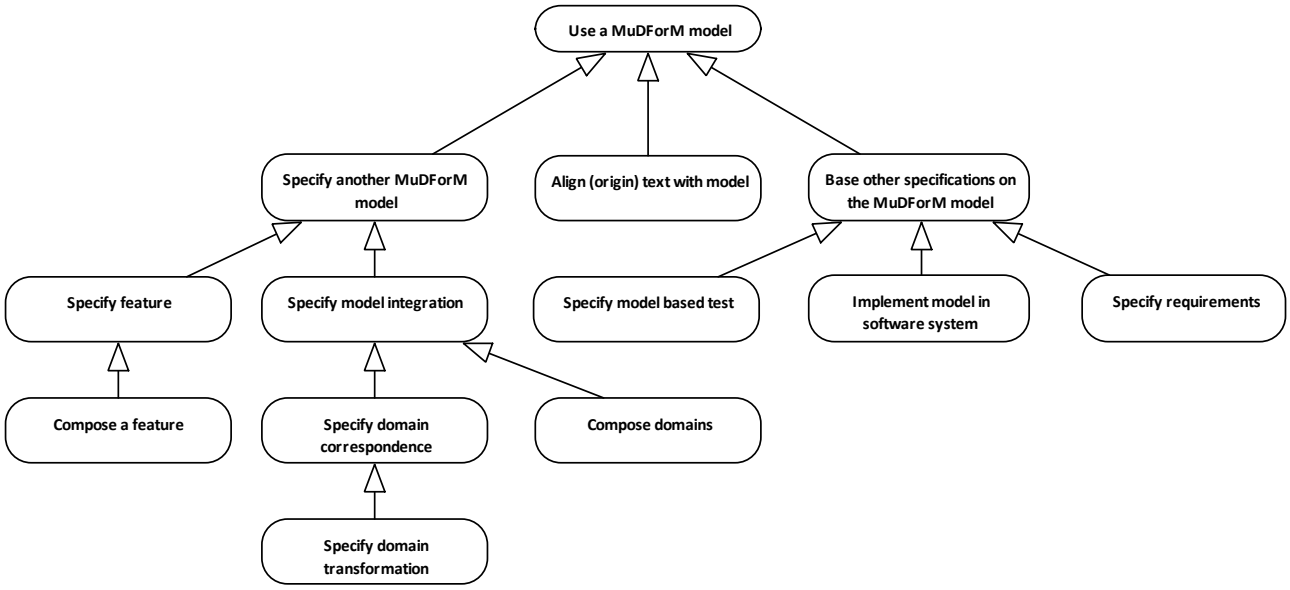
Postcondition: No cyclic dependencies
The graph of all dependencies between specification spaces may not contain cycles. If a cycle appears, it is most likely that a specification space must be split, because it contains parts that differ in abstraction or aggregation level.

7.9 Model usage

This package introduces the different usages of a MuDForM compliant model. We list these activities, but we will not work them out in detail now.

7.9.1 Use MuDForM models diagram

The diagram shows various activities that involve a MuDForM compliant model. It is intended to serve as placeholders for more detailed steps and guidelines to implement those activities.



Use MuDForM models

7.9.2 Use a MuDForM model

7.9.3 Specify another MuDForM model

To use a model in another MuDForM specification. This activity has sub-activities that address a specific type of specification.

7.9.4 Specify feature

To specify what should be true in the context of a specific feature. This includes specifying what shall happen (behavior) and what shall exist (objects). A feature is specified in terms of one or more domains and/or other features, and typically also uses elements from one or more contexts.

7.9.5 Compose a feature

To create a feature out of one or more existing features.

7.9.6 Specify model integration

To create a new MuDForM model out of two or more existing models.

7.9.7 Specify domain correspondence

Specify the relation between two or more domains in the context of a feature. A correspondence specification consists of correspondence rules between the elements from the involved domains.

7.9.8 Specify domain transformation

Specify the transformation from a source domain model to a target domain model. The target domain might also be defined in a MuDForM model. A transformation specification consists of several transformation rules.

7.9.9 Compose domains

To create a new domain model out of one or more existing domain models.

7.9.10 Align (origin) text with model

To rewrite a text based on the model. A text can be the source text on which the model is based. The origin text can be aligned with the resulting terminology after model engineering. In practice, the domain model plays the most important role in this usage.

7.9.11 Base other specifications on the MuDForM model

Build other, non-MuDForM compliant, specifications with the MuDForM model.

7.9.12 Specify model based test

Specify a set of tests to check the elements in the model. Typically, such a set is used to validate one or more implemented features, or to check if a system complies with a domain model.

7.9.13 Implement model in software system

Derive a (software) system specification from a MuDForM model. This can be both from a domain model and from a feature specification. It can also be the derivation of an interface from a context model.

7.9.14 Specify requirements

Use the (domain) model as the terminology of requirements. That means that all words in a requirement are defined by the model. In practice, one also uses requirement terminology, like "must, should, will, the system, and, or, able to, ...) For example, you can use OCL (Object Constraint Language) to make a formal specification of a requirement. The OCL specification may only use elements from the MuDForM model (especially the domain model) or from other formal requirements.

8 Subset of UML

The MuDForM viewpoints are defined as an extension of a subset of UML notations. The used subset is:
Class, activity, action, attribute, parameter, constraint, actor, object, link, selection, merge, fork, join, association, generalization, dependency.
The used diagram types are package diagram, class diagram and activity diagram.
The UML notation will be used as much as possible. But the following extensions are used:
Figure 40: There are different types of model packages indicated by different stereotypes: domain model, context, and feature model. A package can have one or more of those stereotypes. Like in UML, a package forms a namespace.
Figure 41: There are also more stereotypes used to indicate special types of classes (domain class) and special types of activities (domain activity, function, feature, operation).

9 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".

9.1 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.

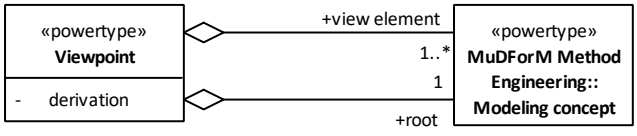


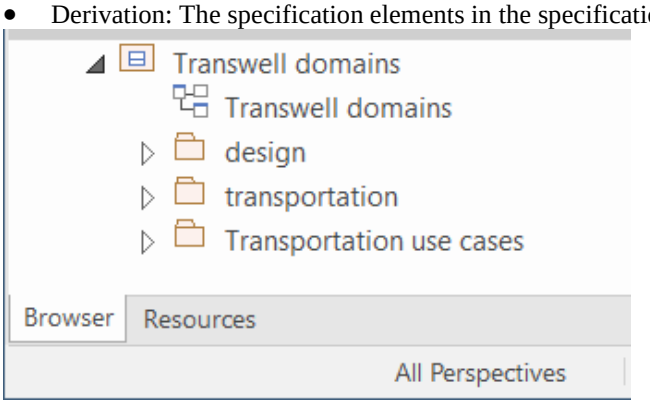
Figure 48: Viewpoints pattern

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.

9.2 For a MuDForM Model

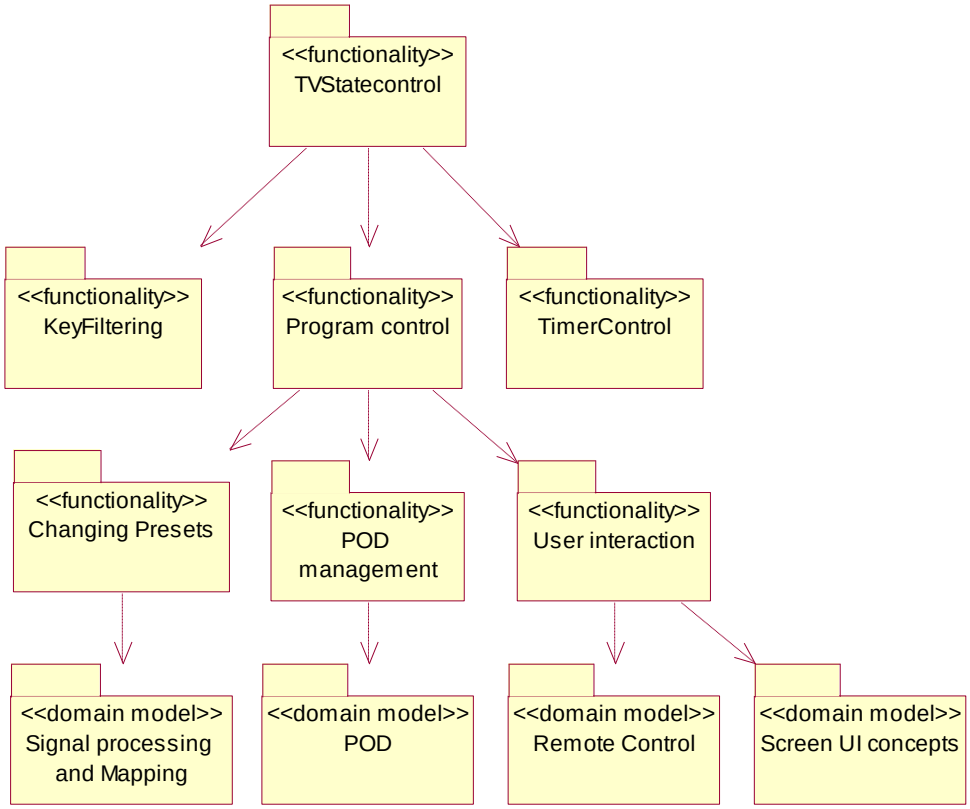
9.2.1 Declarations view:

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



9.2.2 Dependencies view

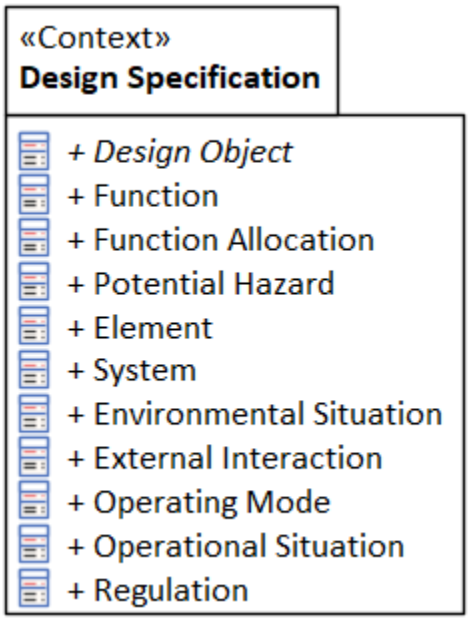
- Root: MudForM model
- Derivation: the references in the specification space dependencies of all the specification spaces in the specification declarations of the MuDForM model. So, show all the specification spaces contained in the MuDForM model, and show the dependencies between those spaces.



9.3 For a context

9.3.1 Declarations view

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



9.3.2 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.

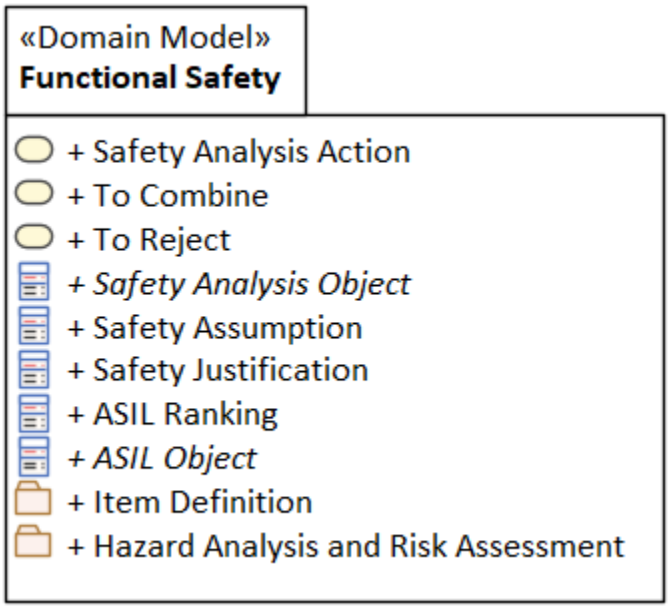
9.3.3 Actors view

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

9.4 For domains

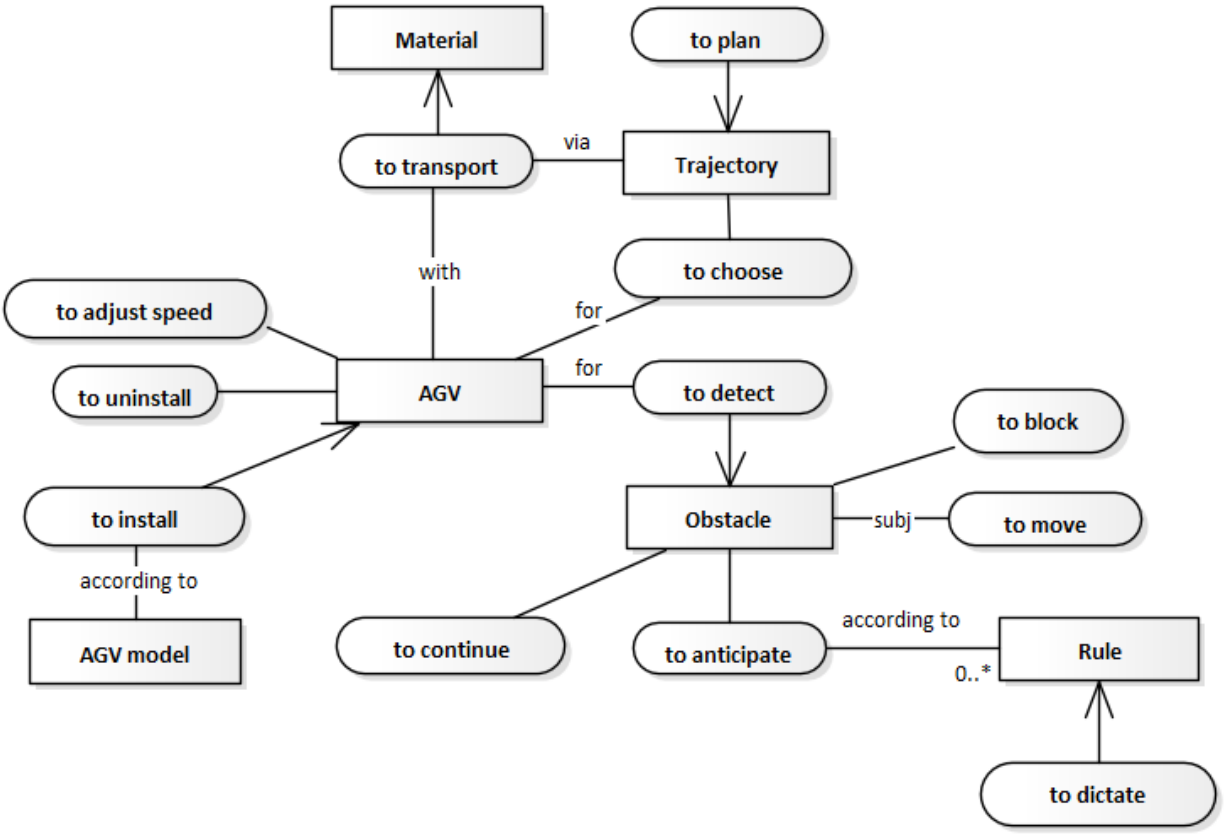
9.4.1 Declarations view

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

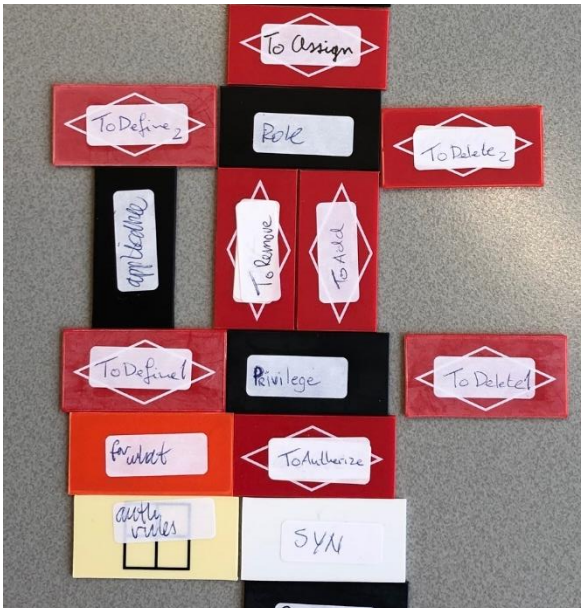


9.4.2 Interaction view

- 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.



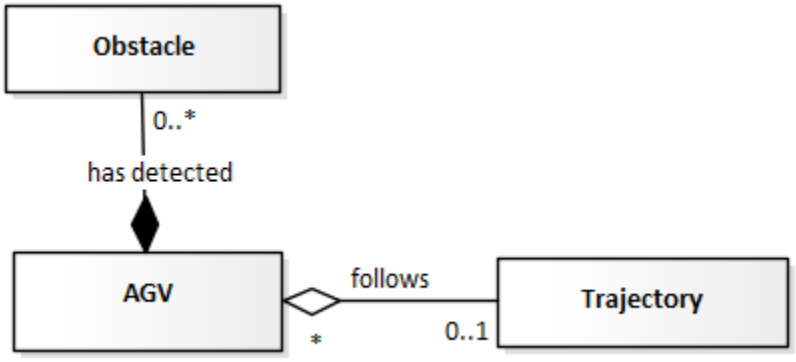
Example with KISS domino:



9.4.3 Class view

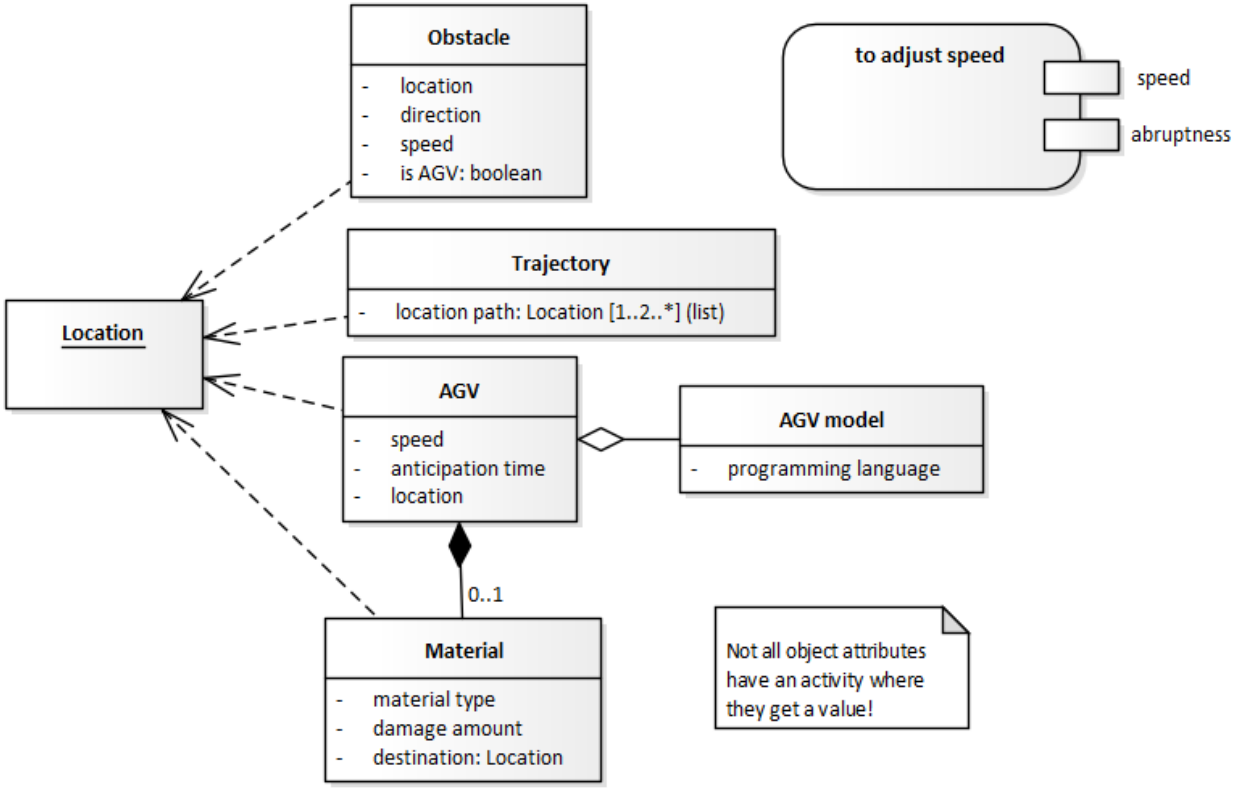
- Root: domain
- Derivation:
- Domain classes in the domain.
- References in the Life dependencies of the domain classes.

- 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.



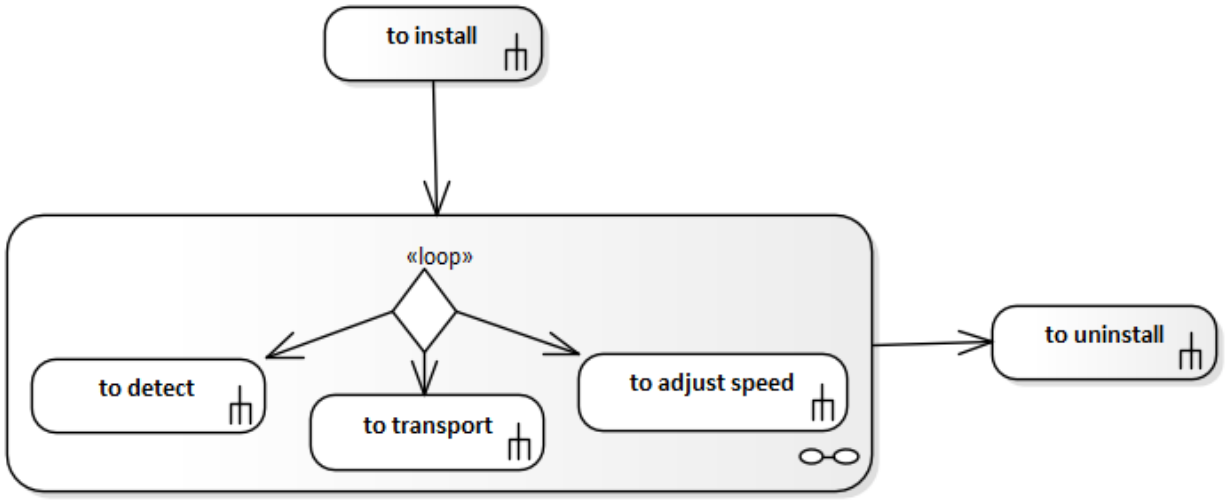
9.4.4 Attribute view

- 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.



9.4.5 Object lifecycle view

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



9.4.6 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. But it looks the same as a function lifecycle, except that all steps are Operations.

9.5 For features

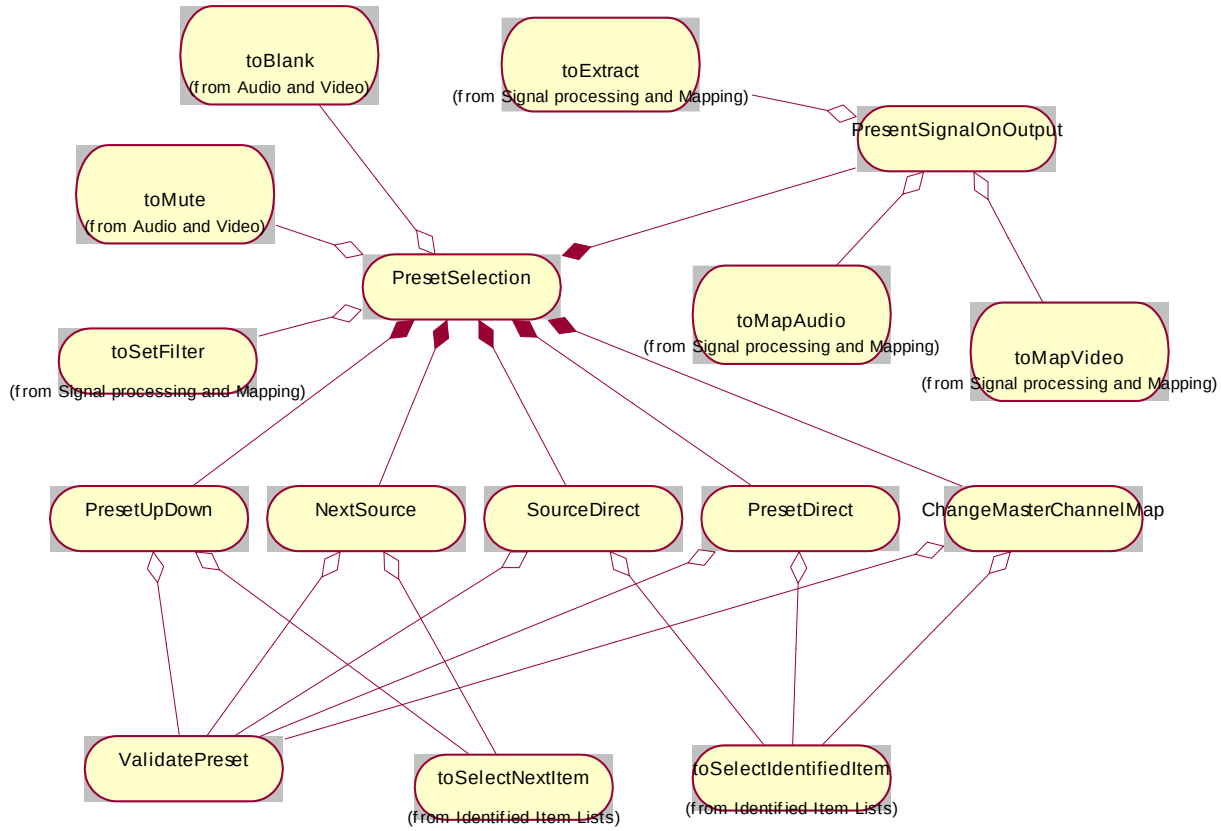
9.5.1 Declarations view

- 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.

9.5.2 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.

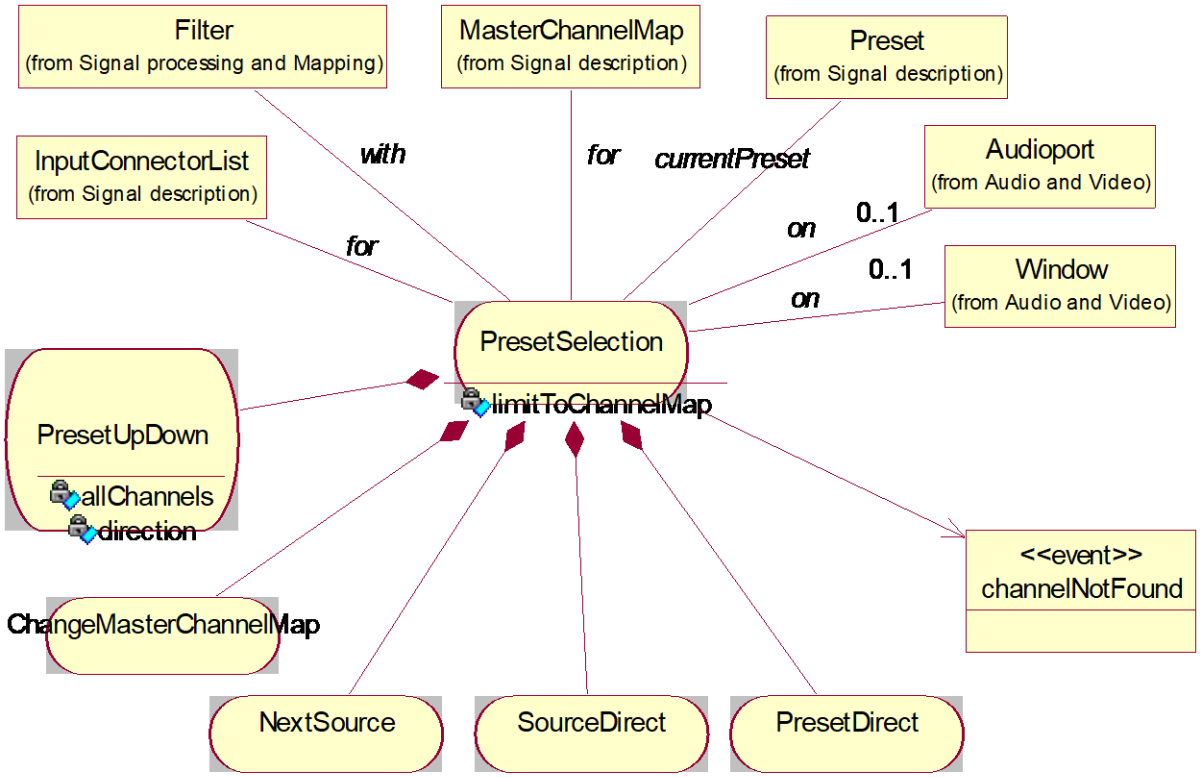
```
if      this event is equal to the n events before
        longPress = true;
        if      the table contains longPress for this event
                for all events in the selected column
                        generate the event as indicated in Table 1;
elif    this event is equal to the m events 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

9.5.3 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	

10 Model usage

This package introduces the different usages of a MuDForM compliant model. We list these activities, but we will not work them out in detail now.

10.1 Use MuDForM models diagram

The diagram shows various activities that involve a MuDForM compliant model. It is intended to serve as placeholders for more detailed steps and guidelines to implement those activities.

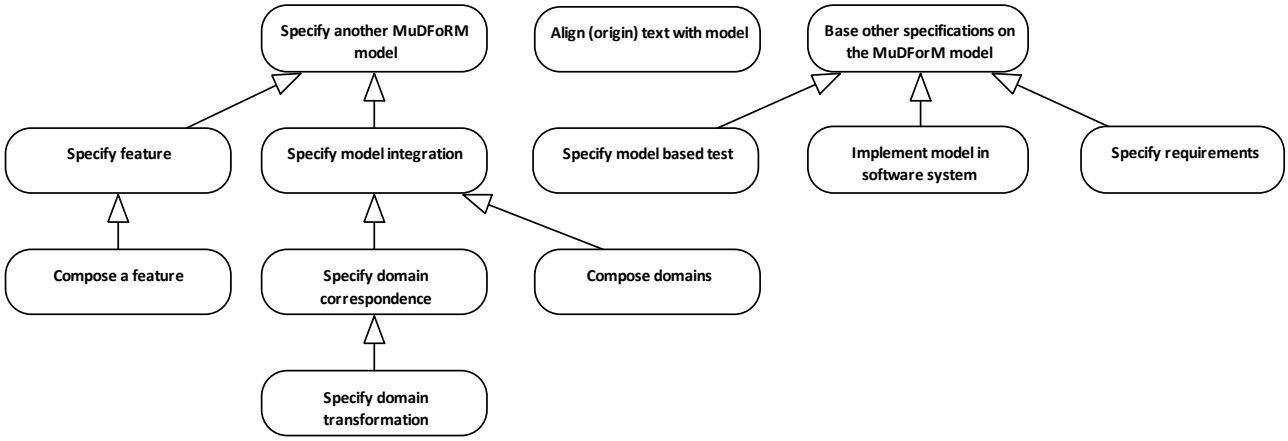


Figure 49: Use MuDForM models

Name	Definition
Specify another MuDFoRM model	To use a model in another MuDForM specification. This activity has sub-activities that address a specific type of specification.
Specify feature	To specify what should be true in the context of a specific feature. This includes specifying what shall happen (behavior) and what shall exist (objects). A feature is specified in terms of one or more domains and/or other features, and typically also uses elements from one or more contexts.
Compose a feature	To create a feature out of one or more existing features.
Specify model integration	To create a new MuDForM model out of two or more existing models.
Specify domain correspondence	Specify the relation between two or more domains in the context of a feature. A correspondence specification consists of correspondence rules between the elements from the involved domains.

Specify domain transformation	Specify the transformation from a source domain model to a target domain model. The target domain might also be defined in a MuDForM model. A transformation specification consists of several transformation rules.
Compose domains	To create a new domain model out of one or more existing domain models.
Align (origin) text with model	To rewrite a text based on the model. A text can be the source text on which the model is based. The origin text can be aligned with the resulting terminology after model engineering. In practice, the domain model plays the most important role in this usage.
Base other specifications on the MuDForM model	Build other, non-MuDForM compliant, specifications with the MuDForM model.
Specify model based test	Specify a set of tests to check the elements in the model. Typically, such a set is used to validate one or more implemented features, or to check if a system complies with a domain model.
Implement model in software system	Derive a (software) system specification from a MuDForM model. This can be both from a domain model and from a feature specification. It can also be the derivation of an interface from a context model.
Specify requirements	Use the (domain) model as the terminology of requirements. That means that all words in a requirement are defined by the model. In practice, one also uses requirement terminology, like "must, should, will, the system, and, or, able to, ...) For example, you can use OCL (Object Constraint Language) to make a formal specification of a requirement. The OCL specification may only use elements from the MuDForM model (especially the domain model) or from other formal requirements.