

- 
- 
- 

# Klasse Objecten

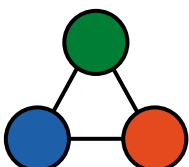
- 
- 
- 

## Unification of Static and Dynamic Semantics of UML

A Study in redefining the Semantics of the UML using the pUML OO Meta Modelling Approach

*Anneke Kleppe and Jos Warmer*

- 
- 
- 



- 
- 
- 
- 
- 
- 
- 
-

# Contents

<b>Contents</b> .....	<b>i</b>
<b>List of figures</b> .....	<b>iii</b>
<b>Legal notice</b> .....	<b>v</b>
<b>Summary</b> .....	<b>vi</b>
<b>History of this document</b> .....	<b>ix</b>
<b>Chapter 1</b>	
<b>Introduction</b> .....	<b>1</b>
1.1 A new semantics definition for the UML .....	1
1.2 The principles underlying this study .....	1
1.3 Related work .....	5
1.4 The approach taken in this study .....	7
<b>Chapter 2</b>	
<b>The kernel meta model</b> .....	<b>9</b>
2.1 The language used to define the kernel package .....	9
2.2 Overview of packages defined in this chapter .....	10
2.3 Common structure of the packages .....	11
2.4 The core package .....	12
2.5 The features package .....	17
2.6 The immutables package .....	19
2.7 The associations package .....	21
2.8 The actionExpressions package .....	23
2.9 The compoundAction Package .....	28
2.10 The messaging package .....	32
2.11 The modelmanagement package .....	36
2.12 Some notes on the syntax and OCL used in this definition .....	37
<b>Chapter 3</b>	
<b>Syntax - the UML diagrams</b> .....	<b>39</b>
3.1 Overview of packages defined in this chapter .....	39
3.2 The graphical symbols package .....	39
3.3 The statechart diagram .....	40
3.4 The activity diagram .....	44
3.5 The sequence and collaboration diagrams .....	45

**Chapter 4**  
**Some examples of additional packages** .....47

- 4.1 The operationCall package .....47
- 4.2 The actionClause package .....48
- 4.3 Suggestions for other packages .....52

**Chapter 5**  
**Extensibility and profile definitions** .....53

- 5.1 Extensibility of this meta model .....53
- 5.2 Known omissions .....54
- 5.3 Conclusion .....54

**Appendix A**  
**Imported types** .....55

- A.1 Predefined immutable types and values .....55

**Appendix B**  
**Definitions of meta classes** .....57

**Appendix C**  
**Overview of kernel model concepts** .....61

**References** .....63

# List of figures

figure 1-1	Different views on model using one semantics. . . . .	2
figure 1-2	Seperate semantics for different views . . . . .	3
figure 1-3	The package structure used in MML . . . . .	6
figure 1-4	The OMG 4-layered architecture . . . . .	6
figure 2-1	Overview of kernel subpackages . . . . .	10
figure 2-2	Relation between OMG layers and common structure of packages . . . . .	11
figure 2-3	One line representing two unary associations . . . . .	12
figure 2-4	Two arrows representing two unary associations . . . . .	12
figure 2-5	The core.model.concepts package. . . . .	13
figure 2-6	The core.instance.concepts package . . . . .	14
figure 2-7	Relations between local snapshots . . . . .	15
figure 2-8	The core.semantics package . . . . .	17
figure 2-9	The features.model.concepts package. . . . .	18
figure 2-10	The immutables.model.concepts package. . . . .	19
figure 2-11	The immutables.instance.concepts package . . . . .	20
figure 2-12	The immutables.semantics package . . . . .	21
figure 2-13	The associations.model.concepts package . . . . .	22
figure 2-14	Instance representation of a two-way association. . . . .	22
figure 2-15	The actionExpressions.model.concepts package . . . . .	24
figure 2-16	The actionExpressions.instance.concepts package . . . . .	25
figure 2-17	The actionExpressions.semantics package . . . . .	27
figure 2-18	The compoundActions.model.concepts package . . . . .	29
figure 2-19	The compoundAction.instance.concepts package. . . . .	30
figure 2-20	The compoundAction.semantics package. . . . .	31
figure 2-21	The messaging.model.concepts package. . . . .	33
figure 2-22	The messaging.instance.concepts package . . . . .	34
figure 2-23	The messaging.semantics package . . . . .	35
figure 2-24	The modelmanagement.model.concepts package. . . . .	36
figure 2-25	Two views: attributes versus associations. . . . .	38
figure 3-1	The stateMachine.model.concepts package . . . . .	41
figure 3-2	The stateMachine.instance.concepts package. . . . .	42
figure 3-3	The stateMachine.model.syntax2concepts package . . . . .	43
figure 4-1	The operationCall.instance.concepts package. . . . .	47
figure 4-2	The actionClause.model.concepts package. . . . .	50
figure 4-3	The actionClause.instance.concepts package . . . . .	51
figure 4-4	The actionClause.semantics package . . . . .	52
figure A-1	The predefTypes.model.concepts package . . . . .	55
figure C-1	Overview of all model concepts from the kernel . . . . .	61



# Legal notice

This report is copyright © Klasse Objecten, Soest, Netherlands. All rights reserved.

Permission to make digital or hard copies of part, or all of this work for personal or classroom use, is granted without fee, provided that the copies are not made for profit, or commercial advantage, and that copies bear this notice, and full citation on the first page.

# Summary

This report describes a study into the rearchitecting of the semantics of the UML. The study is based on the pUML OO meta modelling approach laid down in [Clark2000]. In this approach the UML is defined using a small subset of the UML itself. [Clark2000] presents such a subset, which is based on static semantics only. This study defines a different subset, one that is based on both static and dynamic semantics. In effect, this subset is a small kernel language, which is highly extensible, that forms the heart of a family of UML variants.

## Key notions

The kernel language is based on two main principles:

- **The static and dynamic viewpoints on the system that is being modeled, can not be separated.**
- **The meaning of a model written in UML, the semantics, must be expressed in a manner that users of UML will be able to understand.**

We use the following key notions in this report, that all follow directly or indirectly from the above principles:

- A strict separation of the items in a UML model, and the things they describe. E.g. between a class, and an instance.
- A strict separation between the items in a UML model, and the way they are denoted. E.g. between a class, and the rectangle with name that is used to denote it.
- A strict separation between the definition of a feature, the reference or call of a feature, and the execution of that feature. E.g. an operation is defined as `Plus(x,y: Integer):Integer`, it is referenced (called) as `Plus(1,4)`, and its execution produces the result 5.
- The use of static views as part of a dynamic view. A dynamic view on a system or model is considered to be a sequence of snapshots. Each snapshot holds all values, relations, etc. of the object at that point in time. A static view is nothing more or less than a view of one snapshot, disregarding all other possible snapshots in time.
- The use of snapshots local to an object to indicate changes in that object in time. Time, and snapshots are not global to the system.
- A strict separation of the transport mechanism from the changes in the objects involved in a communication. E.g. an object sending a signal simply puts the signal in its output queue, the transport mechanism then transports it to the input queue of the target object or target objects, each target object takes the signal out of its input queue.
- The use of very general concepts to define class and object, to facilitate the definition of concepts that are very much alike to them, but need a slightly different definition. E.g. component type, and component instance, agent type, and agent instance.
- The use of UML itself to define its semantics. Amongst theoreticians it is popular to define semantics using a mathematical formalism. The disadvantage of using a mathematical formalism is that only a small minority of the people using UML will be able to understand it.

## The semantic domain

Our kernel language makes a clear distinction between the items that are described, or specified, by a UML model, the domain elements, and the items in the model itself, the model elements. The set of domain elements is the semantic domain of our language. The domain elements are objects that

have *slots*, which are references to other values, either data values or other objects. Objects have identity, but their slots may change. In this view everything that exists in our domain, i.e. the world we can model, is an object or a data value, there are no other items besides that. But because we foresee that there will be other items besides objects, e.g. component instances, that will be modeled by the UML, we do not call these items objects, instead we have chosen the more general name *mutable value*. (For the rest of this summary we will continue to use the term object.)

So far, we have described the domain elements from a static viewpoint. From a dynamic point of view, we are interested in the changes in time in the slots of an object. Thereto we define that every object is aware of a history of changes to its slots. Every time a change in a slot takes place the object records a snapshot that holds, for every slot, the combination of the slotname and the value referenced. A series of these snapshots form the history of changes to an object.

## The abstract syntax

The items present in the UML models, the model elements, are not equal to the domain elements they describe, instead they are a model of these items. Our kernel language also defines these model elements. The set of model elements and their relations, is the abstract syntax of our language. In the object-orientation paradigm the model of an object usually is its class, but similar to the renaming of object to mutable value, we use the more general name *role* to indicate the model of a mutable value. The specific class concept is defined as a subtype of role. This enables us to speak about the model counterpart of object-like instances. For example, the model counterpart of a component instance is a component type, which is not a class, but another subtype of role.

Every role contains a set of features, and every role can be part of a generalization or inheritance relation with other roles. All features of a role have three aspects: the definition, the reference (or call), and the execution. Only the execution aspect of a feature resides in the world of the domain elements, the other two aspects belong to the world of the model elements. In the kernel defined in this report this distinction is clearly made through the use of three meta classes: Definition, FeatureRef, and FeatureExec.

## The semantics

The semantics of each model element is given by relating it to one or more domain elements. For the static model elements this is rather straightforward, but for the dynamic model elements the snapshots of the objects need to be taken into account. The semantics of each feature is given by relating the feature definition to a feature reference, and the feature reference to a feature execution. The feature execution may cause changes in the slots of an object, thereby creating new snapshots.

Each (dynamic) feature that may cause changes in the snapshots is associated with an ActionExpression, which is a model element specifying a thing that can happen in a software system. A feature execution is in effect the execution of an action expression, which is represented by the domain element ActionExpExec. Some primitive action expressions are defined, e.g. the writing and reading of an attribute in the same object, and the creation of a new object by a class.

Calling (or referencing) a feature of an object other than the current one, is specified through a messaging mechanism. For this purpose each object is associated with an input and an output queue. When an object calls a feature of another target object it puts a signal addressed to the target object on its own output queue. A transport mechanism (a bus) then takes care of the transport from the output queue to the input queue of the target object.

## The concrete syntax

This report is mainly concerned about the semantics and abstract syntax of the UML. Some notes on how the concrete syntax, i.e. the currently known UML diagrams, can be defined, are given in a separate chapter.

**Extensibility**

The kernel also defines a mechanism for structuring models into small(er) parts called packages. This mechanism is used to build some examples of extensions to the kernel package.

# History of this document

## Version 0.1, November 2000

The first version combines the information on the semantics of the UML already described in three papers:

- "Making UML Activity Diagrams Object-Oriented" [Kleppe2000a], and
- "Extending OCL to Include Actions" [Kleppe2000b], and
- "Integration of Static and Dynamic Core for UML: A Study in Dynamic Aspects of the pUML OO Meta Modelling Approach to the Rearchitecting of UML" [Kleppe2001].

## Version 0.2, July 2001

This version synthesises the ideas from the above mentioned papers into one kernel package, and adds a large number of metaclasses to express them in. It is the first version in which a complete kernel language is defined, and which is published through the internet ([www.klasse.nl/english/uml/semantics.html](http://www.klasse.nl/english/uml/semantics.html)).



# Chapter 1

## Introduction

"... The metalanguage of a formal definition must not become a language known to only priests of the cult. Tempering science with magic is a sure way to return to the Dark Ages." [Marcotty76]

---

### 1.1 A NEW SEMANTICS DEFINITION FOR THE UML

The Unified Modeling Language [UML1.3], the standard set by the Object Management Group for object-oriented modeling and design, has rapidly gained acceptance amongst system analysts and designers. The standard contains definitions of the UML semantics, the UML notation, the object constraint language (OCL), the standard profiles, a CORBA facility interface definition, and an XMI DTD specification.

For a large number of scientists the UML definition has been a subject of study. Many of those found the definition of the semantics lacking in precision, and proposed different ways of building a semantic definition (e.g. [Evans98]). The fact that the UML definition was not yet complete, was recognised by the OMG, and a request for proposals on a definition of the action semantics of the UML was sent out (OMG RFP ad/98-11-01). The response to this request is known as the Action Semantics [AS2001].

On many points we agree with the critique, on UML that was published during the last years. The challenge is, however, not to provide the best critique but to build a UML semantics that will stand the test of the professionals in our international community. Therefore, we do not address the flaws of the current semantics definition, instead in this study we define the semantics of the UML yet again. We define a small number of core concepts, the kernel, from which all concepts in the UML can be defined, thus creating a self-consistent, coherent, and extendable language. The kernel meta model is described in chapter 2 ("The kernel meta model").

This report describes the kernel of the meta model of a family of languages that can all be called UML. It is consistent within itself, but its use lies in the extensions that can be build upon this kernel. The report does not answer all the questions on the semantics of UML. Although we are convinced that the kernel described in this report is indeed a very good basis to built a complete semantics for the UML, we lack both time and means to do the complete job. We hope to gain support from people who are able and willing to fill in the parts they are missing, thus creating other languages within this family, that will hopefully support quality software engineering. (See chapter 5 ("Extensibility and profile definitions") for more information on an open source project we envision.)

The motivation behind this work is to a large extend expressed by the quote given at the start of this chapter. It should be possible to express the semantics of the UML in such a manner that it is formally correct, sound and complete, and still readable for the people who use the UML, i.e. ordinary software developers. In the rest of this chapter you can find more information on the motivation behind this work.

---

### 1.2 THE PRINCIPLES UNDERLYING THIS STUDY

The UML is according to the OMG specification "the proper successor to the object modelling languages of three previously leading object-oriented methods". It is an *object-oriented* language, and the principles under-

lying the object-oriented paradigm should be taken seriously in any definition of its semantics. In this section the most important of the principles used in this study are discussed. (No ordering of the subjects is assumed.)

### 1.2.1 All diagrams need one integrated semantics

Many people have been trying to build the semantics of one of the diagrams of the UML, e.g. [Kwon2000] on statecharts, [Engels2000] on collaboration diagrams, [Evans98] on class diagrams, [Övergaard98] on use cases. Because all diagrams are only views on one and the same model, these attempts are destined to fail in producing the right semantics for the UML. There should be one abstract syntax and one semantics of the model as a whole, and many concrete syntaxes that define that various views or diagrams. Figures 1-1 and 1-2 show the difference between both approaches.

### 1.2.2 Objects

In every object-oriented language, whether a programming language or a modelling language, everything is centered (oriented) around the concept 'object'. In this study this orientation has been well guarded. The notion of object is central, and it must therefore be one of the first to be defined. At the basis of this study lies the understanding that in the world we try to model using the UML, nothing can exist besides objects and data values. And even data values can be seen as very simple objects, they only have a value, and can not do anything. The 'real' objects are more complex, they have properties, and they can do, or execute, 'things'. Looking at the UML standard, we find that a such a 'thing' that is executed is called an action. This term will be used in the remainder of this study.

The fact that there are only objects and data values, also means that there are no other things than objects that perform (execute) actions. Or, to put it differently, everything that executes an action is (a subtype of) an object. Thus, in this study, a class does not have a separate state machine that executes, instead, the instances of that class are state machines that execute according to the specifications in the statechart. This might be considered by some, an unimportant play with words, but the different viewpoints behind both phrases lie worlds apart.

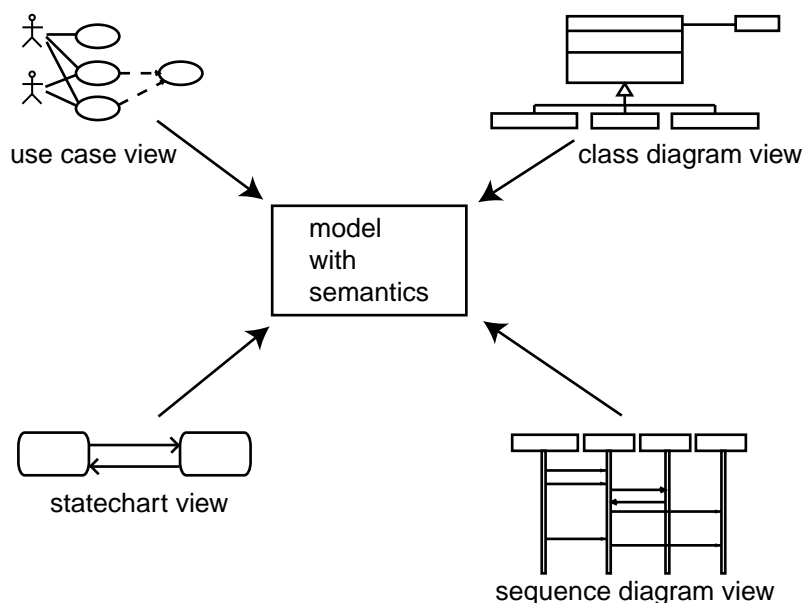


figure 1-1 **Different views on model using one semantics**

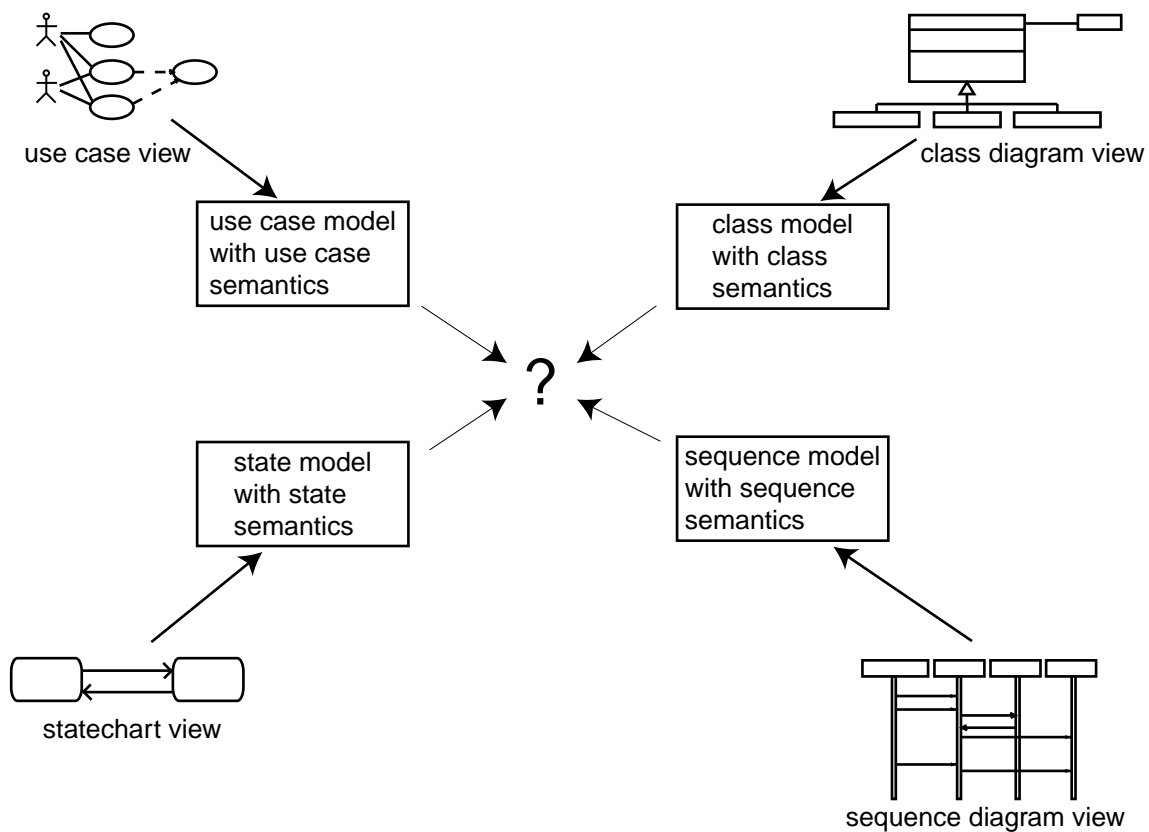


figure 1-2 **Separate semantics for different views**

### 1.2.3 Type - Instance dichotomy

An aspect of object-oriented modeling that is often confusing for the novice, is the type - instance dichotomy. Whenever it is said that a certain class has a property, what is really meant is that every instance of that class has that property. The terms class, object, and instance are (too) often used in a non proper manner. In this study there will be a clear division between the modelling aspects (the type-part of the dichotomy), and the instance aspects. See sections 1.3.1 and 2.3 for more details.

### 1.2.4 Responsibility

Another object-oriented principle we adhere to in this study, is that of responsibility. A responsibility is in the UML standard defined as “a contract or obligation of a Classifier”, where a Classifier can be either a Class, an Interface, a Node, a Component, a UseCase, a Datatype, or a SubSystem. Others define responsibility to be the “purpose or obligation of a system, subsystem or object” [Goldberg95]. The focus should therefore be on the obligations of the UML Classifier.

But here we encounter the diffusion that the type-instance dichotomy often brings. If we define a classifier to have obligations, actually we define each instance of that classifier to have those obligations. The instance has one or more responsibilities, which are realized by the execution of actions. More important is that the reverse holds too: every action that is executed, is executed by an object as result of the commitment of the object to its responsibilities. No action can be executed by any item, other than an object.

### 1.2.5 Encapsulation

Encapsulation is of course an object-oriented principle that needs to be addressed. Encapsulation can be defined as “the act of enclosing elements within a container” [Goldberg95]. Because in our universe there are only data values and objects, and data values do not have properties, it must be the object that “encloses elements”. To put it more boldly, the object *owns* its properties.

One of the possible properties of an object is a link (a reference to another object). The object must own its links. For an association, which defines links for every instance of the related classes, it must be clear which of the classes defines the ownership of the corresponding link. In other words, it must be clear which object owns the link. This is the reason why in the core packages of the semantics definition, we use directed associations. The class from which the association originates (that is, the side that does not have an arrowhead) is the owner.

For easing the modelling task it may be useful to work with a two-way association. We encountered such difficulties in defining a two-way association that we decided that it should not be part of the kernel language. (See section 2.7 (“The associations package”) for more information.)

### 1.2.6 Object independence and delegation

A point that has always been stressed in object-oriented languages, is that objects are independent. This viewpoint is another of the underlying principles of this study, specially in the definition of the communication between objects. All objects are considered to be independent, they have a (virtual) input and an output queue through which they may communicate with other objects, but these input and output queues are independent of the input and output queues of other objects. The messages that go to and from these queues are constrained to the definitions of the transport mechanism used. The principle of delegation can be described as: the ability of one object to send a message to a second object in order to try to fulfil one of its responsibilities.

Each form of communication is defined separately from the definitions of the basic concepts of UML, like class, object, and attribute. One transport mechanism is defined in this study, and more may be defined to take their place. The user of the UML may pick the transport mechanism that fits his or her needs best.

### 1.2.7 Object creation and deletion

In many object-oriented languages garbage collection is an integral part of the underlying system. Objects need not be deleted, they just fall out of the scope of the system. In this report we have taken the same approach: our domain universe is full of objects, which always exist, whether or not we are aware of them. The creation of an object is merely getting aware of it, the object is getting into scope. The deletion of an object is simply removing it from scope.

### 1.2.8 Static and dynamic semantics

There is a fundamental problem in the separation of static and dynamic semantics. One of the principles of object-orientation has always been to regard statics and dynamics as *views* on one and the same (set of) concept(s), e.g. a model. This means that the semantics of an object-oriented modelling language should not (can not) be divided into two (as in the UML1.3 specification and the action semantics), instead there must be one semantics only, covering both static and dynamic aspects.

Another argument for this synthesis is that for modelling purposes time needs to be regarded as discrete. A static view of a mutable entity is nothing more than taking one of the values of that entity in time, regarding that value as ‘the’ value of the entity, and disregarding all other values in time. A static view is, as one might say, ‘less’ than a dynamic view. Therefore the semantics of the UML must be build from a dynamic viewpoint, taking into account static aspects.

As in the Action Semantics [AS2001] we use a purely ‘local’ and relative notion of time. Every mutable entity has its own ‘time’, which means that it migrates through a number of step-by-step changes, each change taking place after the other.

---

### 1.2.9 A language defined in itself

Often the semantics of a new language is defined by giving a mapping of the abstract syntax of the new language to an already known language, e.g. a mathematically based formalism. A drawback of this approach is that often the already known language is a language known to a minority of the people that will or would be using the newly defined language. The quote at the start of this chapter summarises our opinions on this topic.

In this report we have chosen to use the approach from the UML specification, which is to define the language using constructs from that language itself. We define the abstract syntax, the modeling universe, and a domain of elements that can be modeled by our language. The semantics consist of a mapping between the modeling universe and the domain.

---

### 1.2.10 A family of languages

As discussed in [Cook99], the user of the UML really has a family of languages to choose from, picking the parts that fits his or her needs best. This idea is expressed in this report too. The kernel meta model described in chapter 2 (“The kernel meta model”) is itself a template language. Some parts must be instantiated before the language can be used. These parts include the data types and data values, the syntax of the meta class Name, and the meta classes VisibilityKind and ParameterDirectionKind. Our choice for these classes, as used in the definition of the kernel itself, is described in appendix A.

---

## 1.3 RELATED WORK

This study is based on three earlier documents: the UML 1.3 specification<sup>1</sup>, the pUML OO meta modelling approach [Clark2000], and the Action Semantics [AS2001]. The UML specification need not be discussed, a short introduction to the other two is given in this section, as well as an introduction to the OMG 4-layered architecture.

---

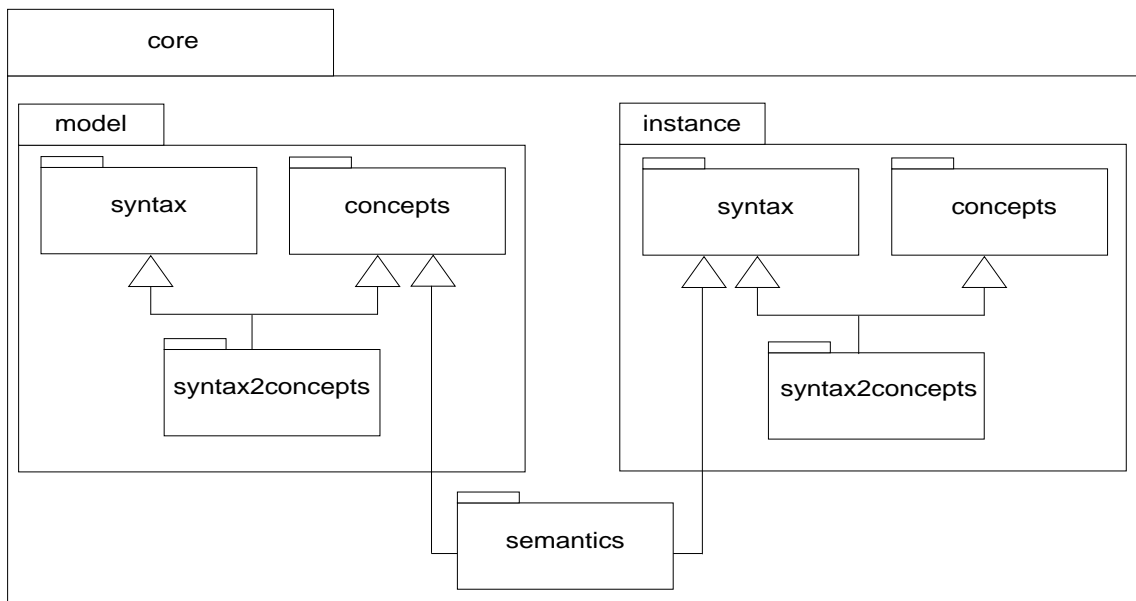
### 1.3.1 The Meta Modelling Language

The MML has been defined by members of the pUML group in collaboration with IBM as a (potential) response to the OMG request for proposals for the UML 2.0 Infrastructure [OMG2000]. Its main purposes are to function as the core of a rearchitected definition of the UML, and to be the language in which this rearchitecture definition is to be expressed. Its most outstanding feature is that the complete language is build by extending one small package called StaticCore. As indicated by its name, the core of the MML has a static view on the domain.

Every package in the MML contains three subpackages: model, instance and semantics. Both the model and instance packages contain a concepts and a syntax subpackage, and a mapping between these two. Each model subpackage describes the modelling concepts defined by the package, each instance subpackage describes the semantic domain of the modelling concepts. The semantics package defines a mapping from modelling concepts to its semantic domain, whereas The syntax package maps syntactical constructs (lines, boxes, etc.) to concepts. Figure 1-3 (taken from [Clark2000], page 35) shows this mapping.

---

1. The UML 1.4 version has been approved since the start of this study. The changes to version 1.3 are minor, therefore we decided there was no need to adapt the results of our study to the new version.

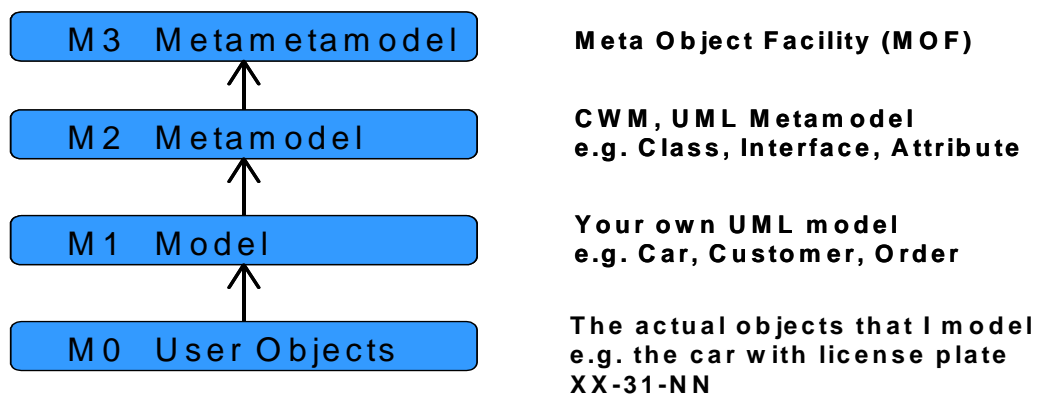
figure 1-3 **The package structure used in MML**

### 1.3.2 The Action Semantics

The Action Semantics submission is a response to a request for proposals from the OMG [OMG98]. It proposes a definition for the semantics of actions in UML. The approach taken is oriented towards the dynamics of the system define by the UML model. Like MML, it separates the model from the instance level. In the Action Semantics these are called 'Actions', and 'Execution Semantics'. The 'Actions' part describes the which kinds of actions can be specified in an UML model, and how they can be build from other actions. The Execution Semantics explains what the result of the execution of an action is. Like MML, the Action Semantics tries to define all concepts building from a single core model, but this core model is defined in a non-normative part of the action-semantics specification (in appendix A of [AS2001]), thereby leaving the "grounding" of their execution model open.

### 1.3.3 The OMG 4-layered architecture

The OMG uses a 4-layered architecture (see figure 1-4) for its standards. Layer M0 contains concepts from 'real life', e.g. the red car with license plate XX-31-NN. Layer M1 contains the concepts needed to reason about the real life concepts, e.g. the concept 'car'. Note that the normal use of the UML lies in the M1 layer,

figure 1-4 **The OMG 4-layered architecture**

here reside the classes of which the instances are the real life counterparts. Layer M2 contains the concepts needed to reason about concepts from layer M1, e.g. a Class, an Association. The UML meta model resides (largely) in this layer. The fourth layer (M3) introduces yet another meta level. It contains concepts needed to reason about concepts in the M2 layer.

Note that the separation between the layers is purely superficial, in order to understand the matter at hand better. Actually all items in all models are present in the M0 layer, simply because you and I are referring to them. (Does the turnover of a company exist? Yes, it does because we speak about it.) This means that some of the items that are defined to reside on layer M0 can and will be present on other layers too. For example, the string value 'name' is present on all four layers because it is defined to be an indication of a (model)element on all layers.

---

## 1.4 THE APPROACH TAKEN IN THIS STUDY

This study follows the path set out by the pUML group in [Clark2000], to build a rearchitected definition of the UML, that meets the following requirements (quoting from [Clark2000]):

- [1] "It should be precise to the degree that conformance can be checked systematically, without argument and, preferably automatically, and that self-consistency of the definition can be established.
- [2] It should be comprehensive, covering syntax, both concrete and abstract, and semantics. On the other hand, redundant and overlapping concepts should be kept at a minimum.
- [3] It should accept that UML is a family of languages, providing mechanisms that allow profiles and language extensions to be defined in a controlled and managed way, and which makes the relationships of profiles and extensions to existing language fragments explicit and unambiguous.
- [4] The definition should be accessible to tool builders and those involved in the standardization of the language."

These requirements are met by an object-oriented meta modelling approach as the MML. The MML however, is based on the static aspects of the UML. The [Clark2000] study contains some initial work on dynamic aspects, but this is not integrated into the core of the MML. Another study by Reggio and Astesiano [Reggio2001], proposes a dynamic core build on top of the static core of MML. Again, static and dynamic core are separated. As argued earlier, a static view is just a view at one moment in time of a dynamic view, thus the semantics of the UML must be build from a viewpoint that integrates static and dynamic aspects. We can not separate a static from a dynamic core, and therefore a meta modelling language, as defined in this report, must be build from one integrated core.

One integrated core in which static and dynamic viewpoints are synthesized is our first starting point. The second starting point is that we feel that the definition of a language, including its semantics, should be understandable to those who use the language. Because most object modelers do not have a mathematical background, we do not use any mathematical notations to define the semantics. Instead we use object modeling techniques to explain what an object model written in UML means.

### 1.4.1 The structure of this report

Only one of the packages in the kernel of our language still resembles the MML. The MML *staticCore* package has been adapted to include the core dynamic aspects, and renamed to *core* package (section 2.4). Besides that, a number of packages that describe static and/or dynamic aspects, have been added to the kernel in the remainder of chapter 2 ("The kernel meta model").

In the current semantics definition the distinction between the concepts (the abstract syntax) and the (concrete) syntax is sometimes not completely clear. To give an example of how to define the syntax of the UML, based on the kernel meta model, we have included some notes on most of the UML diagrams in chapter 3 ("Syntax - the UML diagrams").

To show the feasibility of extending the definition given in chapter 2, some packages are defined that describe non-basic dynamic and static aspects, like operation call communication and components. These can be found in chapter 4 (“Some examples of additional packages”).

Chapter 5 (“Extending the OCL to include dynamic constraints”) describes some of the consequences of the rearchitecting of the UML definition on the Object Constraint Language. One of the requirements above is that the definition should provide mechanisms that allow profiles and language extensions to be defined.

When defining a kernel language extensibility is foremost important. Our views on how to extend the kernel language to fit the needs of today’s practice are included in chapter 5 (“Extensibility and profile definitions”).

The result of this study is a definition of the semantics of the UML that bears in it static as well as dynamic aspects. It is a definition of the core of the UML that shows that extensions can be made fairly easy, in a controlled and manageable manner, and that profiling one of the family of UML languages can be done in a clear and unambiguous way.

There is however, one disadvantage to the meta modelling language approach, the way in which packages after package inherits from another builds very deep inheritance trees. During years of practical experience with object-oriented modelling, we learned that deep inheritance trees tend to make the system or model less flexible. Only time will tell whether this is true for the UML as well.

# Chapter 2

## The kernel meta model

This chapter describes the packages that build up the kernel of the rearchitected UML meta model.

---

### 2.1 THE LANGUAGE USED TO DEFINE THE KERNEL PACKAGE

The kernel of the rearchitected UML language is defined in one package called *kernel*. The kernel package is a parameterized package, i.e. it contains a number of parameters that need to be actualized in order for the package to define a complete language, in other words it is a template package. The parameters that need to be instantiated are listed in table 1. In this report the kernel package has been actualized by the package *PredefTypes* defined in appendix A (“Imported types”). In table 1 you can find the mappings used.

To make a complete language from the kernel package, the user may define his/her own extensions. For instance, by extending the *DataType* and *DateValue* classes in the kernel package to his/her preferred data types and values, the user of the language can decide which data types are part of the language he or she is about to use, thus e.g. making a direct relation between the UML and the programming language used. Although there is no hard evidence in the text of the UML specification, we believe it to have the same intention. The definition of the kernel package in this chapter uses the concepts from the kernel itself, and extensions from the *PredefTypes* package.

Next to actualizing these parameters, and defining type extensions, the user of the complete language, which is here defined, needs to indicate the transport mechanism used for messaging between objects (see section 2.10).

In summary, to build a complete (usable) language from this kernel one has to:

- actualize the parameters,
- extend the types by subtyping, and
- define, or chose a transport mechanism.

By building our kernel package as a parameterized package, in effect we have defined a set of languages. Every time this kernel package is actualized with a different set of elements, a new variant of the UML language is created.

---

kernel parameter	predefType element
Boolean	BooleanType
MultiplicityKind	MultKindType
Name	StringType
Number	IntegerType
ParameterDirectionKind	ParDirKindType
VisibilityKind	VisKindType

**Table 1.** Mapping of kernel parameters to actual values

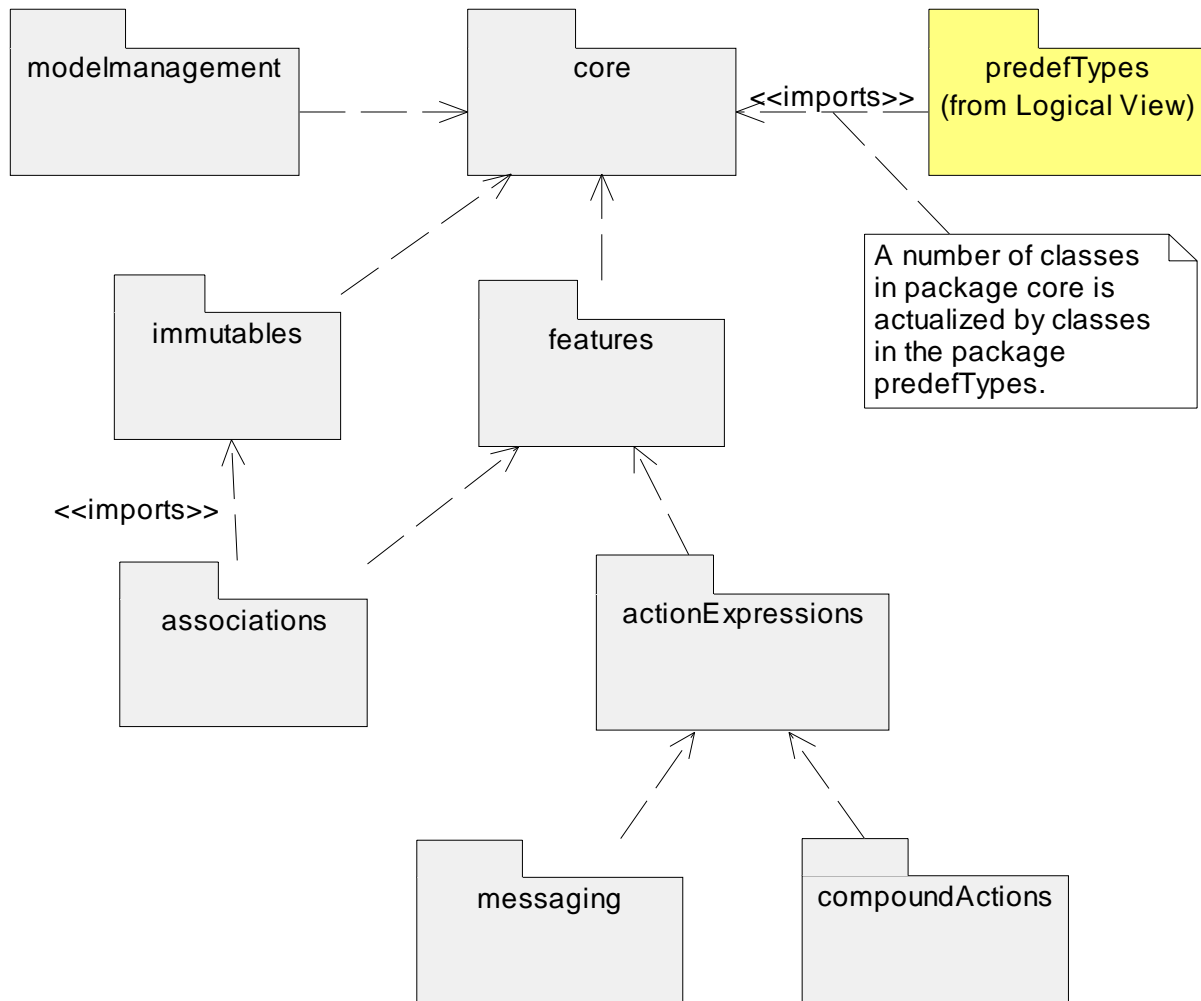


figure 2-1 Overview of kernel subpackages

## 2.2 OVERVIEW OF PACKAGES DEFINED IN THIS CHAPTER

In figure 2-1 you will find an overview of the subpackages of the kernel package that are defined in this chapter. The dependency arrows indicate inheritance between packages, except for the arrows marked `<<imports>>`, they indicate an import of a package (see section 2.11 for definitions of these concepts). The packages are briefly described here below.

**actionExpressions.** A definition of ActionExpression, which is a specification of anything that can happen in a software system. It contains a definition of ActionExpression, and some primitive actions. An ActionExpression is the only concept in the language that has (may have) an effect on the local snapshot of the executing object.

**associations.** A definition of relations between objects. Associations are defined as a directed feature of a class, thus giving the class control (and responsibility) over the association(s) it holds, but not over the associated object(s).

**compoundActions.** A definition of the composition of actions into more complex dynamic structures. It includes GroupAction, ConditionalAction, and LoopAction.

**core.** A definition of the fundamental object-oriented constructs. It includes Object, Classifier, and Local-Snapshot.

**features.** A definition of the distinguishing characteristics of a classifier. It includes StructFeature (structural feature), DynFeature (dynamic feature), and Invariant.

**immutableables.** A definition of fundamental types and values, which are all immutable.

**messaging.** A definition of actions that exchange messages between objects. It includes Signal, SendAction, and ProcessSignalAction.

**modelmanagement.** A definition of packages, inheritance between packages, imports, and parameterized packages.

Note that the well-formedness rules that are described below for each package are not complete, nor, for the sake of the readability of this study, are they all fully formalized. Syntax issues will not be taken into account in this chapter. See section 2.12 (“Some notes on the syntax and OCL used in this definition”), and chapter 3 (“Syntax - the UML diagrams”) for more information on syntax.

## 2.3 COMMON STRUCTURE OF THE PACKAGES

As in the MML, each packages consists of three subpackages: *model*, *instance*, and *semantics*. (See also section 1.3.1 (“The Meta Modelling Language”).) The *model* subpackages describe the concepts that can be used in the language we are defining, i.e. the UML. These concepts will be called ModelElements. The *instance* subpackages describe the ‘real life’ notions that can be modelled or specified using the UML. These notions will be called DomainElements. The *semantics* subpackages relate the ModelElements to DomainElements, thereby giving meaning to the ModelElements. For those familiar with the terms abstract syntax, semantic domain, and semantics: the set of *model* subpackages defines the abstract syntax, the set of *instance* subpackages defines the semantic domain, and the set of *semantics* subpackages defines the semantics of the language.

The position of these subpackages in the OMG 4-layered architecture is complex. Each *model* subpackage is a definition in layer M2 of the language concepts that can be used on layer M1. Each *instance* subpackage is a definition in layer M1 of the ‘real life’ notions in layer M0 that can be described by the language (in layer M1). Each *semantics* subpackage builds a bridge between the definitions in a *model* and an *instance* subpackage. It can therefore not be placed in one layer, by definition it crosses the boundary. Figure 2-2 depicts the relation between the common structure of the packages and the OMG 4 layered architecture.

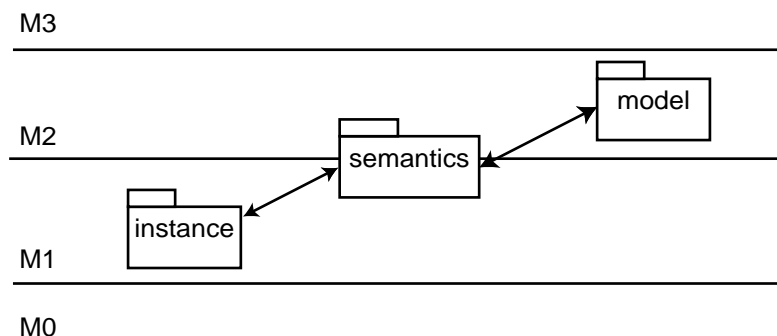


figure 2-2 Relation between OMG layers and common structure of packages

## 2.4 THE CORE PACKAGE

The *core* package describes the fundamental concepts in the language. All other packages will build on the concepts defined here.

Note that we use a special syntax for two directed associations for which a number of constraints hold. The diagram shown in figure 2-3 has exactly the same meaning as the diagram in figure 2-4 combined with the following rules:

```
context Class X inv:
roleY.roleX->includes( self )
context Class Y inv:
roleX.roleY->includes( self )
```

### 2.4.1 The core.model.concepts package

The *core.model.concepts* package is presented in figure 2-5. It is different from MML in two aspects: there is no class Generalisation, because we feel that it may be convenient but not necessary, and the metaclass Class has been replaced by the metaclass Role. Role is a more general concept, which can very well be used for role modeling as defined by Trygve Reenskaug in [Reenskaug96], or for defining the features of agents. Class will be defined as a submetaclass of Role.

#### 2.4.1.1 Definitions

**ModelElement:** a part of a specification or model written in the UML.

**Container:** a ModelElement that is able to contain other ModelElements.

**Generalisable:** a ModelElement from which other ModelElements can be made. It captures the notion of extensibility which is needed to build the meta model from this core package.

**Classifier:** a ModelElement that defines a set of definitions (or rules) according to which Values (see section 2.4.2) can be classified. This follows the dictionary definition of the word ‘to classify’, which says to group or order items into collections. It can contain other ModelElements and it can be extended. Synonym: Type.

**Definition:** a definition of an aspect of a Value. Synonyms: Rule, Property, Feature.

**ImmutableType:** a definition of Values that are immutable.

**Role:** a Classifier, therefore a set of rules, including: ‘an instance of this type is mutable’, and ‘an instance of this type may have references to other instances’.



figure 2-3 **One line representing two unary associations**

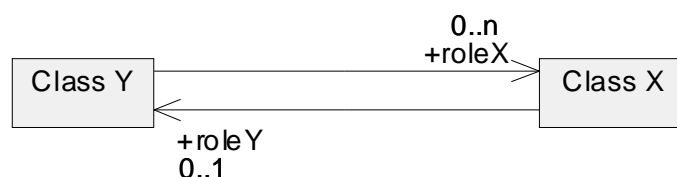


figure 2-4 **Two arrows representing two unary associations**

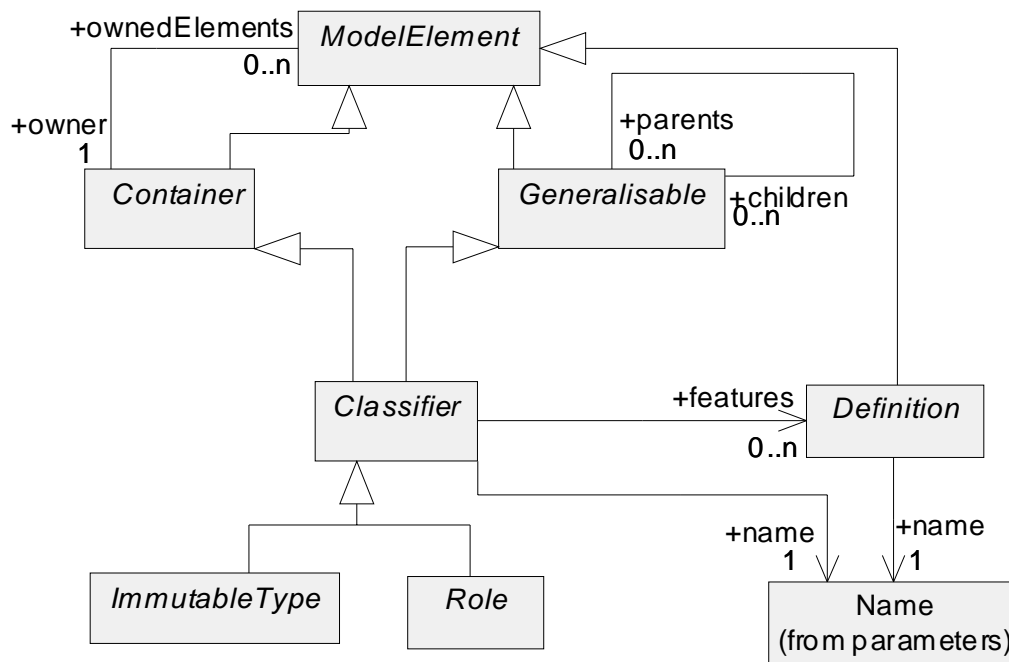


figure 2-5 The core.model.concepts package

Name: a reference. (Needs to be instantiated in order to pick one UML variant. Usually it takes the form of a string type.) Synonym: Variable.

#### 2.4.1.2 Well-formedness Rules

- [1] No two elements belonging to a container can have the same name.
- [2] Circular inheritance is not allowed.
- [3] The parents of a generalisable Element must be of the same type.
- [4] A generalisable element must conform to its parents.

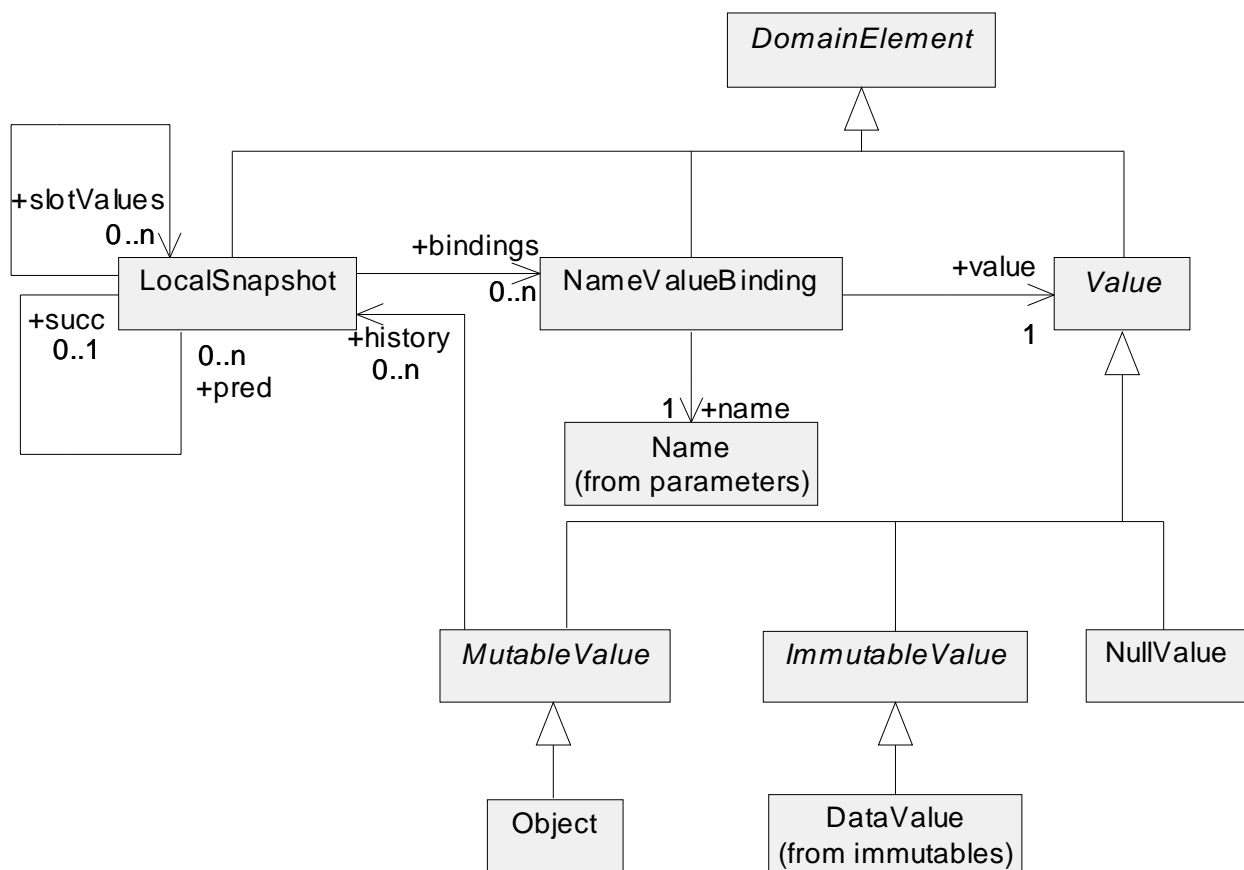
All of these have been formalized in [Clark2000]. Because of the changes made to the core.model.concepts package, they should be reformulated.

### 2.4.2 The core.instance.concepts package

The contents of the *core.instance.concepts* package is shown in figure 2-6. It introduces the fundamental domain elements, the things we can reason about using our UML. The most important notion is that of Value, of which there are mutable and immutable versions. The most obvious subclass of MutableValue is Object, but there may be others, e.g. Component, or Agent could in an extension of this kernel be defined as another subclass of MutableValue. The most obvious subclass of an ImmutableValue is DataValue, but there could be an extension in which an immutable structured type, e.g. a C language struct, or an SQL tuple, is defined.

Mutable values may contain references to other values. These references are bindings at a certain point in time of a name (the reference) to a value, which is represented by the metaclass NameValueBinding. Because the bindings may change from moment to moment, the history of these changes are captured in a list of LocalSnapshots held by the MutableValue. Each LocalSnapshot holds all bindings of one MutableValue at a certain point in time. This allows us to speak of ‘changes in time’ to a certain object or component.

Note that the concept named LocalSnapshot in our approach is different from the concept called snapshot in the MML, but it is like to the definition of the concept given in the Action Semantics. A LocalSnapshot is

figure 2-6 **The core.instance.concepts package**

purely local to the object, it is not an instance of a complete package (as it is in MML). A `LocalSnapshot` contains only the names known directly to its owner object, e.g. all its attributes and associations. It does not contain any names known indirectly to the object, e.g. attributes of associated objects.

As in the Action Semantics time is considered to be relative to the `MutableValue`, i.e. each `MutableValue` has its own notion of time. Time is considered to pass in discrete steps, each step being represented by another snapshot. The real time between the snapshots is not relevant in this kernel. An extension can be built in which the real time between local snapshots is a prominent aspect. (See section 4.3 (“Suggestions for other packages”).) This view on time is in accordance with Einstein’s relativity theory. Every space-continuum, in our case every mutable value, has its own time. Furthermore, every mutable value has its own specific view of the total universe: it sees only the values that are associated with it.

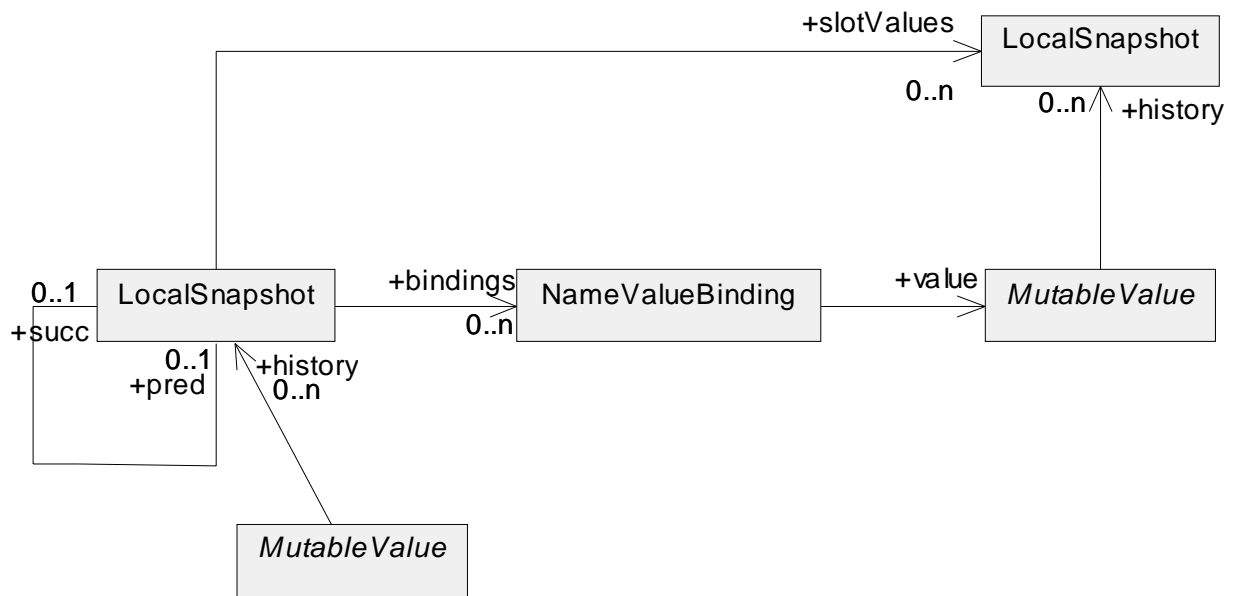
Figure 2-7 shows how the ‘time-reality’ of a mutable value connects to the time-reality of another mutable value. To make the diagram more clear, the classes `MutableValue` and `LocalSnapshot` occur twice. The diagram shows that for every mutable value (call it A) that is in a slot of a mutable value (call it O), there may be a link between the local snapshots of both. This way we can determine the value of the slots of A from the snapshot of O. A minimal synchronisation between the two linked mutable values occurs.

### 2.4.2.1 Definitions

**DomainElement:** an element that can be reasoned about (specified, modelled) using the UML.

**LocalSnapshot:** a collection of `NameValueBinding`s for a certain `MutableValue` at a certain point in time.

**NameValueBinding:** a combination of a `Name` (reference) and a `Value`.

figure 2-7 **Relations between local snapshots**

**Value:** a DomainElement that represents one of the elements of the world that can be modeled using the UML.

**MutableValue:** a Value that has slots of which the value may change.

**ImmutableValue:** a Value that can not change, or be changed.

**NullValue:** a Value that represents void (null, nul, nil).

**Object:** a MutableValue that has a state, represented in a LocalSnapshot, and is able to change that state, thereby creating a new LocalSnapshot.

#### 2.4.2.2 Well-formedness Rules

[1] A name value binding contains its name and value.

```
context NameValueBinding inv:
ownedElements->includes( name ) and ownedElements->includes( value )
```

[2] A mutable value contains its snapshots.

```
context MutableValue inv:
ownedElements->includesAll( history )
```

[3] The history of an object is ordered. The first element does not have a predecessor, the last does not have a successor.

```
context MutableValue
inv: history->oclIsTypeOf( Sequence( LocalSnapshot ) )
inv: history->last().succ->size = 0
inv: history->first().pre->size = 0
```

[4] The slotValues of a LocalSnapshot of a MutableValue may not contain the LocalSnapshot self.

```
context MutableValue inv:
history->collect( slotValues )->intersection( history )->isEmpty
```

#### 2.4.2.3 Additional Operations

[1] *All predecessors* is the collection of all snapshots before a snapshot. *All successors* is the collection of all snapshots after a snapshot.

```
context LocalSnapshot
```

```

def: Let allPredecessors : Set(LocalSnapshot) =
  if pred->notEmpty then
    pred->union(pred.allPredecessors)
  else
    Set {}
  endif
def: Let allSuccessors : Set(LocalSnapshot) =
  if succ->notEmpty then
    succ->union(succ.allSuccessors)
  else
    Set {}
  endif

```

[2] The additional operation *slots* results in all name-type combinations in the last snapshot of the history of a mutable value.

```

context MutableValue
def: Let slots() : Set(NameTypeBindings) =
  history->last()->collect( bindings )

```

### 2.4.3 The core.semantics package

The *core.semantics* package is depicted in figure 2-8. It builds the connection between the concepts Classifier and Value, Role and MutableValue, DataType and DataValue. Note that a MutableValue may have more than one Role. This enables us to model agents, and work with the role-modelling ideas from the OORAM method [Reenskaug96].

Using the type-instance dichotomy in a strict manner, leaves us with the problem where the concept 'class' belongs. A class can execute actions, e.g. create an object, execute class operations, but it also contains the rules by which every instance of that class must be structured and behave. This is a concept that resides on two layers: the M1 and M0 layers. In the definition of the core.semantics package this fact is expressed by a multiple inheritance from both Role and MutableValue.

#### 2.4.3.1 Definitions

Class: a MutableValue that represents a Role, or a Role that is itself a MutableValue.

#### 2.4.3.2 Well-formedness rules

[1] The name of a NameValueBinding must be equal to the name of the corresponding NameTypeBinding, and the value of a NameValueBinding must conform to the type of the corresponding NameTypeBinding.

```

context NameValueBinding
inv: name = model.name
inv: value.model.conformsTo(model.resultType)

```

[2] The null value is an instance of every classifier, and every Classifier has a null value as instance.

```

context Value inv:
self = NullValue implies value.model.conformsTo(OclAny)
context Classifier inv:
instance->includes( NullValue )

```

#### 2.4.3.3 Additional Operations

The additional operation conformsTo(g: Generalisable) has been formalized in [Clark2000]. It defines what it means for a Classifier to conform to its parents.

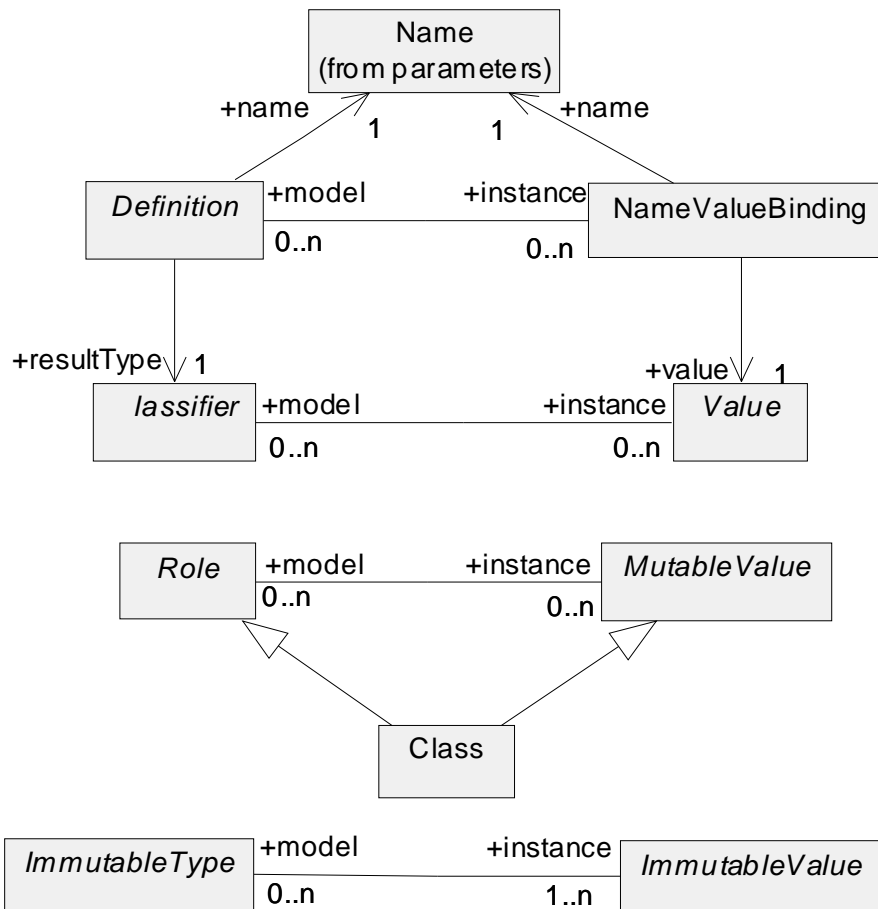


figure 2-8 The core.semantics package

## 2.5 THE FEATURES PACKAGE

The *features* package is the second subpackage of the kernel to be defined. It is a specialisation of the *core* subpackage. It defines the characteristics of a Classifier. These can be divided into (1) structural features, the traditional attributes and associations, (2) dynamical features, the traditional operations and methods, and (3) invariants. This is conform the commonly accepted view that objects have a responsibility to (1) know things, to (2) perform services, and to (3) maintain their own integrity. A fourth type of feature is a parameter definition, which is used to define a parameterized classifier.

Note that every feature has three aspects: it must be defined, it can be called or referenced, and it can be executed or evaluated. The distinction between these three aspects must be clearly made, in order to understand the semantics of features. This package only defines the definitions of features, therefore it contains only a model subpackage, and no instance or semantics subpackage.

### 2.5.1 The features.model.concepts package

The *features.model.concepts* package is depicted in figure 2-9. It defines the four types of features: structural, invariant, dynamic, and formal parameters. All four inherit from Definition, therefore they have a name, and a selfType (a Classifier). A StructFeatureDef and a ParamDef may have an initial value, which is represented

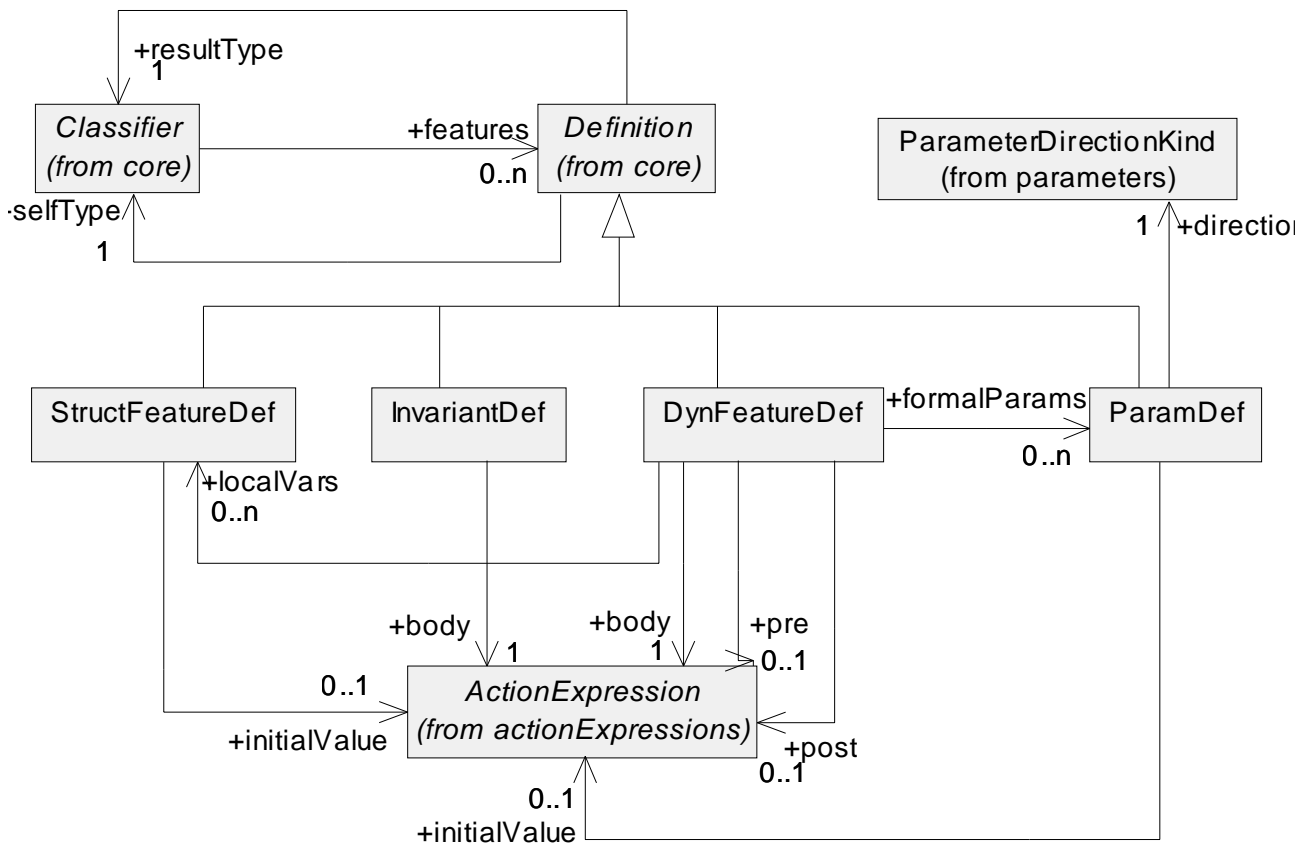


figure 2-9 The features.model.concepts package

by an ActionExpression. The concept ActionExpression is defined in the actionExpressions package. An InvariantDef and a DynFeatureDef both have a body, which is represented by an ActionExpression. An DynFeatureDef may have local variables, which take the same form as structural feature definitions for a classifier. The formal parameters of a DynFeatureDef take the same form as the parameters which are features of a classifier. A dynamic feature is optionally associated with a pre- and postcondition expression.

Note that a Definition inherits the visibility kind indication from ModelElement (defined in the modelmanagement package).

### 2.5.1.1 Definitions

**StructFeatureDef:** a definition of a feature that will not execute and/or change state, it merely is.

**DynFeatureDef:** a definition of a feature that may execute and may change state.

**InvariantDef:** a definition of a rule, mostly based on structural features.

**ParamDef:** a definition with a ParameterDirectionKind, that can be actualized by an actual parameter.

**ParameterDirectionKind:** a definition of an enumeration that denotes if the parameter is used for supplying an argument and/or for returning a value. (Needs to be instantiated in order to pick one UML variant.)

### 2.5.1.2 Well-formedness rules

[1] The type of an Invariant is always Boolean (one of the parameters of the kernel package.)

```

context Invariantdef inv:
resultType = BooleanType
  
```

[2] The resultype of the ActionExpression indicating the body conforms to the type of the InvariantDef.

```
context InvariantDef inv:
body.resultType.conformsTo(self.resultType)
```

[3] The resultype of the ActionExpression indicating the body conforms to the type of the DynFeatureDef.

```
context DynFeatureDef inv:
body.resultType.conformsTo(self.resultType)
```

[4] The resultype of the ActionExpression indicating the initial value conforms to the type of the ParamDef.

```
context ParamDef inv:
initialValue.resultType.conformsTo(self.resultType)
```

[5] The resultype of the ActionExpression indicating the initial value conforms to the type of the StructFeatureDef.

```
context StructFeatureDef inv:
initialValue.resultType.conformsTo(self.resultType)
```

## 2.6 THE IMMUTABLES PACKAGE

The *immutables* package is a subpackage of the kernel that defines a number of generic subtypes of *ImmutableType* and *ImmutableValue*. It is another specialisation of the core package. The generic types that are defined are rather straightforward, in fact one can think of this package as a mere ‘hook’ for users of the language to hang their own types extensions.

### 2.6.1 The *immutables.model.concepts* package

The *immutables.model.concepts* subpackage defines a number of types. It is shown in figure 2-10.

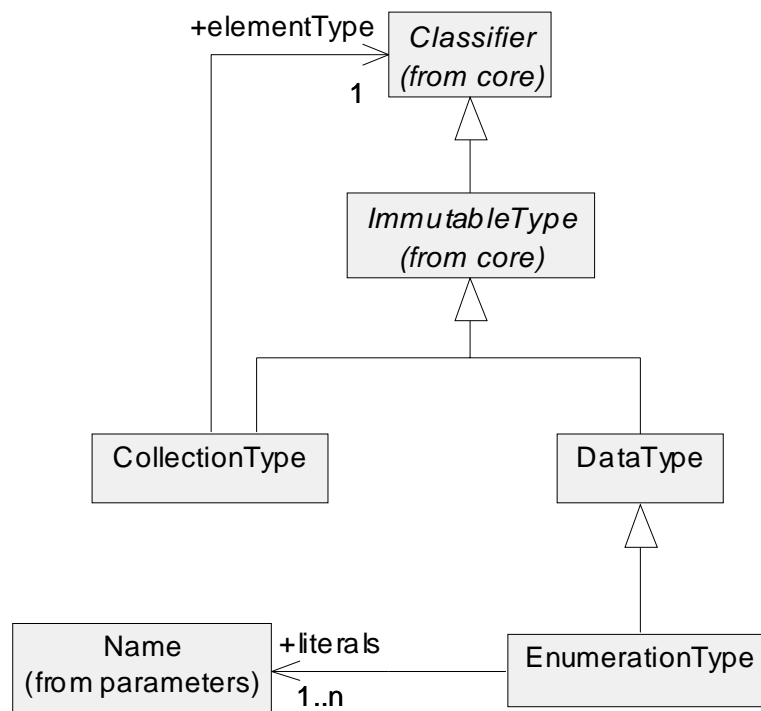


figure 2-10 The *immutables.model.concepts* package

### 2.6.1.1 Definitions

CollectionType: a type that defines values that can hold other values.

DataType: a Classifier, therefore a set of rules, including: 'an instance of this type is immutable', and 'an instance of this type has no references to other instances'.

EnumerationType: a type that defines a limited set of values.

### 2.6.1.2 Well-formedness rules

none

## 2.6.2 The `immutable.instance.concepts` package

The `immutable.instance.concepts` subpackage defines a number of values. It is shown in figure 2-11. It is a mirror image of the `immutable.model.concepts` subpackage, except for the Element meta class. This is needed because a collection value may contain the same value more than once.

### 2.6.2.1 Definitions

CollectionValue: an immutable value that can hold other values.

DataValue: a Value that can not be changed and does not contain any parts.

EnumerationValue: a Value picked from a limited set of values.

Element: an unnamed reference to a value.

Number: serial number determining the order (if any) of elements in the collection.

### 2.6.2.2 Well-formedness rules

[1] Number must be either present for all elements of a collection, or for none.

```
context CollectionValue inv:
elements->forall( nr->size = 0 ) or elements->forall( nr->size = 1 )
```

[2] Number (when present) must be unique within collection.

```
context CollectionValue inv:
elements->isUnique( nr )
```

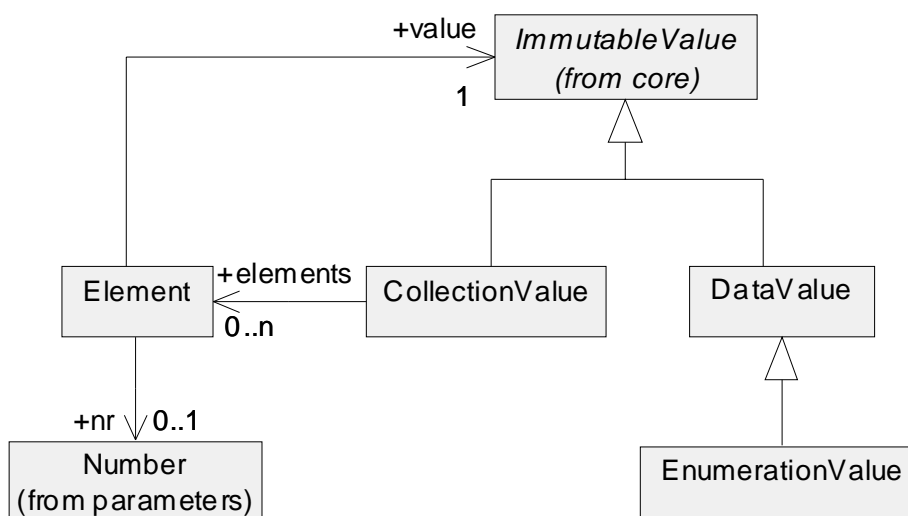


figure 2-11 The `immutable.instance.concepts` package

### 2.6.3 The `immutables.semantics` package

The `immutables.semantics` subpackage defines the relation between the types and the values. It is shown in figure 2-12.

#### 2.6.3.1 Well-formedness rules

[1] An `EnumerationValue` must be equal to one of the literals in the `EnumerationType`.

```
context EnumerationValue inv:
model.literals->contains(self)
```

[2] The type of the value of an element of a collection must be equal to the `elementType` of corresponding collection type.

```
context CollectionValue inv:
elements->forall( value.model = self.model.elementType )
```

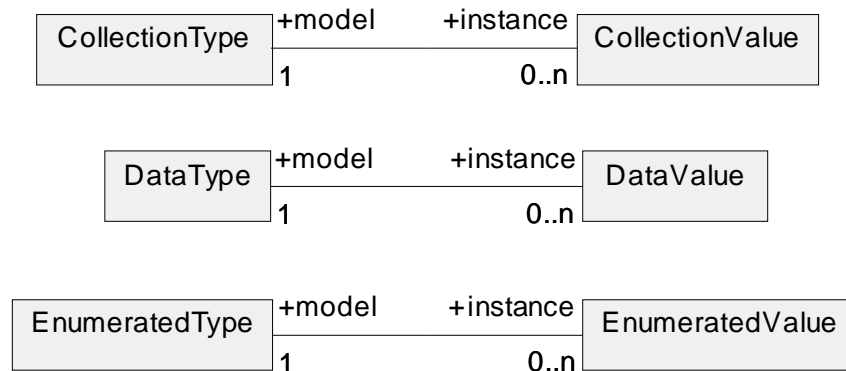


figure 2-12 The `immutables.semantics` package

## 2.7 THE ASSOCIATIONS PACKAGE

The `associations` package is the third subpackage of the kernel to be defined. It is a specialisation of the `features` package, and it imports the `immutables` package. It defines a special type of structural feature known as association. The ‘standard’ association is a one-way directed reference to another mutable value.

### 2.7.1 The `associations.model.concepts` package

The `associations.model.concepts` package is shown in figure 2-13. The `associations` package contains only extra definitions, therefore the instance and semantics subpackage are omitted.

#### 2.7.1.1 Definitions

**UnaryAssociation:** a structural feature that can be used to reference another mutable value, therefore its type can only be a role, or a collection type. (The collection types need to be actualized, e.g. from the package `predefTypes`.)

**MultiplicityKind:** an indication of the number of elements in the collection of the unary association.

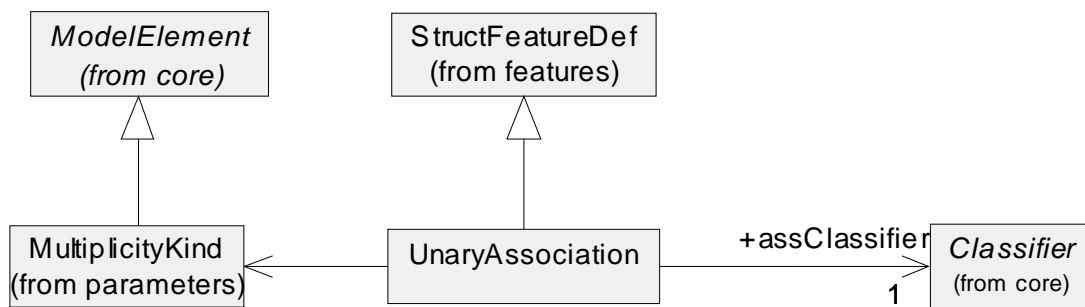


figure 2-13 The associations.model.concepts package

### 2.7.1.2 Well-formedness rules

[1] When the multiplicity is zero or one, the type of a UnaryAssociation is equal to its associated Classifier.

```

context UnaryAssociation inv:
(multiplicity = MultiplicityKind.0..1 or multiplicity = MultiplicityKind.1)
implies self.resultType = assClassifier
  
```

[2] When the multiplicity is zero to many, or one to many, the type of a UnaryAssociation is equal to the predefined type Set, and this Set has as elementType the role of the UnaryAssociation.

```

context UnaryAssociation inv:
(multiplicity = MultiplicityKind.0..* or multiplicity = MultiplicityKind.1..*)
implies self.resultType = Set( assClassifier )
  
```

### 2.7.2 Observations

One would expect the two-way association to be an element in this kernel language also. We might define it to be a restriction on two one-way associations: both should represent each other's inverse. But what does it mean to be each other's inverse? On the model level this is fairly straightforward: the type of the first one-way association must be equal to the selfType of the second, or to a collection which element's types are equal to the selfType of the second one-way association.

On the instance level a two-way association is much more difficult to define. Figure 2-14 shows an instance representation of a two-way one-to-one association. (Note that the syntax used here to denote instances, is not part of the kernel language.) Object X holds a history of snapshots, each of which holds a set of name-value-bindings. One of these name-value-bindings holds as value object Y. In its turn, object Y holds a history of snapshots, each of which holds a set of name-value-bindings, of which one holds as value object X. When we take time into account what do we know, or what do we need to assert? Must every snapshot of X hold object Y? No, the values in a binary association may change, as long as the 'binary' aspect of the association is again re-established. The only thing we can assert is that when object Y is referenced (if only in one snapshot), there

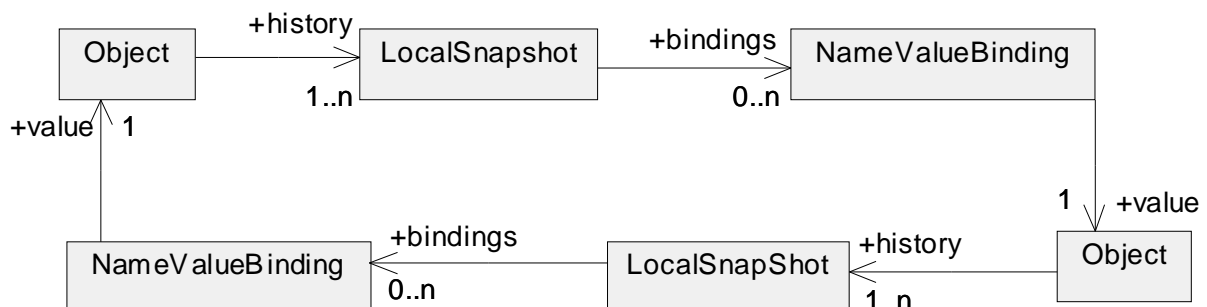


figure 2-14 Instance representation of a two-way association

should be at least one snapshot in Y that holds object X. This assertion is a minimal definition and does not fully grasp the meaning that is normally given to a two-way association.

Another aspect of a two-way association also has to do with time. Whenever a value in a two-way association is changed, it must be changed in a series of steps: one object needs to change its value first, the not longer referenced object needs to dereference the first object, and the newly referenced object needs to set a reference to the first object. Each of the steps may be defined to be primitive, and atomic, but the combination of steps can never be considered atomic at the level of the kernel language here defined.

Because of these difficulties we have decided not to include a two-way association into the kernel package.

---

## 2.8 THE ACTIONEXPRESSIONS PACKAGE

In every expression of the UML language values may be read, or changed. Although it would be convenient to distinguish between expressions that do not change values, and expression that do, this is not possible. Take as example the expression *aap.noot.mies.schapen(gijs).does()* (written as an OCL expression, but without the restriction that states may not be changed), even if the feature *noot* of *aap* does not change values, nor the feature *mies* of *noot*, the feature *schapen(gijs)* may change some values, or the feature *does()* may. Still the example is one expression. In this package the notion of *actionExpression* is defined, which is a formalisation of expressions that may or may not change values.

The *actionExpressions* package defines the fundamental constructs that generate ‘changes in time’. It includes primitive actions as *WriteAction* and *CreateObjectAction*. *ActionExpressions* (may) change the value of the slots of a mutable value, thus creating one or more new snapshots. This means that every action expression execution is associated with a before and an after snapshot.

### 2.8.1 The *actionExpression.model.concepts* package

The *actionExpression.model.concepts* package is shown in figure 2-15. An *actionExpression* in its simplest form is merely a data value expression, e.g.  $4 + 3 * 7$ . The way these expressions are formed and calculated must be defined in the package that actualizes the template parameters for the kernel package. More complex expressions are build by using references to features (feature calls), or by primitive or compound actions. Note that every subtype of *Definition* can be referenced, even *InvariantDefs*. This is an extension of the current UML specification.

#### 2.8.1.1 Definitions

**ActionExpression:** a *ModelElement* with which one can indicate a certain *Value*, or a certain action to be performed.

**FeatureRef:** a reference or call to a *Definition* (see section 2.5.1 (“The *features.model.concepts* package”)).

**PrimitiveAction:** an action that specifies a change of the snapshot of the mutable value that executes the action, the self instance, only.

**DataValueExp:** an expression consists solely of data values and operations on data values.

**CreateObjectAction:** an expression that specifies a change of the snapshot of a class in such a way that the slot ‘new’ contains a new object which is an instance of that class.

**ReadAction:** an expression that specifies a certain value, within the context of a single mutable value.

**WriteAction:** an action that specifies a change of the value of one of the slots of the related mutable value.

**NameTypeSpace:** a set of *Definitions*, coupling *Names* to *Types* (or *Classifiers*).

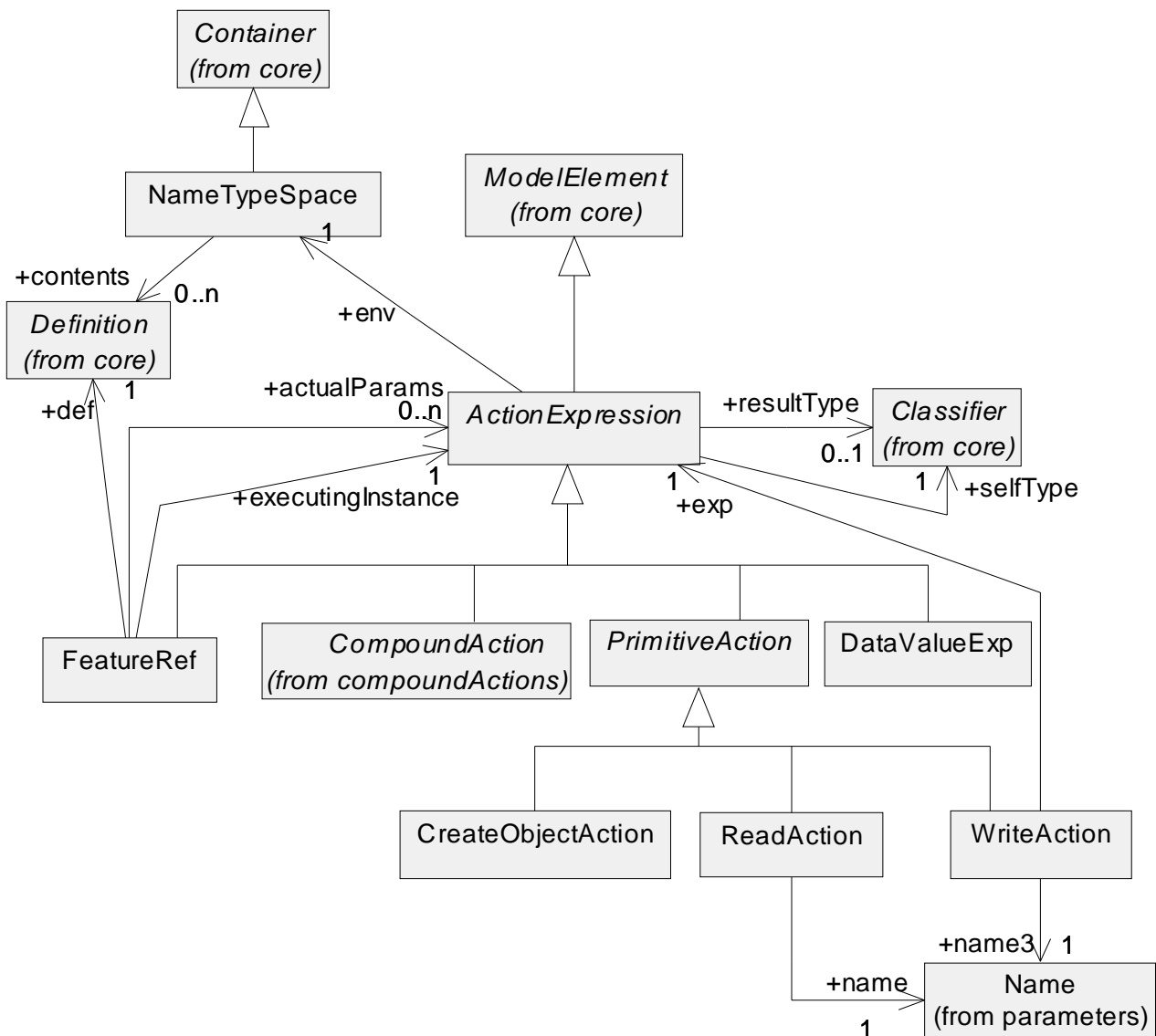


figure 2-15 The actionExpressions.model.concepts package

### 2.8.1.2 Well-formedness rules

[1] The environment of an action expression contains all names from its selfType.

```
context ActionExpression inv:
env.bindings->includesAll( selfType.features )
```

[2] The selfType of a feature reference is equal to the selfType of its definition.

```
context FeatureRef inv:
selfType = def.selfType
```

[3] The environment of the body of a dynamic feature is its own environment plus the name type bindings for the formal parameters and the variable declarations.

```
context DynFeatureDef inv:
body.env->includesAll( formalParams ) and body.env->includesAll( localVar )
```

[4] A feature reference should be a reference to a feature of the selfType of the executing instance (the source).

```
context FeatureRef inv:
  executingInstance.selfType.features->includes( def )
```

[5] As in the Action semantics the CreateObjectAction does not model the execution of the constructor, it only creates an object of the right class with structural features, therefore a CreateObjectAction has no parameters.

```
context uml.actions.model.CreateObjectAction inv:
  params->size = 0
```

## 2.8.2 The actionExpressions.instance.concepts package

The *actionExpressions.instance.concepts* package is depicted in figure 2-16. It defines the concept of ActionExpExec, which is the execution or evaluation of an actionExpression within a certain context. This context is determined by the mutable value that performs the action: the selfInstance. An ActionExpExec has a before and an after snapshot representing the state of the self instance before and after the execution or evaluation of the ActionExpression.

The term execution is normally used for executions that change state, whereas the term evaluation is normally used for executions that do not change state. As explained earlier we can not distinguish between expressions that do and do not change state, therefore we use the term execution to denote both execution and evaluation in the following.

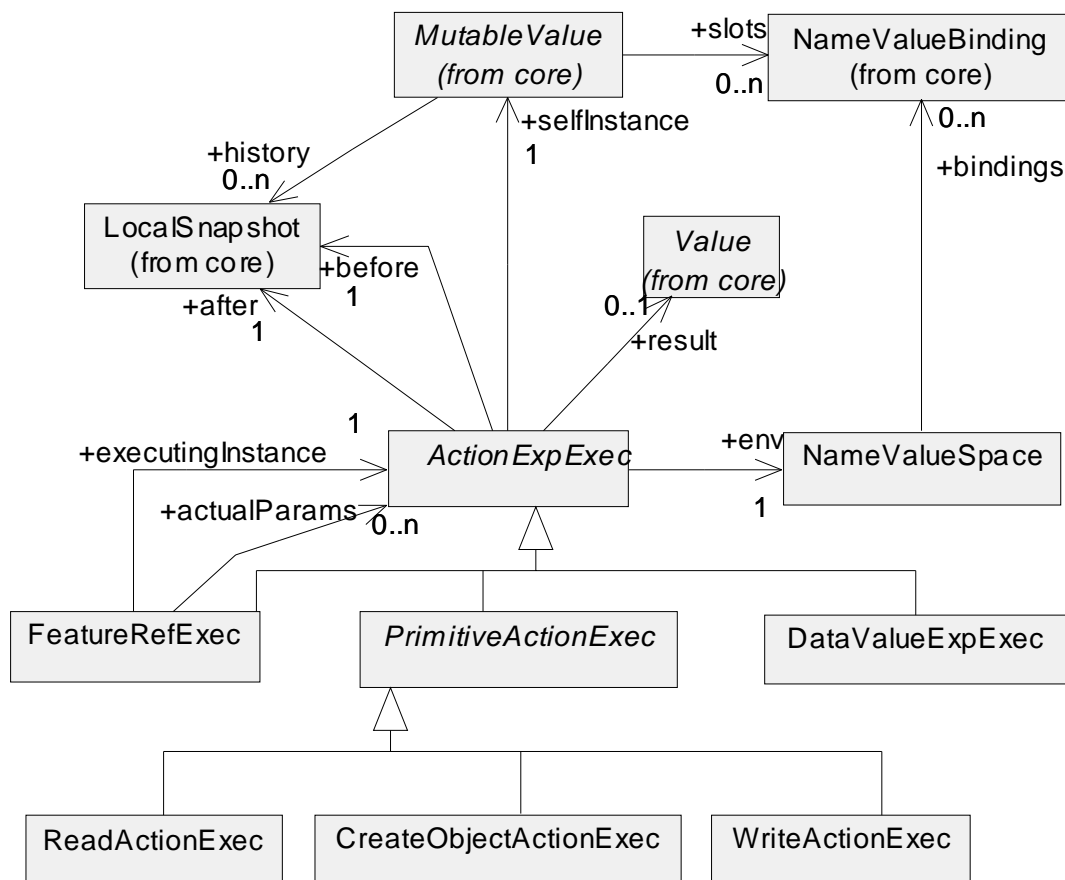


figure 2-16 The actionExpressions.instance.concepts package

### 2.8.2.1 Definitions

ActionExpExec: an executable unit that may generate changes in snapshots of related mutable values.

NameValueSpace: a set of NameValueBindings that are used to evaluate an actionExpression. Synonym: Context, Environment.

FeatureRefExec: the execution or evaluation of a feature.

PrimitiveActionExec: the execution of a primitive action.

DataValueExpExec: an expression that results in a data value.

ReadActionExec: an ActionExpExec that evaluates the NameValueBindings in the NameValueSpace, and results in the Value that is bound to a given Name.

CreateObjectActionExec: the execution of an expression that creates (makes known to the environment) a mutable value.

WriteActionExec: the execution of an expression that changes the value of a slot of a mutable value.

### 2.8.2.2 Well-formedness rules

[1] A FeatureRefExec can not be used as its own actual parameter.

```
context FeatureRefExec inv:
  not actualParams->includes( self )
```

[2] The after snapshot comes always later in the history than the before snapshot.

```
context ActionExpExec inv:
  after.allPredecessors->includes(before)
```

[3] When a value of slot changes a new snapshot is created.

```
context WriteActionExec inv:
  after <> before
```

[4] A Primitive action can only change the executing object.

```
context PrimitiveActionExec inv:
  selfInstance.history->includes(after) and
  selfInstance.history->includes(before)
```

[5] The selfInstance of a CreateObjectAction is a Class.

```
context uml.actions.instance.CreateObjectActionExec inv:
  selfInstance.ocIsKindOf(Class)
```

[6] When a CreateObjectAction is executed, a new object is available in the special slot 'new' of its corresponding Class.

```
context CreateObjectAction
inv: selfInstance.history->last->bindings->contains( x: NameValueBinding |
  x.name = 'new' and x.value.ocIsKindOf( MutableValue ) )
```

[7] The selfInstance of a WriteAction is a MutableValue.

```
context uml.actions.instance.WriteActionExec inv:
  selfInstance.ocIsKindOf(MutableValue)
```

[8] The environment of an ActionExpExec is based on the before snapshot.

```
context ActionExpression inv:
  env->includesAll( before.bindings )
```

### 2.8.3 The actionExpressions.semantics package

The *actionExpressions.semantics* package is depicted in figure 2-17. It shows the relationships between the concepts in the model subpackage and the concepts in the instance subpackage, which are all straightforward.

### 2.8.3.1 Well-formedness rules

[1] The result of an ActionExpExec must conform to the result type of the corresponding actionExpression.

```
context ActionExpExec inv:
  result.conformsTo( model.resultType )
```

[2] The ActionExpExecs of the actual parameters of a FeatureRefExec must correspond with the ActionExpressions for the actual parameters of a FeatureRef.

```
context FeatureRefExec inv:
  actualParams->forall( exec: ActionExpExec |
    self.model->forall( actualParams->exists( exp: ActionExpression |
      exp.instance->includes( exec ) ) )
```

[3] All names in the environment of an execution must be known within the environment of the corresponding expression.

```
context ActionExpExec
  inv: env.bindings->collect(name) = mode.env.bindings->collect(name)
```

[4] The precondition of a dynamic feature definition must hold in the before snapshot of the corresponding feature reference execution.

*to be formalized*

[5] The postcondition of a dynamic feature definition must hold in the after snapshot of the corresponding feature reference execution.

*to be formalized*



figure 2-17 The actionExpressions.semantics package

[6] The after snapshot of a WriteActionExec holds a binding that binds the name of the corresponding WriteAction to its value.

*to be formalized*

[7] A WriteActionExec can only change the value of a slot of its selfInstance.

*to be formalized*

Many more well-formedness rules can be conceived. We do not claim this to be an exhaustive list.

---

## 2.8.4 Observations

The actions package defines two primitive actions. Whenever additional primitive actions are needed, they can be added as subtypes of PrimitiveAction. The messaging package defined in section 2.10 does exactly this.

---

## 2.9 THE COMPOUND ACTION PACKAGE

The *compoundAction* package defines three basic types of compound actions, which is an action that is build from a number of, possibly primitive, actions. In “Making UML Activity Diagrams Object-Oriented” [Kleppe2000a] we defined four basic types of compound action, that form the core building blocks with which all other actions can be defined: SequentialAction, ParallelAction, AlternativeAction (or choice) and IterativeAction (or loop).

In [AS2000] the types GroupAction, LoopAction and ConditionalAction cover the same area. Since the action semantics considers all actions inherently parallel unless synchronized explicitly, sequential and parallel actions are not distinguished separately. This is an abstraction of [Kleppe2000a] which make sense in the context of this kernel, and is therefore used in the following.

Note that the following definitions are based on the idea that all subactions of a compound action are executed by the same object. Without the notion of messaging this is obviously the case, with the notion of messaging described in section 2.10 this still is the case, as outgoing messages, like feature calls, are considered to be an action executed by the self instance. The response of that other object to a feature call, is another action executed by that other object, and not considered to be part of the compound action, although we may (as shown in section 4.1) delay the further execution of the compound action until the execution of the called feature has finished.

---

### 2.9.1 The compoundAction.model.concepts package

The *compoundAction.model.concepts* package is shown in figure 2-18. It defines a conditional action that contains a subaction and a test, a loop action that also contains a subaction and a test, and a group action that contains a number of subactions. Compound actions are defined as subtypes of ActionExpression, therefore all three inherit an environment (the 'env' association from ActionExpression with NameTypeSpace) that may contain any local variables.

#### 2.9.1.1 Definitions

**CompoundAction:** an action that is an aggregate of (sub)actions.

**GroupAction:** an action that consists of a number of subactions that may be executed in any order.

**LoopAction:** an action that consists of a test and a subaction, which specifies that the subaction has to be executed a number of times until the test results in false.

**ConditionalAction:** an action that consists of a test and a subaction, which specifies that the subaction has to be executed only if the test results in true.

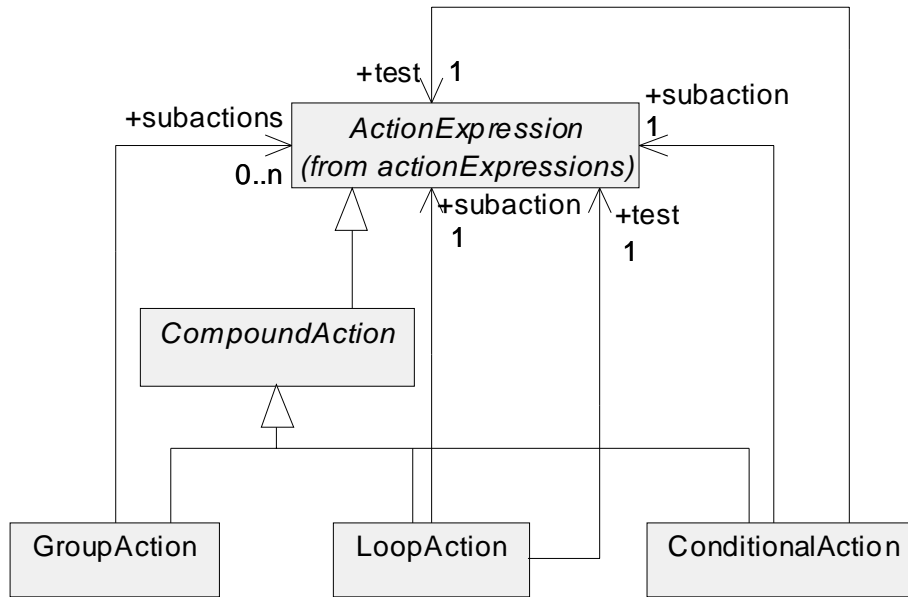


figure 2-18 The compoundActions.model.concepts package

### 2.9.1.2 Well-formedness rules

[1] The test of a ConditionalAction must have a boolean result.

```
context ConditionalAction inv:
test.resulttype.isTypeOf(Boolean)
```

[2] The test of a LoopAction must have a boolean result.

```
context LoopAction inv:
test.resulttype.isTypeOf(Boolean)
```

## 2.9.2 The compoundAction.instance.concepts package

The *compoundAction.instance.concepts* package is shown in figure 2-19. It defines the three counterparts of the compound actions in the model subpackage: GroupActionExec, LoopActionExec, and ConditionalActionExec. The GroupActionExec and the ConditionalActionExec are defined as expected, but the definition of LoopActionExec needs some explanation.

A LoopActionExec is considered to be a compound action that sequentially combines the execution of the same test and action combination time and again, but with different bindings. As the ConditionalActionExec is defined as exactly such a test and action combination, a LoopActionExec is modelled as a series of ConditionalActionExecs.

### 2.9.2.1 Definitions

CompoundActionExec: an execution of a compound action.

GroupActionExec: an execution of a group action.

LoopActionExec: an execution of a loop action.

ConditionalActionExec: an execution of a conditional action.

### 2.9.2.2 Well-formedness rules

On ConditionalActionExec:

- [1] The environment of the ActionExpExec representing the test of a ConditionalActionExec is the same as the binding in the before snapshot of the ConditionalActionExec.

```
context ConditionalActionExec inv:
test.env = self.env
```

- [2] The subaction ActionExpExec of a ConditionalActionExec is only present when the test results in true.

```
context ConditionalActionExec
inv: test.value = true implies sub.size = 1
inv: test.value = false implies sub.size = 0
```

- [3] The after snapshot of a ConditionalActionExec is the after snapshot of its subaction, if present.

```
context ConditionalActionExec inv:
sub.size = 1 implies self.after = sub.after
```

- [4] If the subaction ActionExec is not present, then the after snapshot of the ConditionalActionExec is equal to the before snapshot.

```
context ConditionalActionExec inv:
sub.size = 0 implies after = before
```

- [5] The before snapshot of the subaction is identical to the before snapshot of the ConditionalActionExec.

```
context ConditionalActionExec inv:
sub.size = 1 implies sub.before = before
```

#### On LoopActionExec:

- [6] The ActionExecs belonging to a LoopActionExec, i.e. the execList, are ordered.

```
context loopActionExec inv:
execList.isOclTypeOf( Sequence( ConditionalActionExec ) )
```

- [7] Any ActionExec in the execList of a LoopActionExec for which the test results in true, is not the last.

```
context LoopActionExec inv:
execList->forall( e | e.test.value = true implies not e = execList->last )
```

- [8] The test of the last ActionExec in the execList of a LoopActionExec always results in false.

```
context LoopActionExec inv:
execList->last.test.value = false
```

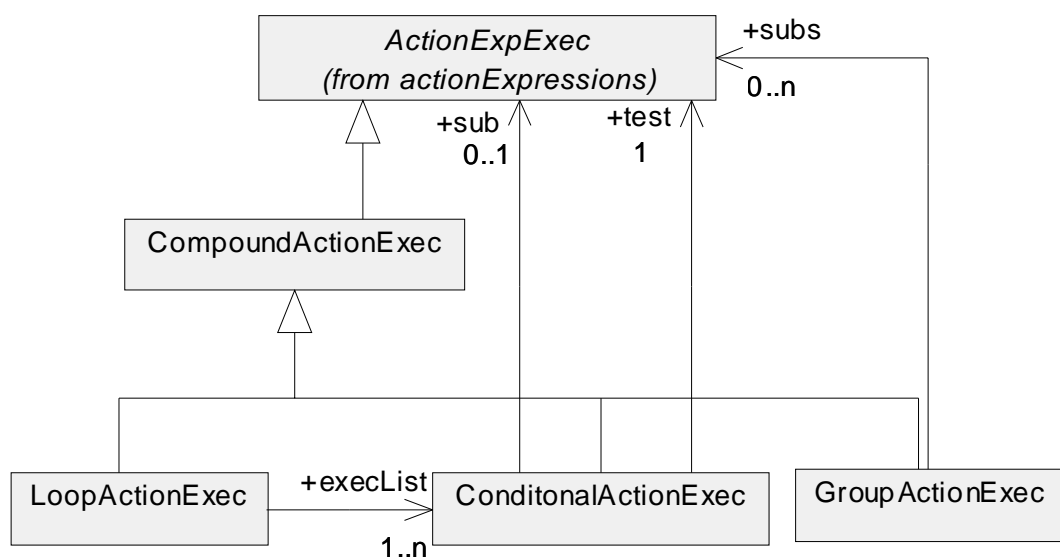


figure 2-19 **The compoundAction.instance.concepts package**

[9] The before snapshot of any element in the `execList` of a `LoopActionExec` comes later than the after snapshot of its predecessor, thus the binding of the test changes during the loop execution.

```
context LoopActionExec inv:
Sequence{1..execList->size}->forall( i: Integer |
    execList->at(i+1).before.allPredecessors->includes( execList->at(i).after ) )
```

[10] The before snapshot of the `LoopActionExec` is identical to the before snapshot of the first `ConditionalActionExec` in the `execList`.

```
context LoopActionExec inv:
before = execList->first.before
```

[11] The after snapshot of the `LoopActionExec` is identical to the after snapshot of the last `ConditionalActionExec` in the `execList`.

```
context LoopActionExec inv:
after = execList->last.after
```

On `GroupActionExec`:

[12] The subactions of a `GroupActionExec` must all be executed sequentially. (Equal to the run-to-completion semantics in the Action Semantics document.) But we cannot assume any ordering according to the specification of the `GroupAction`. The latter will be dealt with in the `compoundAction.semantics` package.

```
context GroupActionExec inv:
subs->oclIsTypeOf( Sequence( ActionExpExec ) )
```

[13] The before snapshot of the `GroupActionExec` is identical to the before snapshot of the first subaction execution.

```
context GroupActionExec inv:
before = subs->first.before
```

[14] The after snapshot of the `GroupActionExec` is identical to the after snapshot of the last subaction execution.

```
context GroupActionExec inv:
after = subs->last.after
```

## 2.9.3 The `compoundAction.semantics` package

The `compoundAction.semantics` package is shown in figure 2-20.

### 2.9.3.1 Well-formedness rules

[1] The test of a `ConditionalActionExec` is linked to the test of the corresponding `ConditionalAction`.

```
context ConditionalActionExec inv:
self.test.model = model.test
```

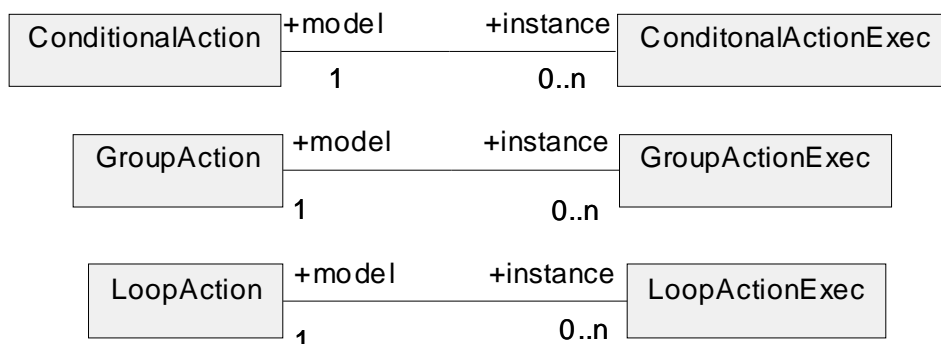


figure 2-20 The `compoundAction.semantics` package

[2] The subaction of a ConditionalActionExec is linked to the action of the corresponding ConditionalAction.

```
context ConditionalActionExec inv:
self.sub.model = model.sub
```

[3] The test of every ConditionalActionExec in the execList of a LoopActionExec is linked to the same ActionExpression, which is the test of the LoopAction.

```
context LoopActionExec inv:
execList->forall( e : ConditionalActionExec |
  e.test.model = model.test )
```

[4] The subaction of every ConditionalActionExec in the execList of a LoopActionExec is linked to the same ActionExpression, which is the subaction of the LoopAction.

```
context LoopActionExec inv:
execList->forall( e : ConditionalActionExec |
  e.sub.model = model.sub )
```

[5] Each subaction of the GroupAction has a corresponding ActionExec in the GroupAction's GroupActionExec, and vice versa, in other words, the subactions that are defined in a GroupAction will all be executed.

```
context GroupActionExec inv:
subs->forall( exec : ActionExpExec |
  self.model->forall( subs->exist( exp: ActionExpression | exp = exec )
```

[6] If the subactions of a GroupAction are ordered, then so are the subaction executions of its GroupActionExec.

```
context GroupAction inv:
subs.oclIsTypeOf( Sequence( ActionExpression ) ) implies
instance->forall( i: GroupActionExec |
  i.subs.oclIsTypeOf( Sequence( ActionExpression ) )
```

## 2.9.4 Observations

Instead of reusing ConditionalActionExec for the definition of LoopActionExec, we might introduce a separate class LoopStep in the instance package, but its definition would be similar to ConditionalActionExec. The consequence of our choice is that the association between ConditionalAction and ConditionalActionExec must be optional on the side of ConditionalAction. So far, we do not foresee any difficulties with this.

Note that the given well-formedness rules define the LoopAction to have a 'while-do' semantics. It is equally feasible to give a set of well-formedness rules that define a 'do-while', or 'do-until' semantics. A number of packages defining different loop action semantics could be part of the complete semantics of UML.

The conditional action in the Action Semantics is a somewhat broader concept, it also incorporates the if-then-else and case statements. The conditional action defined in this section only defines an if-else statement. However, together with the group action concept, we can build the if-then-else and case statement by grouping together a number of conditional actions of which the tests are related. By constraining the before snapshots of all conditional actions in the group to be equal, the definitions of the if-then-else and case statement are given.

---

## 2.10 THE MESSAGING PACKAGE

To define messaging between objects two aspects are important: what happens to the objects involved, and the transport mechanism used. To allow the UML to be used for modelling systems with varying transport mechanisms, these two aspects should be defined separately in the definition of the semantics of UML. This means that there should be different packages for different transport mechanisms, and a separate package that defines how the objects are affected.

The *messaging* package described in this section only defines how the objects are affected. For this purpose the object is extended with an input and an output queue, both of which may contain signals. Furthermore, two new subclasses of PrimitiveAction are introduced: SendAction and ProcessSignalAction. A SendAction sim-

ply puts a signal in the output queue of the self instance, a `ProcessSignalAction` takes a signal out of the input queue.

Note that a signal is only the wrapping of the real message, it may contain any action expression that represents this message, e.g. it could be a data value expression or a feature reference. A signal is like the bottle or envelope in which a letter is conveyed. Processing a signal is unwrapping the message, and determining what needs to be done.

Building on this very basic notion of messaging, transport mechanisms may be defined in other packages that take a signal from an output queue and put it in an input queue. Many different, simple or complex, transport mechanisms may be defined in this manner, ranging from a simple synchronous operation call communication, to asynchronous, time-consuming, or faulty communications.

## 2.10.1 The `messaging.model.concepts` package

The `messaging.model.concepts` package is shown in figure 2-21.

### 2.10.1.1 Definitions

**Signal:** the packaging of information that has as source one mutable value, and as targets one or more mutable values.

**SendAction:** an action that specifies a change to its self value so that a signal is placed in the output queue.

**ProcessSignalAction:** an action that specifies a change to its self value so that a signal is removed from the input queue.

### 2.10.1.2 Well-formedness rules

[1] The source and targets of a signal must be a mutable value.

```
context Signal
inv: source.resultType.oclIsTypeOf( Role )
inv: targets->forall( resultType.oclIsTypeOf( Role ) )
```

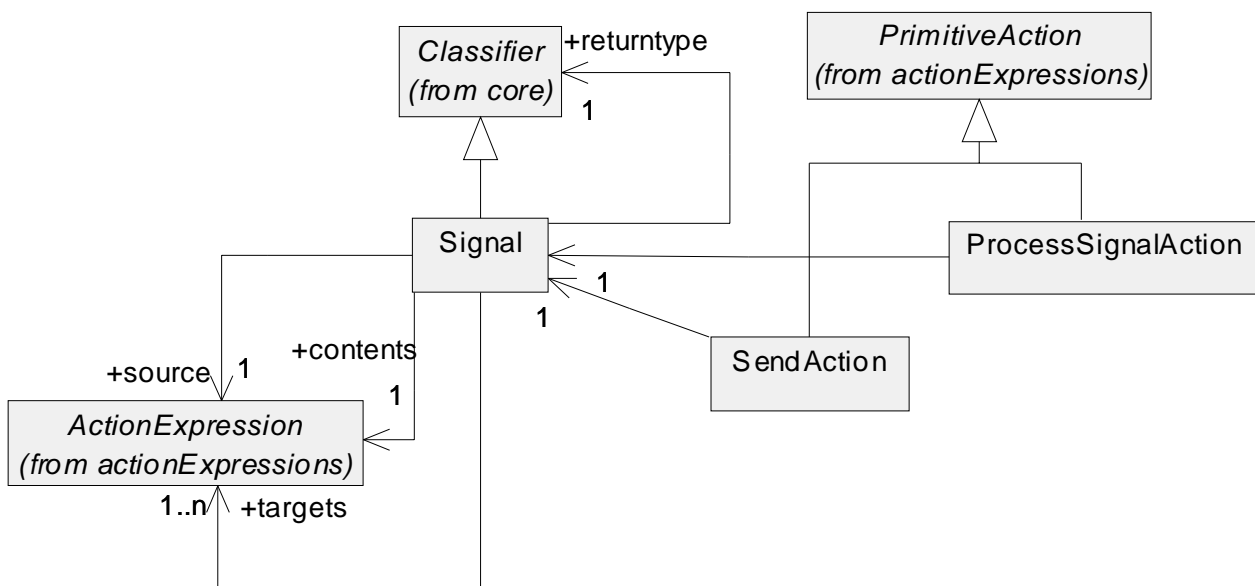


figure 2-21 The `messaging.model.concepts` package

## 2.10.2 The messaging.instance.concepts package

The *messaging.instance.concepts* package is shown in figure 2-22. There are few well-formedness rules, because this package needs to be a very general one and every rule restricts the concepts further. E.g. we are tempted to add the well-formedness rule specifying that a signal instance must be at some point in time in its source's output queue, or that at some point in time it must be in its target's input queue. Both are notions that must be defined by the transport mechanism, and both are notions that may not be true for every transport mechanism, therefore we must resist this temptation.

Note that the contents of a *SignalInstance* is still a model element, not a domain element. The *ActionExpression* that represents the content is not executed.

### 2.10.2.1 Definitions

**Bus:** a transport mechanism that transports signals from output queue to input queues.

**SignalInstance:** an occurrence of a signal.

**SendSignalExec:** the execution of the sending of a signal, i.e. putting a signal with its content in the output queue of the 'self' mutable value.

**ProcessSignalExec:** the execution of the receipt of a signal, i.e. taking a signal with its content from the input queue of the 'self' mutable value.

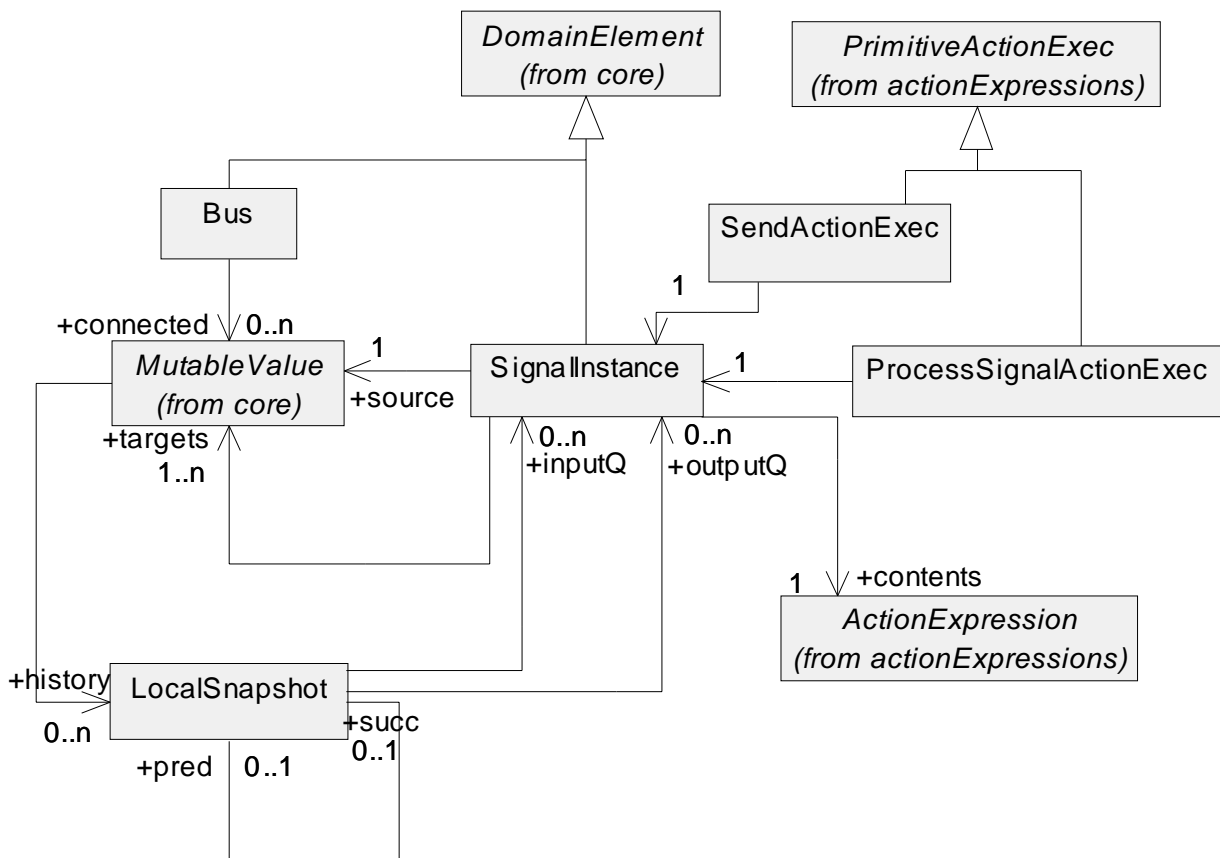


figure 2-22 The messaging.instance.concepts package

### 2.10.2.2 Well-formedness rules

- [1] Executing a reference to a dynamic feature will create a signal instance, of which the contents is equal to the feature reference, and the source is equal to the self instance, and the target(s) are equal to the executing instance link of the dynamic feature reference.

```
context FeatureRefExec inv:
def.oclIsTypeOf( DynFeatureDef ) implies
  selfInstance.after.outputQ->exists( s: SignalInstance |
    s.contents = self.model and
    s.source   = selfInstance and
    s.targets  = self.model.executingInstance )
```

- [2] A SendActionExec may only add a signal to the output queue of the object that is executing this action.

*This follows from rule[4] of the actionExpression.instance.concepts package*

- [3] A ProcessSignalAction may only remove a signal from the input queue of the object that is executing this action.

*This follows from rule[4] of the actionExpression.instance.concepts package*

- [4] The target of the signal processed by a ProcessSignalAction must be the selfInstance of that ActionExec.

```
context ProcessSignalAction inv:
signal.target = selfInstance
```

- [5] All targets of all signal instances in the output queues of a mutable value, must be equal to one of the values connected to the bus.

```
context Bus inv:
connected->forAll( mv: MutableValue |
  mv.history.outputQ->forAll( s: Signal |
    self.connected->includesAll( s.targets )))
```

## 2.10.3 The messaging.semantics package

The *messaging.semantics* package is shown in figure 2-23. It is very straightforward. It maps Signal to SignalInstance, and the actions to their 'exec' counterparts.

### 2.10.3.1 Well-formedness rules

- [1] The target of a SignalInstance corresponds with the target of its Signal.

```
context SignalInstance inv:
self.target.model = self.model.target
```

- [2] The source of a SignalInstance corresponds with the source of its Signal.

```
context SignalInstance inv:
self.source.model = self.model.source
```

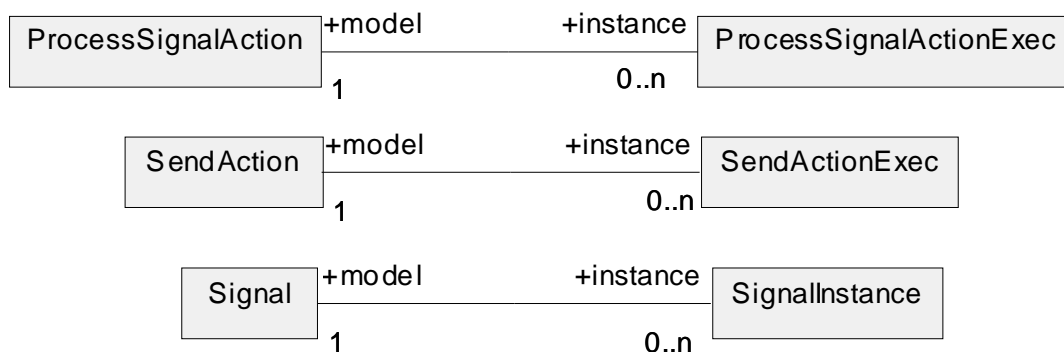


figure 2-23 The messaging.semantics package

## 2.10.4 Observations

Here we define `SendAction` and `ProcessSignalAction` as primitive actions, another approach might be to regard both actions as compound actions. The `SendAction`, for instance, composes sequentially a create object action which creates a signal object and a write action that adds the new signal to the output queue. In order to obtain a sound definition of `SendAction` as a compound action, another primitive action that adds an element to a collection needs to be defined.

## 2.11 THE MODELMANAGEMENT PACKAGE

### 2.11.1 The `modelmanagement.model.concepts` package

The *modelmanagement* package is shown in figure 2-24. It provides primitives for model management. It includes the concept `Package`, which is defined to be an administrative means that has no connection whatsoever to the meaning of the `ModelElements` it contains.

Key to the *modelmanagement* package is the distinction between owning and referencing a model element. Every model element is owned by only one package. A package may import model elements from other packages, in that case the importing package owns a reference to that model element, it does not own the model element itself. A model element reference may be used in exactly the same manner as the model element it refers to, given visibility aspects. One can distinguish between pure import and extension of model elements. In the first case the reference to the original model element may not hold any extra features, in the second case extra features are allowed. The first is called import, the latter is called extension or inheritance.

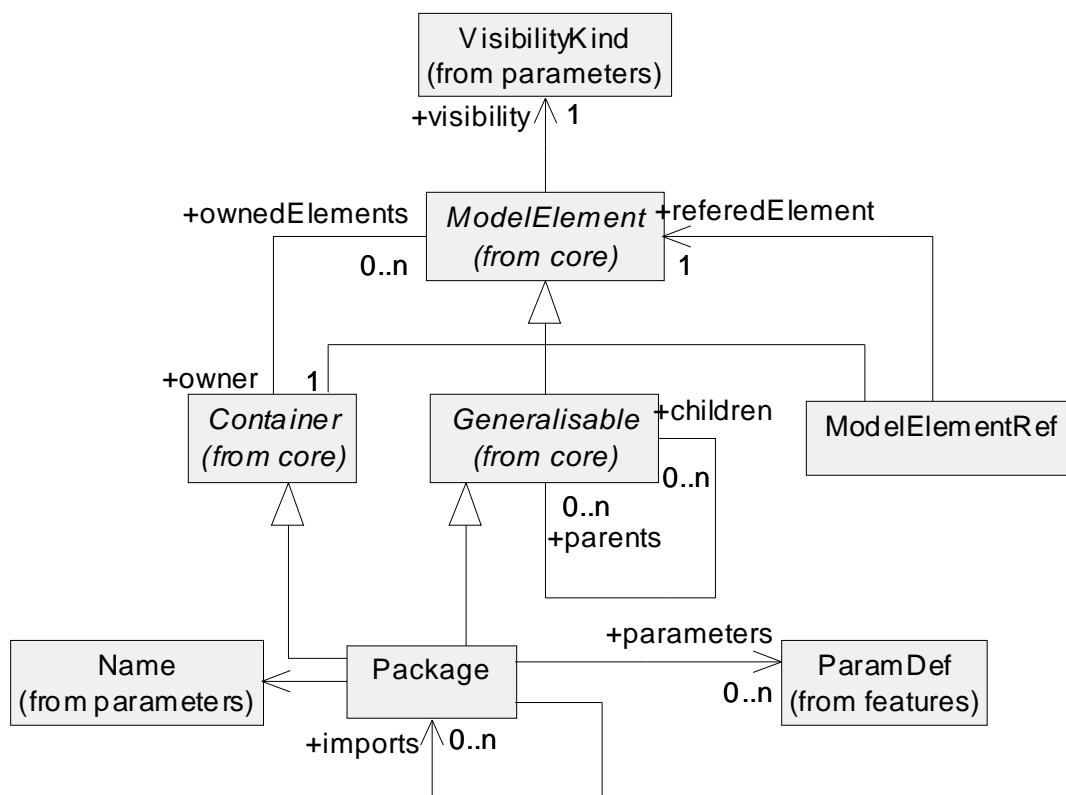


figure 2-24 The `modelmanagement.model.concepts` package

Another important fact is that visibility is not only an attribute of a `ModelElement`, but of `ModelElementRef` too. If a `ModelElement` is visible outside the container that owns it, other containers may hold a (their own) reference to it. The visibility of this reference may be restricted too. Every package that owns a reference to a model element may determine for itself only if and how that reference is visible to other packages, e.g. the ones that import the current package. Various visibility schemes can be defined using these primitives.

As model management is merely a means to handle the models written in the UML, there is no counterpart on the instance level, and no semantics subpackage either.

### 2.11.1.1 Definitions

**Package:** a generalizable container with a name.

**VisibilityKind:** an enumeration that denotes how the `ModelElement` to which it refers is visible outside its `Container`. (Value needs to be instantiated in order to pick one UML variant.)

**ModelElementRef:** a reference to a `ModelElement`.

### 2.11.1.2 Well-formedness rules

[1] The `referredElement` of a `ModelElementRef` is not owned by the container that owns the `ModelElementRef`.

```
context Container inv:
ownedElements->forall( o | o.isTypeOf( ModelElementRef ) implies
    not self->ownedElements->includes( o.referredElement ) )
```

[2] A package owns references to all `ownedElements` of its parents.

```
context Package inv:
parents.ownedElements->forall( p | self.ownedElements->exists( e |
    e.isTypeOf( ModelElementRef ) and e.referredElement = p )
```

[3] A model element reference may be used in exactly the same manner as the model element it refers to.

```
context ModelElementRef inv:
self.featureRef implies referredElement.featureRef
```

[4] A package owns references to the `ownedElements` of its imported packages based on visibility.

```
context Package inv:
imports.ownedElements->forall( p |
    p.visibility = public implies self.ownedElements->exists( e |
        e.isTypeOf( ModelElementRef ) and e.referredElement = p )
```

[5] A package may not add features to imported model elements.

```
context Package inv:
ownedElements->forall( e | e.isTypeOf( ModelElementRef ) implies
    e.features->isEmpty()
```

---

## 2.12 SOME NOTES ON THE SYNTAX AND OCL USED IN THIS DEFINITION

In [Clark2000] the requirements for a kernel meta modeling language are enumerated. One of these is that it should be possible to define the language in itself. As a consequence of this requirement the definition of our kernel meta model may contain only the concepts that are defined in the kernel itself. The definition in this chapter meets this requirement with some exceptions: the use of the Object Constraint Language [Warmer99], and the syntax. Indeed the kernel meta model contains the concept *invariant*, and an *expression* concept that fits the expressions in the OCL, but the predefined OCL operations (like 'oclIsTypeOf', 'includes', etc.) are not defined in this meta model, nor in the imported `predefTypes` package. The specification of the version 2.0 of the OCL, which does contain definitions of the predefined OCL operations, syntax as well as semantics, contains enough parallels with the work presented here to be easily integrated.

Another consequence of the requirements in [Clark2000] is that a syntax for the concepts defined in the kernel must be given in the kernel meta model itself. Again, this requirement is not completely met. In order to present the kernel in the diagrams we have used the concrete syntax of the OCL, plus only a number of con-

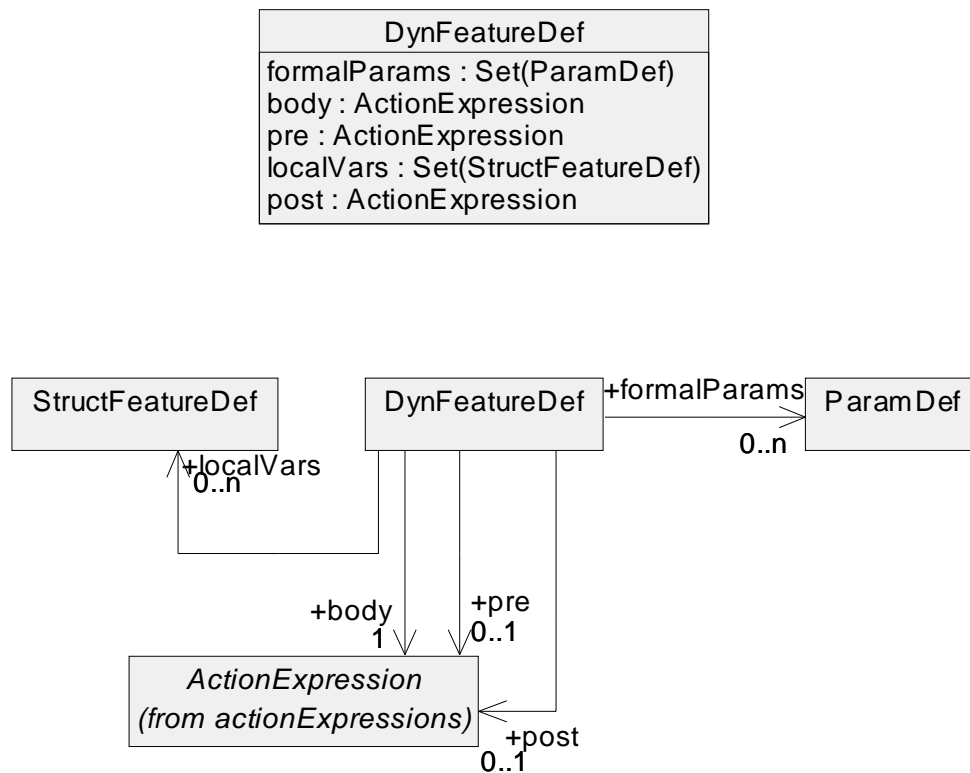


figure 2-25 **Two views: attributes versus associations**

create syntax constructs, without giving a precise definition. The syntax definitions given in MML of the following constructs are sufficient, and the concrete syntax of the OCL is well known.

To summarise, we have used the following symbols to represent the kernel meta model in diagrams:

- The class symbol is used to represent Roles. It is shown by a rectangle as in the UML1.3 specification, without the attributes or operations shown.
- The inheritance symbol is used to show the parent-child relationship between Generalisable model elements. It is depicted by an arrow with an closed, large arrowhead on the side of the parent (Generalisable) model element.
- The unary association symbol is used to show a unary association between two roles. It is depicted by an arrow with a small, open arrowhead. The arrow goes from the role that holds the unary association feature, and goes to the role that gives the type of the value in the slot (the referenced role). The string next to the symbol is the name of the unary association feature. A multiplicity is given only on the side that is referenced.
- The package symbol is used to represent Packages. It is depicted as in the UML1.3 specification, with only the name of the package shown.
- The dependency arrow is used to show the parent-child relationship between Packages. It is depicted by a dotted arrow. The arrowhead is shown on the side of the parent Package. When a dependency arrow is marked <<imports>>, the meaning is that the package at the arrowhead is imported into the package from which the arrow flows.

In the diagrams all use of attributes and operations is avoided. In our opinion an attribute named *n* in class *C* with type *A* is equal to an association from class *C* to class or data type *A* with rolename *n*. The use of attributes in the kernel meta model would obscure the fact that there is an association between two classes (or a class and a data type). Figure 2-25 shows the differences between both views.

# Chapter 3

## Syntax - the UML diagrams

In this chapter the packages are defined that link the core concepts defined in chapter 2, to the UML diagrams. Up to this point we have been interested in concepts, not in syntax. In this chapter only few new concepts will be defined, instead the focus will lie on the syntax of given concepts. We will indicate how the more simple forms of some of the UML diagrams can be linked to the concepts, thus hopefully showing that building the definitions for the more complex forms of these diagrams is merely a matter of transpiration, not inspiration. Not defined in this chapter is the syntax for the class diagram, which can be found in [Clark2000].

The packages defined in this chapter are not formalised in detail as the packages in the previous chapter. We acknowledge that the language definition in this report is not (yet) complete.

---

### 3.1 OVERVIEW OF PACKAGES DEFINED IN THIS CHAPTER

**compoundActions.model.activitySyntax2concepts.** Defines the mapping between the graphical activity diagram symbols, and the concepts in the `compoundActions.model` package

**messaging.instance.collaborationSyntax2concepts.** Defines the mapping between the graphical sequence diagram symbols, and the concepts in the `messaging.instance` package, thus defining a collaboration diagram on the instance level.

**messaging.instance.sequenceSyntax2concepts.** Defines the mapping between the graphical sequence diagram symbols, and the concepts in the `messaging.instance` package

**messaging.model.collaborationSyntax2concepts.** Defines the mapping between the graphical sequence diagram symbols, and the concepts in the `messaging.model` package, thus defining a collaboration diagram on the specification level, also called a role model.

**stateMachine.** Defines the concepts and syntax of state machines. Complete package that includes the usual subpackages: `model`, `instance`, and `semantics`.

---

### 3.2 THE GRAPHICAL SYMBOLS PACKAGE

In the following definitions we depend on the existence of a package that defines the graphical symbols. This package should be completely defined, as is the package `syntaxLibrary.baLDiagrams` in [Clark2000], but for the sake of brevity we have not included the complete definition here. It suffices to know that a package called `syntaxLibrary.graphSymbols` exists and that it contains the definitions of the following symbols:

- `GraphSymbol`: the superclass of all other concepts in this list
- `RoundedRectangle`, with three three separate text blocks called `text1`, `text2`, and `text3`
- `Arrow`, with three separate text blocks called `text1`, `text2`, and `text3`
- `Line`, with an unlimited list of separate text blocks called `text1`, `text2`, etc.
- `Rectangle`, with one text block
- `LongRectangle`, without any attributes

- FlattenedOval, with one text block
- Oval, with one text block
- Diamand
- Dot
- CircledDot
- ActivityDiagram
- Statechart
- SequenceDiagram
- CollaborationDiagram

### 3.2.0.1 Well-formedness rules

- [1] A Line is always associated with two rectangles, one called 'left', the other called 'right'.
- [2] A LongRectangle is always associated with a Rectangle.
- [3] A Diamond always has at least one ingoing and at least one outgoing arrow.
- [4] An Arrow may not enter or leave another Arrow.
- [5] An Arrow is always associated with two other GraphSymbols.
- [6] A Dot never has an ingoing arrow.
- [7] A CircledDot never has an outgoing arrow.

---

## 3.3 THE STATECHART DIAGRAM

The statechart diagram is defined by the *stateMachine* package, which inherits from the compoundActions package and the messaging package. It defines non-nested statecharts only.

### 3.3.1 The current statechart diagram definition

The current statechart diagram definition is centered around the notion of a state machine. In the UML specification one often finds the phrases 'the object has a state machine', or 'the object's state machine does ..'. This is a way of thinking that does not conform to the viewpoint taken in this study, which is that we can abstract our world to exist solely of objects that have relations with each other through their slots. In this view state machines are objects too, but what is their relationship to the primary object? Either the state machine is, as it were, an attribute of the primary object, or it is equal to the primary object. In this definition we take the viewpoint that the state machine is equal to the object, because it is the primary object that performs actions, and thus responds to state changes.

### 3.3.2 The stateMachine.model package

The *stateMachine.model.concepts* package, depicted in figure 3-1, defines two new concepts: state and transition. Key to the definition of the state concept is that it is considered to be a constraint, in stead of being a completely new aspect of a classifier. This is conform the definition given in the 1.3 version of the UML specification: "A state is an abstract metaclass that models a situation during which some (usually implicit) invariant condition holds." [UML1.3, page 2-135] We argue that although the state invariant may be kept implicit for a while during the modeling task, in the resulting model the invariant must be explicit in order to obtain, and maintain, a clear and definite relation between the model and the code of the resulting system.

Note that in general there are three manners of expressing state invariants:

- The state invariant may be expressed in terms of attributes that have their meaning even without regard to the states the object may be in. The attributes 'send-at-date' and 'payment-received-date' in an instance of

an Order class both have a specific meaning, but can also be used to express the state 'send-but-not-yet-paid-for'.

- The state invariant may be expressed by a dedicated attribute of Enumeration type that holds the state. For instance, if in an instance of the class Order the attribute 'order-state: {new, send, paid}' has the value 'send', it is in the state 'send-but-not-yet-paid-for'. Note that each level of nested states needs its own attribute.
- The state invariant may be expressed by a number of dedicated attributes of boolean type that hold the state. In our Order example an attribute 'send-but-not-yet-paid-for' could be defined; if true the instance is in the 'send-but-not-yet-paid-for' state, if false the instance is not in that state. Note that some extra constraints on these attributes are necessary to make sure that only one (or more, depending on the level of nesting of states) of the group is true at a certain point in time. You might call this the radio-button approach.

Note that a state may execute, which means that we evaluate the state expression and determine whether it is true or false.

The transition concept is, as state, not a completely new concept either. It is regarded as a special form of a conditional action, the test being that the object is in the correct state (i.e. the right state invariant is true) and the guard is true. If the test does not fail, the subaction of the conditional action is a combination of the exit action of the fromState, the transition-action, the entry action of the toState, and the do-activity, conform to the 1.3 specification.

### 3.3.2.2 Well-formedness rules

[1] The test of a Transition is an 'AND' combination of its guard, and its fromState.

```
context Transition inv:
test = guard and fromState.body
```

[2] The resulttype of the state must be of Boolean type.

*follows from rule [1] in package features.model.concepts.*

[3] Only roles may have states.

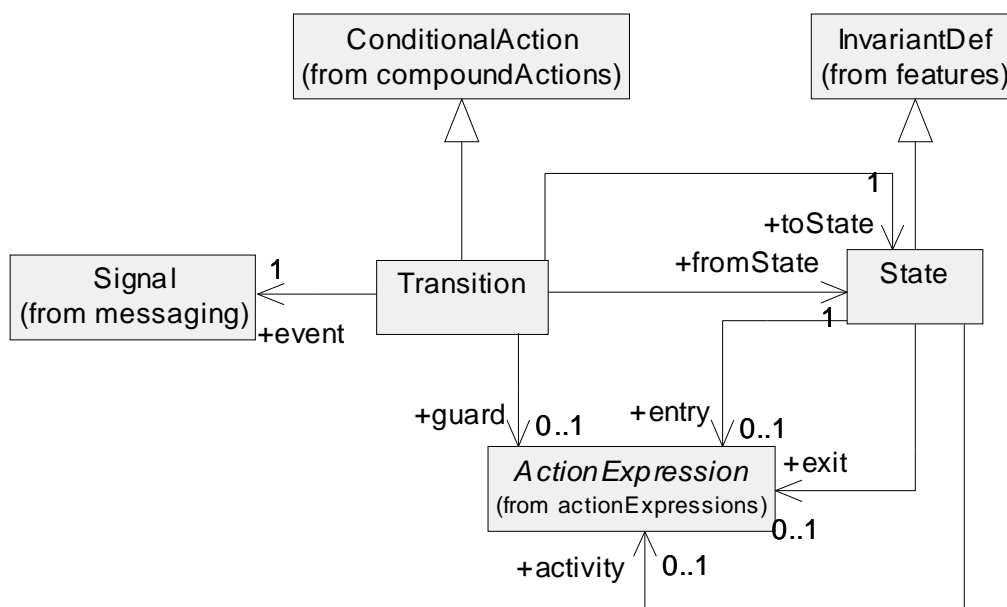


figure 3-1 The stateMachine.model.concepts package

```
context State inv:
selfType.isTypeOf( Role )
```

[4] The to and from states of a transition must belong to the same class as the transition.

```
context Transition inv:
toState.selfType = selfType AND fromState.selfType = selfType
```

### 3.3.3 The stateMachine.instance package

#### 3.3.3.2 The stateMachine.instance.concepts package

The *stateMachine.instance.concepts* package is shown in figure 3-2. Because State is defined as an invariant definition, and invariant definition does not have its counterpart on the instance level other than the execution of its body ActionExpression, there is no need for a corresponding concept in the instance package. The package only defines the concept of an execution of a transition. The execution of a transition is logically split into two steps, the first to determine whether the SignalInstance is on the input queue of the object that executes the transition, the second to test the state and guard, and execute the appropriate actions.

##### 3.3.3.3 Well-formedness rules

[1] The test of a TransitionExec checks whether the signalInstance is in the input queue of its self object.

*To be formalized.*

[2] The subaction of a TransitionExec is a ConditionalActionExec.

*To be formalized.*

### 3.3.4 The stateMachine.semantics package

As usual this package maps the concepts in the model subpackage to the concepts in the instance subpackage. It does not specify any syntax mappings.

#### 3.3.4.2 Well-formedness rules

[1] A TransitionExec may only be executed by a mutable value that has a role that has states for which transitions are defined.

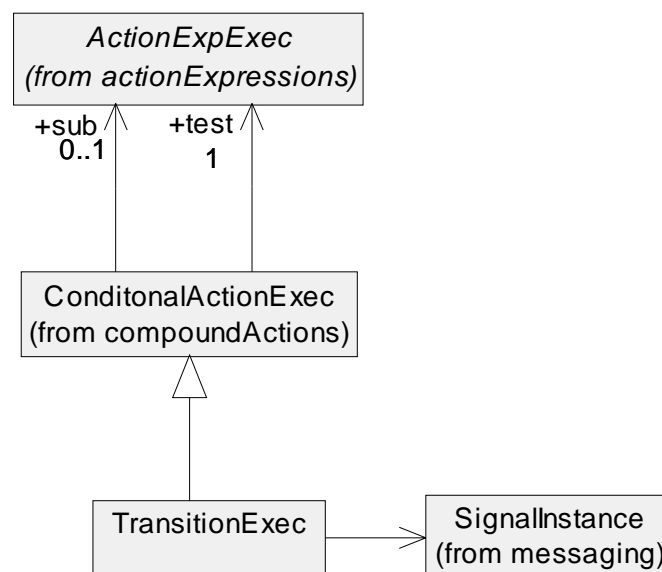


figure 3-2 The stateMachine.instance.concepts package

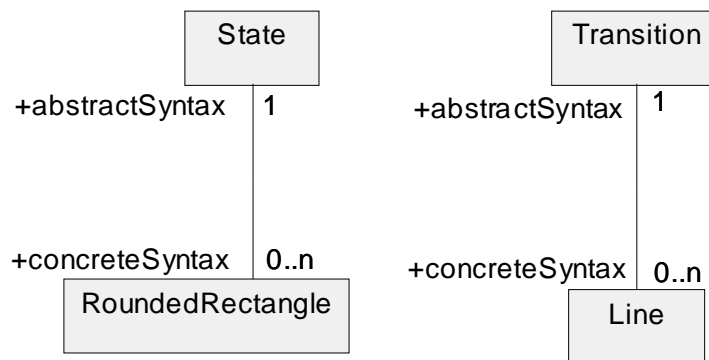


figure 3-3 **The stateMachine.model.syntax2concepts package**

[1] As stated in the stateMachine.instance package the subaction of a TransitionExec is a conditionalAction-Exec. The test of this subaction is the combination of the guard and state, the subaction of this subaction is the combination of the exit action of the fromState, the transition action, the entry action of the toState, and the do-activity of the toState.

*All well-formedness rules to be formalized.*

### 3.3.5 State machine syntax

The *stateMachine.instance.syntax* and *stateMachine.instance.syntax2concepts* packages are empty, because the statechart diagram is an expression of a model, and not of an instance of a model.

#### 3.3.5.2 The stateMachine.model.statechartSyntax package

The syntax of statecharts is well-known. In its simplest form a statechart is build up from rounded rectangles and arrows, both of which associated with some additional text.

#### 3.3.5.3 The stateMachine.model.statechartSyntax2concepts package

The stateMachine.model.statechartSyntax2concepts package is shown in figure 3-3.

#### 3.3.5.4 Well-formedness rules

- [1] A statechart may contain only RoundedRectangles, Arrows, Dots, and CircledDots.
- [2] The name in the RoundedRectangle represents the name of the State.
- [3] The exit text block in the RoundedRectangle represents the exit action of the State.
- [4] The entry text block in the RoundedRectangle represents the entry action of the State.
- [5] The do-activity text block in the RoundedRectangle represents the do-activity of the State.
- [6] The action text block of a Line represents the subaction of the Transition.
- [7] The guard text block of a Line represents the guard of the Transition.
- [8] The event text block of a Line represents the event of the Transition.

*All well-formedness rules to be formalized.*

### 3.3.6 Observations

The *stateMachine* package could have been defined without introducing the concepts State and Transition. This would merely take the rewriting of the well-formedness rules to apply to the parent concepts Invariant-Def and ConditionalAction. Because the State and Transition concepts are well known and much used, we have decided to add their definitions to this package. Furthermore, their presence opens up a path for a simple connection between the semantics described in this study and the semantics in the UML 1.3 specification.

---

## 3.4 THE ACTIVITY DIAGRAM

This section gives the definition of a basic activity diagram. It does not include object states, nor any other means to specify data flow.

### 3.4.1 The current activity diagram

The way the activity diagram in the Unified Modeling Language is currently defined is not object-oriented. This is a bold statement, but it is backed up by experts in the field. In a recent presentation for the OMG, Conrad Bock mentioned: an “application is completely OO when all action states invoke operations, and all activity graphs are methods for operations.” [Övergaard2000] In other words, using an activity diagram one can model a system in a completely non object-oriented way. In such an activity diagram the object-oriented principle of responsibility is not applied.

In fact, the current activity diagrams look in suspiciously many ways like the results of structured analysis and design, which uses functional decomposition to develop a software system. It is well known to object-oriented experts that structured analysis and design is a modeling paradigm that does not fit to the object-oriented paradigm. In the presentation for the OMG mentioned earlier [Övergaard2000] it is stated that activity graphs specify data/object flow. At the same time the UML 1.3 standard [UML1.3] includes the following quote (page 1-4):

“Why does not UML support data-flow diagrams? Simply put, data-flow and other diagram types that were not included in the UML do not fit as cleanly into a consistent object-oriented paradigm.”

As a kind of a compromise the concept of ‘swimlanes’ was introduced in the UML activity diagrams. A swimlane is a division of the diagram into parallel rectangles, where activities in the rectangle are supposed to be executed by the same instance. But a swimlane is merely an arbitrary partitioning of elements in an activity diagram, there is no link whatsoever with the concepts in the class diagram. So although swimlanes appear to denote some coupling of activities to classes and instances, their meaning is not strict and formally defined in the UML meta model. Furthermore, in practice diagrams with more than three swimlanes are more confusing than illuminating.

So, why not argue for removal of the activity diagram from UML? Indeed, currently we advise people using UML not to build any activity diagram, or at least, to be very careful with them. In many cases activity diagrams are not needed in an object-oriented model, but clearly there are circumstances where activity diagrams can be useful. In order for activity diagrams to be meaningful, we need to establish the semantics of the activity diagrams in terms of the other UML diagrams. This study relates all UML diagrams by defining each diagram to be the syntax of the common concepts defined in the kernel packages.

### 3.4.2 Activity diagram as syntax for compound actions

To give meaning to the activity diagram no new concepts are needed. It is enough to define a syntax package for the `compoundAction` package. Please note that an activity diagram is a diagram on the specification (or model) level, thus its symbols must be related to the `compoundAction.model` subpackage.

### 3.4.3 The `compoundAction.model.activitySyntax2concepts` package

An activity diagram represents a group action. It must therefore be associated with a single Role for which this group action is being defined. Every Oval represents a subaction in this group, i.e. an `ActionExpression`. A diamond indicates one or more conditional or loop actions. The test of each conditional or loop action is represented by the text at the arrow going out from the diamond. The oval to which the arrow points is the subaction of that conditional or loop action. If an arrow flows from that last oval back to the diamond a loop action is specified, otherwise a conditional action is specified.

### 3.4.3.2 Well-formedness rules

- [1] An ActivityDiagram may contain only FlattenedOvals, Arrows, Diamonds, Dots, and CircledDots.
- [2] The text in an Oval symbol represents the name and parameters of the Action.
- [3] The number of outgoing arrows of a Diamond is equal to the number of conditional or loop actions it represents.
- [4] The text at outgoing arrow of a diamond represents the test action.
- [5] The subactions of the GroupAction associated with an ActivityDiagram are partially ordered. The action associated with an Oval comes before any action associated with an Oval at the other end of an outgoing arrow, or with an Oval after an arrow, diamond, arrow combination.
- [6] The Action associated with the Oval directly after the Dot, is the first subaction of the GroupAction, i.e. there are no others before.
- [7] The Action associated with the Oval directly before the CircledDot, is the last subaction of the GroupAction, i.e. there are no others after.

*All well-formedness rule to be formalized.*

---

## 3.5 THE SEQUENCE AND COLLABORATION DIAGRAMS

### 3.5.1 The current interaction diagrams

The interaction diagrams show messaging, they form the two syntaxes for the messaging package. Both of the interaction diagrams show items on the instance level. Only a collaboration diagram may also be used as a model diagram, when it is used for role modeling. Therefore we need to establish a relation between the messaging package and the graphSymbols package on two levels: twice, for both sequence and collaboration diagram, to the instance subpackage, and once, for the role modeling collaboration diagram, to the model subpackage of messaging.

### 3.5.2 The messaging.instance.sequenceSyntax2concepts package

The mapping of the concepts from the messaging.instance subpackage to the graphSymbols package is straightforward. A Rectangle with its associated LongRectangle (lifeline) in a sequence diagram represents a MutableValue, an Arrow represents a signalInstance. The source of the SignalInstance is the MutableValue associated with the Rectangle/LongRectangle combination from which the arrow goes. The target of the SignalInstance is the MutableValue associated with the Rectangle/LongRectangle combination to which the arrow points. The contents of the SignalInstance is represented by text block with arrow. Note that a specific limitation of sequence diagrams is made very clear by this mapping: a SignalInstance can have only one target in a sequence diagram.

#### 3.5.2.2 Well-formedness rules

- [1] A SequenceDiagram may contain only Rectangles, LongRectangles (lifelines), and Arrows. (An extension would be to allow the actor symbol.)

*To be formalized.*

### 3.5.3 The messaging.instance.collaborationSyntax2concepts package

Again there is a very straightforward mapping between the messaging.instance subpackage and the graphSymbols package. A Rectangle represents a MutableValue, a Line represents a set of slots according to the semantics of the binary association given in section 2.7. The symbol Line holds an unlimited list of text blocks. Each of these text blocks represents a SignalInstance. If the text block contains the string '->' then the

source of the SignalInstance is the Rectangle which is 'left' of the Line, and the target is the Rectangle which is 'right' of the Line. If the text block contains the string '<' then the source of the SignalInstance is the Rectangle which is 'right' of the Line, and the target is the Rectangle which is 'left' of the Line.

### 3.5.3.2 Well-formedness rules

[1] A CollaborationDiagram may contain only Rectangles, Lines, and Arrows.

*To be formalized.*

---

## 3.5.4 The messaging.model.collaborationSyntax2concepts package

The next mapping to be defined is the mapping between the messaging.model subpackage and the graphSymbols package. Here a Rectangle represents a Role, not a MutableValue. The answer to the question whether to interpret a collaboration diagram as an instance or as a model diagram, can be found in the text in Rectangle. If the substring '/' is present, then the diagram depicts a mapping to the model subpackage.

The interpretation of Line is also different on the model level. Each of the text blocks associated with a Line represents a Signal, not a SignalInstance. The mapping of source and target can be defined in the same manner as for the instance level.

### 3.5.4.2 Well-formedness rules

To be defined.

# Chapter 4

## Some examples of additional packages

### 4.1 THE OPERATIONCALL PACKAGE

The operationCall package contains only the instance subpackage, because there is no way in the UML to specify the transport mechanism, and therefore there are no model elements that deal with communication.

#### 4.1.1 The operationCall.instance.concepts package

The operationCall.instance.concepts package is shown in figure 4-1. It defines only two new concepts: the bus, a very general concept that takes care of transportation of signals, and a subtype of bus: the OperationCallBus. A bus has a number of mutable values that are connected to it. These are the values to and from which the bus can transport signals.

Note that we can not state that the mutable value does not change between the moment a signal (an operation call) is removed from the output queue and the result signal is present in the input queue. The execution of the called operation may indeed cause changes to the slots of the calling object, when it includes a second call, which in turn is targeted at the calling object.

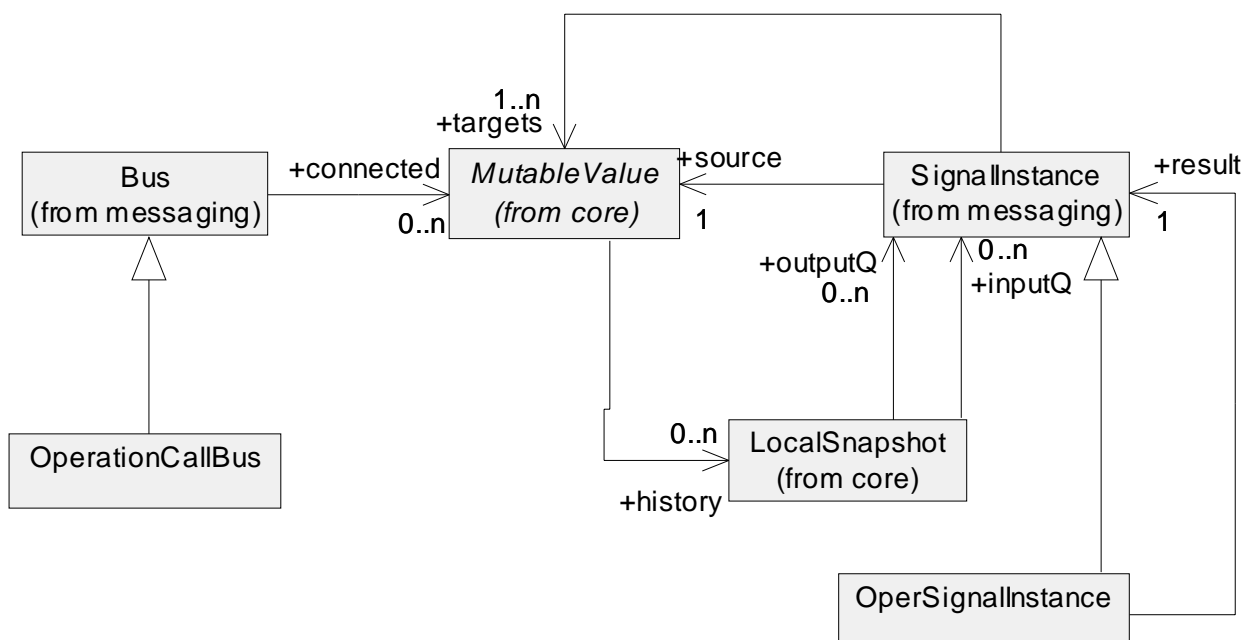


figure 4-1 The operationCall.instance.concepts package

### 4.1.1.2 Definitions

**OperationCallBus:** a transport mechanism that considers every signal to contain a reference to a dynamic feature definition.

**OperSignalInstance:** a `SignalInstance` that always is coupled with a result `SignalInstance`.

### 4.1.1.3 Well-formedness rules

- [1] When a signal instance contains a feature reference, there must be a resulting signal that contains the collection of the result and out parameters of the corresponding feature execution.

```
context OperSignalInstance inv:
  contents->isTypeOf( FeatureRef ) implies
    result.contents->isTypeOf( CollectionType( Classifier ) )1
```

- [2] If the feature reference that is the content of a signal, is part of a compound action, none of the other subactions will be executed before the result signal is received.

```
context GroupActionExec inv:
  subs->forall( sub: ActionExpressionExec |
    sub->isTypeOf( FeatureRefExec ) implies
      sub.after.previous.inputQ->includes( sub.resultSignal ) )
```

- [3] Directly after the point in time when a signal is put in the output queue of a mutable value, it is removed.

```
context MutableValue inv:
  history->forall( h: history |
    h.outputQ->notEmpty() implies h.next.outputQ->isEmpty() )
```

- [4] Directly after a signal is removed from the output queue, it is present on the input queue(s) of the target value(s).

```
context MutableValue inv:
  history->forall( h: history |
    h.outputQ->notEmpty() implies h.next.slotValues->exists( s: Snapshot |
      s.inputQ->notEmpty() ) )
```

- [5] Directly after a signal is present in the input queue of a mutable value it is processed.

```
context MutableValue inv:
  history->forall( h: history |
    h.inputQ->notEmpty() implies h.next.inputQ->isEmpty() )
```

- [6] Directly after a signal is processed, its result and out parameters are the contents of a signal that is present in the output queue of the processing value. The target of the signal is the source of the processed signal.

*To be formalized.*

- [7] Directly after the result signal is in the output queue it is removed, and present in the input queue of its target.

*To be formalized.*

---

## 4.2 THE ACTIONCLAUSE PACKAGE

In "Extending OCL to Include Actions" [Kleppe2000b] we proposed a new type of constraint: the action clause, which is a way to express that signals are or will be send, or that operations are or will be called. This is an important, and very useful extension to the OCL, e.g. for doing business modeling, as in Eriksson and Penker [Eriksson2000], and specifying outgoing events of components. This package defines the action clause in terms of the meta classes from the kernel package, but first we will introduce the action clause, because not many people will be already familiar with it. An action clause contains three parts:

---

1. To specify this constraint more precisely we need an extra collection type, one that may contain elements of different types. The expectation is that such a type will be introduced in version 2.0 of the OCL.

- [1] the set of target instances to which the signal (or signals) is send, called targetSet,
- [2] the set of signals that has been (or should be) send to this targetSet, called signalSet,
- [3] a condition, which is optional.

As concrete syntax for an action clause we use the following:

```
action: if <condition> to <targetSet> send <signalSet>
```

The targetSet is a list of objects reachable from the executing object, separated by comma's. The signalSet is a list of names with parameters between brackets, also separated by comma's. The condition can be any OCL expression of Boolean type.

Action clauses connected to an operation (in the terminology of the kernel package: dynamic feature) will be evaluated at postcondition time. Postconditions define what has been achieved by the operation, and so does the action clause. An example of the usage of an action clause with an operation is:

```
context CustomerCard::invalidate()
pre    : none
post  : valid = false
action: if customer.special to customer
                send politeInvalidLetter()
action: if not customer.special to customer
                send invalidLetter()
```

Another part of a system specification where the action clause is useful, is as invariant to a classifier (e.g. a class, interface, or component). An invariant states what should be true “at all times” for an instance of that classifier. In practice, violation of invariants will happen. Some of these violations could be fatal, that is, the system can not function correctly if the invariant does not hold. Other violations may be acceptable and only need some “mending” of the instance for the system to function properly again. There is a need to specify what should happen when such an invariant fails. We call this type of invariant action-invariant. An example of an action clause for a classifier:

```
context CustomerCard
inv    : validFrom.isBefore(goodThru)
action: if goodThru.isAfter(Date.now) to self
                send invalidate()
```

Assume we have an action, where we want to specify that, as a result of the action, some events has been sent. (We use the notation @post to indicate the point in time of the after snapshot of the action.)

- We can not guarantee that an event that was send by the action to a certain target, was received by that target @post (asynchronous messages). The transport mechanism might take a long time.
- We can not guarantee that an event that was send by the action to a certain target, and was received by that target, is still in the inputqueue of that target at action@post. It may be dispatched or removed in some other way (a pure real time view on dynamic semantics).
- We can not guarantee that the postcondition of an operation called (taking the operation call view on dynamic semantics) by the action is still true at action@post. Attribute and other values could have been changed in a number of ways after the called operation, as is shown in the example in section 2.
- We can not guarantee that an event that was send by the action is still in the outputqueue of the action@post.

Concluding, we may say that the only thing that can reasonably be guaranteed @post, is that an outgoing event has been in the virtual outputqueue during the execution of the operation. Therefore our definition of an action clause is:

*An action clause evaluates to true if, and only if, whenever the condition holds, the virtual outputqueue of the instance that executes the operation has contained at some point in time during execution of the operation, all <target, signal> pairs that are specified by the combination (Cartesian product) of the targetSet and the eventSet.*

### 4.2.1 The `actionClause.model.concepts` package

The `actionClause.model.concepts` package is shown in figure 4-2. It defines only one new concept: the action clause expression, and a new association from dynamic feature definition to action expression called *actClauses*. The association indicates any action clauses related to the definition of that dynamic feature.

#### 4.2.1.2 Definitions

ActionClauseExp: an expression that states that a signal should be (or have been) send.

#### 4.2.1.3 Well-formedness rules

- [1] The `targetSet` of an `ActionClause` is a list of objects reachable from the executing object.
- [2] If a signal in the `SignalSet` of an `ActionClause` contains a feature reference, this reference should be a reference to a feature of the target or targets.  
*To be formalized.*
- [3] The test of an `ActionClause` is of Boolean type.  
*To be formalized.*

### 4.2.2 The `actionClause.instance.concepts` package

The `actionClause.instance.concepts` package is shown in figure 4-3. It defines the execution counterpart of the action clause expression. This `ActionClauseExpExec` is equal to the `ActionExpExec`, apart from one extra association to a local snapshot. This snapshot is called the reference snapshot, and it represents the snapshot that is used to indicate the point in time from which the signal(s) send could/should have been in the output queue of the sending mutable value.

#### 4.2.2.2 Definitions

ActionClauseExpExec: the execution or evaluation of an action clause expression.

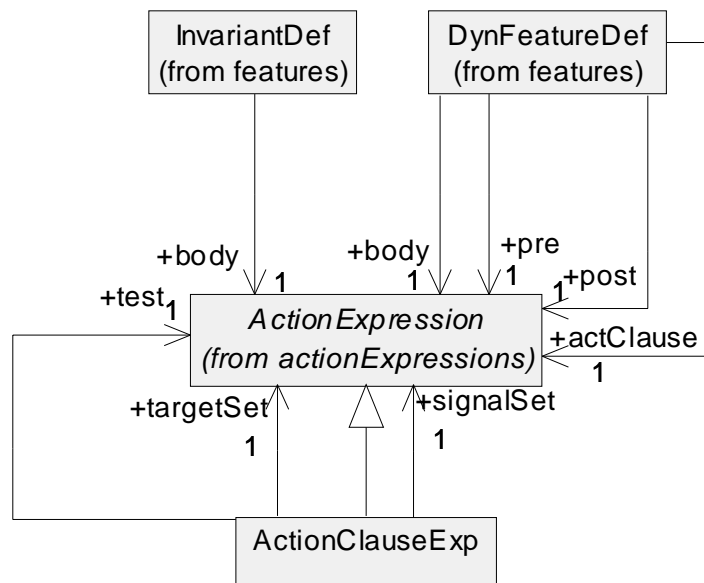


figure 4-2 The `actionClause.model.concepts` package

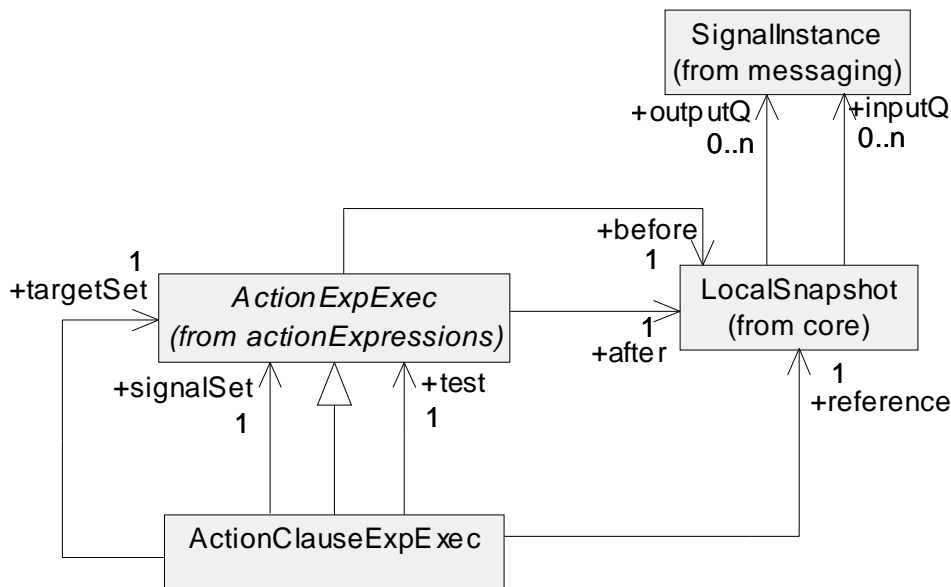


figure 4-3 The actionClause.instance.concepts package

#### 4.2.2.3 Well-formedness rules

- [1] The reference snapshot of an action clause expression execution must always be a part of the history of its selfInstance.

```
context ActionClauseExpExec inv:
selfInstance.history->includes( reference )
```

- [2] When the action clause is used as a postcondition of a dynamic feature, the reference snapshot is equal to the before snapshot of the body of the dynamic feature.

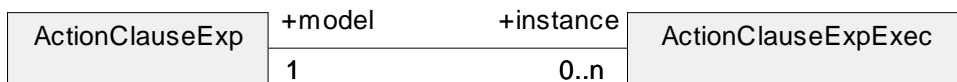
```
context DynFeatureDef inv:
actClause.instance.reference = body.instance.before
```

- [3] When the action clause is used as a action-invariant of a classifier, the reference snapshot is equal to its before snapshot.

```
context ActionClauseExpExec inv:
reference = before
```

- [4] Between the reference snapshot and the after snapshot of an action clause expression execution there should be at least one snapshot in the history of the selfInstance that holds in the output queue a signal instance such that the targets are equal to the result of the targetSet expression execution, and the contents are equal to the signalSet expression.

```
context ActionClauseExpExec inv:
selfInstance.history->exists( snap: SnapShot |
snap->indexOf() =< after->indexOf() and
snap->indexOf() >= reference->indexOf() and
snap->outputQ->includes( sign: Signal |
sign.targets = targetSet.result and
sign.contents = signalSet
)
)
```

figure 4-4 **The actionClause.semantics package**

### 4.2.3 The actionClause.semantics package

The actionClause.model.concepts package is shown in figure 4-4. It simply maps the ActionClauseExp to ActionClauseExpExec.

#### 4.2.3.2 Well-formedness rules

To be defined.

---

## 4.3 SUGGESTIONS FOR OTHER PACKAGES

The number of additional packages that may be defined is in principle unlimited. Below we give some suggestions on how other concepts could be defined on top of the kernel package.

**The time package.** A limited interpretation of time could be defined by adding a time slot in every mutable value, and adding a timer object that generates write actions on the time slots on a regular basis. Every change the timer triggers will lead to a new snapshot of the mutable value. Note that the notion of time being relative to the mutable value still holds. There are numerous reasons why the time in different mutable values may be different: the communication mechanism may delay the triggers in a random fashion, one mutable value may not respond to the trigger in the same speed as another mutable value, etc.

**The component package.** Component specifications (using the terminology from [Cheesman2001]) can be defined as a subtype of role. Component instances can be defined as a subtype of mutable value, where all slots of the component must be filled by objects, or other components, i.e. components may not have attributes. Interfaces can also be defined as a subtype of Role, component instances will have more than one role. Component instances will have an input and output queue, and will be connected to a communication bus.

# Chapter 5

## Extensibility and profile definitions

This section explains the inherent extensibility of the language defined by the kernel package. It proposes a new format for profile definitions based on these extensibility features.

---

### 5.1 EXTENSIBILITY OF THIS META MODEL

The meta model defined in chapter 2 (“The kernel meta model”) incorporates its own extensibility. Using the ways defined in the model management package to ‘inherit’ from the kernel package, anyone can build his or her extensions to this language. Chapter 4 gives a number of examples of extensions to the kernel. We argue that the extensibility of the language defined in this report is unlimited, and that any number of languages can be build on top of the given kernel.

However, UML is supposed to be a standard language for modeling object oriented software, therefore we need to set some limitations. Which languages may be called UML, and which may not? In our view the OMG should set up a number of packages that form a coherent set. This set constitutes the UML standard. Any combination of packages from this set defines a language that is compliant with the UML standard.

Users may build their own packages, as long as these packages ‘inherit’ from one of the standard packages. The resulting language will not be compliant with the UML in a strict sense, but it will be recognised as one of the UML family of languages.

#### 5.1.1 Profile definitions

The naming of the packages used in the definition of a certain UML variant, is called a profile. Such a profile may be expressed graphically, using a package diagram (e.g. figure 2-1), or textually. An example profile of a very simple UML variant is {kernel, procedureCall}.

#### 5.1.2 Extending this meta model as open source project

The extensibility of the meta model given in this report is such that the further development of the set of packages that together can be called the UML standard, can be done in a distributed manner, similar to the way open source projects are run. Small groups of people may produce a variety of packages based on the same kernel. The rules for such a project are easily stated:

- [1] Always build a new package by ‘inheriting’ from an existing package that is already part of the standard set of packages. Be sure to indicate this inheritance relationship clearly.
- [2] Always provide the model, instance, semantics, and syntax mapping subpackages.
- [3] Explain the meaning of the concepts in your package clearly.
- [4] Use class diagrams to express the relationships between the concepts.
- [5] Use invariant constraints to indicate any restrictions on the class diagrams.

- [6] Understand that a definition is always ‘owned’ by the package in which it was stated. You can not undo the definitions given in a package from which you inherit, although you may restrict them.
- [7] Post your package on an international forum and have it reviewed.
- [8] Revise any ailments.
- [9] Submit it to the standardisation party.

Note that for now there are no serious efforts for establishing this kind of open source project. We merely state that it would be feasible.

---

## 5.2 KNOWN OMISSIONS

The work on this topic is ongoing, this study describes only part of it. Some of the issues that need further examining are:

- A mechanism for name spacing, filling the NameTypeSpace and NameValueSpace objects used in calculating an expression.
- More options for visibility. In the kernel definition the visibility is always assumed to be public, this should be extended when building a complete language.
- The pUML group have provided a MML tool, this should be used (if possible) to check details of this study.
- Other types of transport mechanisms need to be defined, in order to further prove the feasibility of our approach.

---

## 5.3 CONCLUSION

The results of this study fit into the scheme proposed in pUML report for rearchitecting the UML. They provide enough evidence to indicate the feasibility of rearchitecting the semantics of the UML using the meta-modeling approach. This study also shows that the static and dynamic aspects of the semantics of the UML can not be separated. The semantics should be build from one integrated core consisting of objects, slots, and snapshots that bind the slots at a certain point in time to a value.

Although this report describes only parts of a new rearchitected meta model for the UML, and there are issues that need to be explored still, we are convinced that continued work following these guidelines will produce a semantics for the UML that is:

- understandable to people familiar with UML modeling,
- extendible,
- flexible,
- and will have more of the usual desirable features.

We hope that many of you, our readers, will get enthousiastic about the idea of starting an open source project and that you will all cooperate in making the semantics of UML as complete and sound as possible.

---

# Appendix A

## Imported types

---

This appendix defines the `predefTypes` package that can be used to actualize the kernel package defined in this report. You may freely replace this package by your own definitions in order to satisfy your specific needs, for instance, you may define the data types used in the programming language of your choice to use them in your UML models.

---

### A.1 PREDEFINED IMMUTABLE TYPES AND VALUES

The `predefTypes.model` subpackage is shown in figure A-1. It contains the three subtypes of the collection type defined in the OCL. Note that we do not need the Bag type, and that the SequenceType is used as an ordered set instead of an ordered bag. Furthermore, a number of basic data types is defined, amongst which four subtypes of EnumerationType: BooleanType, MultKindType, ParDirKindType and VisKindType. These are shown using the UML 1.3 syntax for enumeration types.

In the `predefTypes.instance` subpackage corresponding values are defined for all types. Because the types defined are well-known, the instance subpackage is not shown. Likewise not explained, is how expressions using data values, in the kernel package referred to as `DataValueExp`, can be formed.

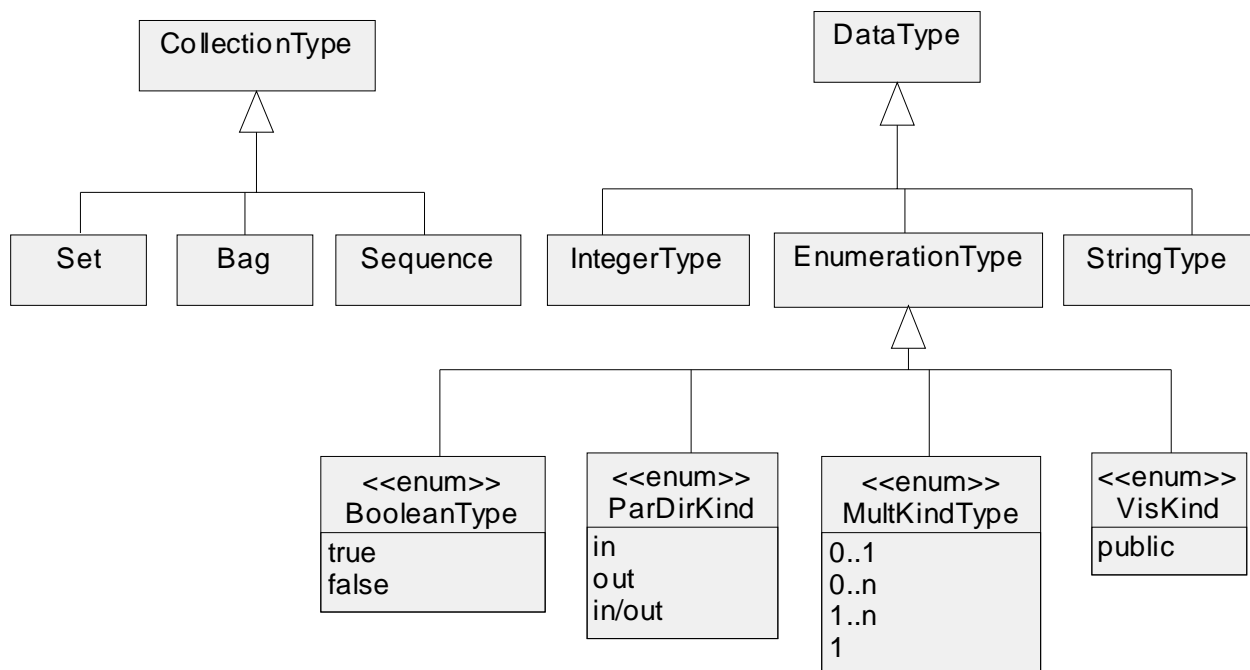


figure A-1 The `predefTypes.model.concepts` package



---

# Appendix B

## Definitions of meta classes

---

This appendix lists the definitions of all metaclasses from the kernel in alphabetical order.

---

<b>term</b>	<b>definition</b>
ActionExpExec	an executable unit that may generate changes in snapshots of related mutable values.
ActionExpression	a ModelElement with which one can indicate a certain Value, or a certain action to be performed.
Class	a MutableValue that represents a Role, or a Role that is itself a MutableValue.
Classifier	a ModelElement that defines a set of definitions (or rules) according to which Values can be classified. This follows the dictionary definition of the word ‘to classify’, which says to group or order items into collections. It can contain other ModelElements and it can be extended. Synonym: Type.
Bus	a transport mechanism that transports signals from output queue to input queues.
CollectionType	a type that defines values that can hold other values.
CollectionValue	an immutable value that can hold other values.
CompoundAction	an action that is an aggregate of (sub)actions.
CompoundAction-Exec	an execution of a compound action.
ConditionalAction	an action that consists of a test and a subaction, which specifies that the subaction has to be executed only if the test results in true.
ConditionalAction-Exec	an execution of a conditional action.
Container	a ModelElement that is able to contain other ModelElements.
CreateObjectAction	an expression that specifies a change of the snapshot of a class in such a way that the slot ‘new’ contains a new object which is an instance of that class.
CreateObjectAction-Exec	the execution of an expression that creates (makes known to the environment) a mutable value.
DataType	a Classifier, therefore a set of rules, including ‘an instance of this type is immutable’, and ‘an instance of this type has no references to other instances’.

---

<b>term</b>	<b>definition</b>
DataValue	a Value that can not be changed and does not contain any parts.
DataValueExp	an expression consists solely of data values and operations on data values.
DataValueExpExec	an expression that results in a data value.
Definition	a definition of an aspect of a Value. Synonyms: Rule, Property, Feature.
DomainElement	an element that can be reasoned about (specified, modelled) using the UML.
DynFeatureDef	a definition of a feature that may execute and may change state.
Element	an unnamed reference to a value.
EnumerationType	a type that defines a limited set of values.
EnumerationValue	a Value picked from a limited set of values.
FeatureRef	a reference or call to a Definition.
FeatureRefExec	the execution or evaluation of a feature.
Generalisable	a ModelElement from which other ModelElements can be made. It captures the notion of extensibility which is needed to build the meta model from this core package.
GroupAction	an action that consists of a number of subactions that may be executed in any order.
GroupActionExec	an execution of a group action.
ImmutableType	a definition of Values that are immutable.
ImmutableValue	a Value that can not change, or be changed.
InvariantDef	a definition of a rule, mostly based on structural features.
LocalSnapshot	a collection of NameValueBindings for a certain MutableValue at a certain point in time.
LoopAction	an action that consists of a test and a subaction, which specifies that the subaction has to be executed a number of times until the test results in false.
LoopActionExec	an execution of a loop action.
ModelElement	a part of a specification or model written in the UML.
ModelElementRef	a reference to a ModelElement.
MultiplicityKind	an indication of the number of elements in the collection of the unary association.
MutableValue	a Value that has slots of which the value may change.
Name	a reference. (Needs to be instantiated in order to pick one UML variant. Usually it takes the form of a string type.) Synonym: Variable.
NameTypeSpace	a set of Definitions, coupling Names to Types (or Classifiers).
NameValueBinding	a combination of a Name (reference) and a Value.
NameValueSpace	a set of NameValueBindings that are used to evaluate an actionExpression. Synonym: Context, Environment.
NullValue	a Value that represents void (null, nul, nil).

---

<b>term</b>	<b>definition</b>
Number	a serial number determining the order (if any) of elements in the collection.
Object	a MutableValue that has a state, represented in a LocalSnapshot, and is able to change that state, thereby creating a new LocalSnapshot.
Package	a generalizable container with a name.
ParamDef	a definition with a ParameterDirectionKind, that can be actualized by an actual parameter.
ParameterDirection-Kind	a definition of an enumeration that denotes if the parameter is used for supplying an argument and/or for returning a value. (Needs to be instantiated in order to pick one UML variant.)
PrimitiveAction	an action that specifies a change of the snapshot of the mutable value that executes the action, the self instance, only.
PrimitiveActionExec	the execution of a primitive action.
ProcessSignalAction	an action that specifies a change to its self value so that a signal is removed from the input queue.
ProcessSignalExec	the execution of the receipt of a signal, i.e. taking a signal with its content from the input queue of the 'self' mutable value.
ReadAction	an expression that specifies a certain value, within the context of a single mutable value.
ReadActionExec	an ActionExpExec that evaluates the NameValueBindings in the NameValueSpace, and results in the Value that is bound to a given Name.
Role	a Classifier, therefore a set of rules, including 'an instance of this type is mutable', and 'an instance of this type may have references to other instances'.
SendAction	an action that specifies a change to its self value so that a signal is placed in the output queue.
SendSignalExec	the execution of the sending of a signal, i.e. putting a signal with its content in the output queue of the 'self' mutable value.
Signal	the packaging of information that has as source one mutable value, and as targets one or more mutable values.
SignalInstance	an occurrence of a signal.
StructFeatureDef	a definition of a feature that will not execute and/or change state, it merely is.
UnaryAssociation	a structural feature that can be used to reference another mutable value, therefore its type can only be a role, or a collection type. (The collection types need to be actualized, e.g. from the package predefTypes.)
Value	a DomainElement that represents one of the elements of the world that can be modeled using the UML.
VisibilityKind	an enumeration that denotes how the ModelElement to which it refers is visible outside its Container. (Value needs to be instantiated in order to pick one UML variant.)

---

<b>term</b>	<b>definition</b>
WriteAction	an action that specifies a change of the value of one of the slots of the related mutable value.
WriteActionExec	the execution of an expression that changes the value of a slot of a mutable value.

# Appendix C

## Overview of kernel model concepts

This appendix gives an overview of all concepts defined in the model subpackages of the kernel package. It is shown in figure C-1. All inheritance relations are shown as in chapter 2, but for spatial reasons no associations are shown. The colouring indicates the package in which the item is defined.

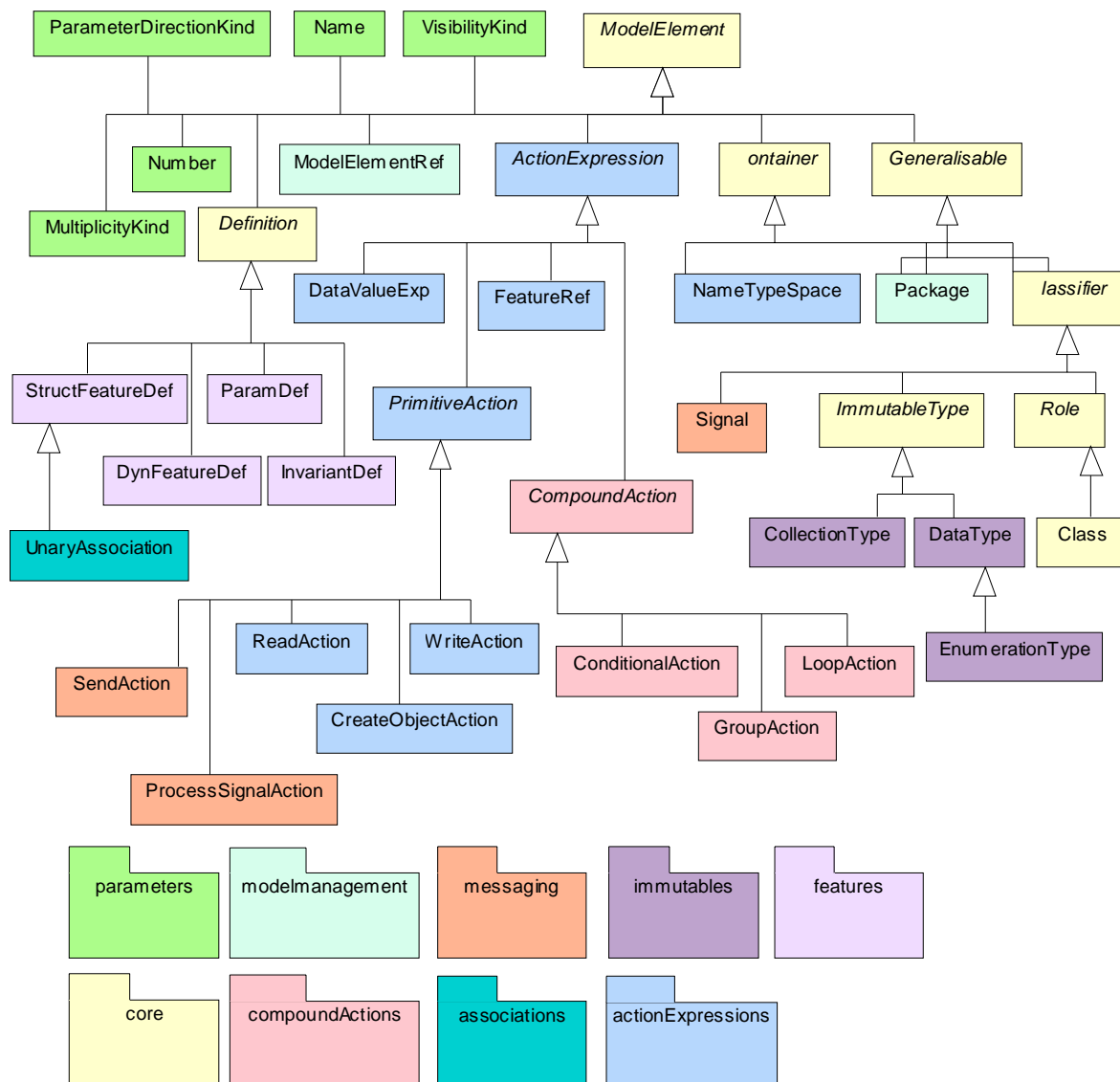


figure C-1 Overview of all model concepts from the kernel



# References

- [AS2001] Response to OMG RFP ad/98-11-01 *Action Semantics for the UML*, March 24, 2001, available from [www.kc.com/as\\_site/home.html](http://www.kc.com/as_site/home.html)
- [Cheesman2001] John Cheesman and John Daniels, *UML components, A simple Process for Specifying Component-Based Software*, 2001, Addison-Wesley
- [Clark1999] Tony Clark, Andy Evans, Stuart Kent, France R., Rumpe B., *pUML Response to UML2.0 RFI*, 1999, available from [www.puml.org](http://www.puml.org)
- [Clark2000] Tony Clark, Andy Evans, Stuart Kent, Steve Brodsky, Steve Cook, *A feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modelling Approach*, version 1.0, September 2000, available from [www.puml.org](http://www.puml.org)
- [Cook94] Steve Cook and John Daniels, *Designing Object Systems: Object-oriented modeling with Syntropy*, 1994, Prentice-Hall
- [Cook99] Steve Cook, Anneke Kleppe, Richard Mitchell, Jos Warmer, and Alan Wills, *Defining the Context of OCL expressions*, UML'99, 1999.
- [Cox96] Brad Cox, *Superdistribution: Objects As Property on the Electronic Frontier*, 1996, Addison-Wesley
- [Doesburg94] Ed Doesburg, Doré Eijkman, Anneke Kleppe and Jos Warmer, *Practical Experiences with OMT*, in Proceedings of TOOLS 13, 1994, Prentice Hall
- [D'Souza99] Desmond D'Souza and Alan Wills, *Objects, Components and Frameworks with UML, The Catalysis Approach*, 1999, Addison-Wesley
- [Engels2000] *Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML*, Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer, UML 2000, 2000
- [Eriksson2000] Hans-Erik Eriksson and Magnus Penker, *Business Modeling with UML, Business Patterns at Work*, 2000, Wiley
- [Evans98] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe, *The UML as a Formal Modeling Notation*, in The Unified Modeling Language: first international workshop, Mulhouse, France, June 1998, 1999, Springer
- [Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, 1995, Addison-Wesley
- [Goldberg95] Adele Goldberg, and Kenneth S. Rubin, *Succeeding with Objects, Decision Frameworks for Project Management*, 1995, Addison-Wesley
- [Kleppe2000a] *Making UML Activity Diagrams Object-Oriented*, Anneke Kleppe and Jos Warmer, Tools Europe 2000, June 2000
- [Kleppe2000b] *Extending OCL to Include Actions*, Anneke Kleppe and Jos Warmer, UML 2000, September 2000
- [Kleppe2001] *Integration of Static and Dynamic Core for UML: A Study in Dynamic Aspects of the pUML OO Meta Modelling Approach to the Rearchitecting of UML*, Anneke Kleppe and Jos Warmer, Tools Europe 2001, March 2001
- [Kwon2000] *Rewrite Rules and Operational Semantics for Model Checking UML Statecharts*, Gihwon Kwon, UML 2000, October 2000
- [Marcotty76] M. Marcotty, H.F. Ledgard and G.V. Bochmann, A sampler of formal definitions, *Comput. Surveys* 8 (1976), pages 191-276
- [Martin98] James Martin and James J. Odell, *Object-Oriented Methods, A Foundation*, 1998, Prentice-Hall
- [OMG2000] OMG Request For Proposals ad/2000-09-01 for a UML 2.0 Infrastructure, September, 2000

- [OMG98] OMG Request For Proposals ad/98-11-01 for an Action Semantics for the UML, November, 1998
- [Övergaard98] A Formal Approach to Use Cases and Their Relationships, Gunnar Övergaard and Karin Palmkvist, UML 1998, 1998
- [Övergaard2000] Gunnar Övergaard, Bran Selic, and Conrad Bock, *Object Modeling with UML, Behavioral Modeling*, Presentation in the OMG tutorial series, January 2000.
- [Reggio2001] G. Reggio, and E. Astesiano, *A Proposal of a Dynamic Core fro UML Metamodelling with MML*, available from <ftp://ftp.disi.unige.it/person/ReggioG/ReggioAstesiano01a.pdf> , April 2001
- [UML1.3] OMG Unified Modeling Language Specification, Version 1.3, June 1999
- [Warmer99] Jos Warmer and Anneke Kleppe, *The Object Constraint Language, Precise Modeling with UML*, 1999, Addison-Wesley