ULRICH FRANK

THE MEMO META-METAMODEL

Juni 1998

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 9

UNIVERSITÄT KOBLENZ · LANDAU

Institut für Wirtschaftsinformatik
Fachbereich Informatik
Universität Koblenz-Landau

ULRICH FRANK

THE MEMO META-METAMODEL

Juni 1998

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 9

Anschrift des Verfassers/
Address of the author:

Prof. Dr. Ulrich Frank
Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
D-56075 Koblenz

Arbeitsberichte des Instituts für Wirtschaftsinformatik
Herausgegeben von / Edited by:

Prof. Dr. Ulrich Frank
Prof. Dr. J. Felix Hampe

Bezugsquelle / Source of Supply:

Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
56075 Koblenz

Tel.: 0261-9119-480
Fax: 0261-9119-487
Email: iwi@uni-koblenz.de
WWW: http://www.uni-koblenz.de/~iwi

iwi Institut für Wirtschaftsinformatik
Fachbereich Informatik
Universität Koblenz-Landau

Abstract

"Multi Perspective Enterprise Modelling" (MEMO) is a method to support the development of enterprise models. It suggests a number of abstractions which allow to analyse and design various interrelated aspects like corporate strategy, business processes, organizational structure and information models. Any of those views can be modelled with a specific modelling language or diagram technique respectively. In order to allow for a tight integration of the various perspectives, the modelling languages suggested by MEMO are based on common concepts. Those concepts are defined using a common meta level language. It serves to specify more specialized languages, such as the MEMO Object Modelling Language (MEMO-OML) or the MEMO Organisation Modelling Language (MEMO-OrgML). The meta level language itself is defined within a meta-metamodel. Based on a discussion of the purpose to be fulfilled by a meta-metamodel and a comparison with other meta-metamodels, this report provides a semi-formal specification of the semantics, the abstract syntax and the notation (or concrete syntax) of the MEMO meta-metamodel.

1. Introduction

MEMO ("Multi Perspective Enterprise Modelling") is a method that supports the design of multi perspective enterprise models. The models that cover a particular perspective on an enterprise are designed to support specific tasks related to the planning, developing and introducing corporate information systems. There are, for instance, models of the corporate strategy, the business processes, the organisation structure, or the information objects. The subjects of these models are interdependent. Therefore, the models should be highly integrated in order to allow for synergy. For instance: A model of a business process will often include descriptions of information that is required or produced within the process. In this case, it can be a good idea to refer to corresponding parts of an object model. On the other hand, every partial model is related to specific tasks - like analyzing and (re-) designing a company's organisation or developing software to be used within its information system. Therefore, a particular kind of model requires specific abstractions and concepts. In order to accomplish both goals - the need for integration and for specialised concepts, MEMO provides a set of specialised modelling languages. They can be compared to specialised terminologies, like the terminology of software-engineers, of management consultants, etc. The various modelling languages provided by MEMO (for an overview see [Fra97]) are integrated via common concepts which are defined in a common metalevel language.

2. The MEMO Meta-Metamodel

There are different ways do define a language, like grammars or metamodels. Both can be introduced with more or less formal rigour. While a grammar provides advantages for languages with sequential representations, it is less intuitive to apply for languages with graph-oriented representations. Furthermore, using a grammar would also result in a paradigm clash since descriptions on other levels within the MEMO framework are represented by (graphical) models. Therefore, we introduce a model to define the metalanguage that is used for the specification of the MEMO modelling languages. Since the use of the term "meta" depends on the context, it can be misleading. In order to avoid confusion, we will use the term meta-metamodel for the model that describes the metalevel language which is used in turn to define the metamodels that specify the modelling languages.

2.1 Purpose of a Meta-Metamodel

Introducing a meta-metamodel is motivated by various reasons. They are all based on the assumption that a meta-metamodel can be restricted to a small set of core concepts which are stable over time. In this case, a meta-metamodel facilitates the exchange of object models which may be instantiated from different metamodels - provided those are instances of a common meta-metamodel: It allows to map instances of corresponding concepts from one metamodel to another - which, however, does not guarantee to avoid the loss of semantics. In case a modelling method includes various modelling languages/diagram techniques which supplement each other, a common meta-metamodel helps to integrate different diagrams in a coherent way. The existence of a meta-metamodel also fosters the extensibility of a modelling language: The metamodel can be modified without changing the existing foundation. Additionally a meta-metamodel helps with understanding a metamodel since it identifies the - usually small number - of core concepts.

Fig. 1 gives an impression of the hierarchy of models used within MEMO. Notice that the metamodels (like the MEMO-OML metamodel) are not intended to be an object model for a CASE tool. Instead, our emphasis is on the specification of a modelling *language* that can be used without a tool. In order to allow for a tool supported development and maintenance, the metamodels will be reconstructed by object models.

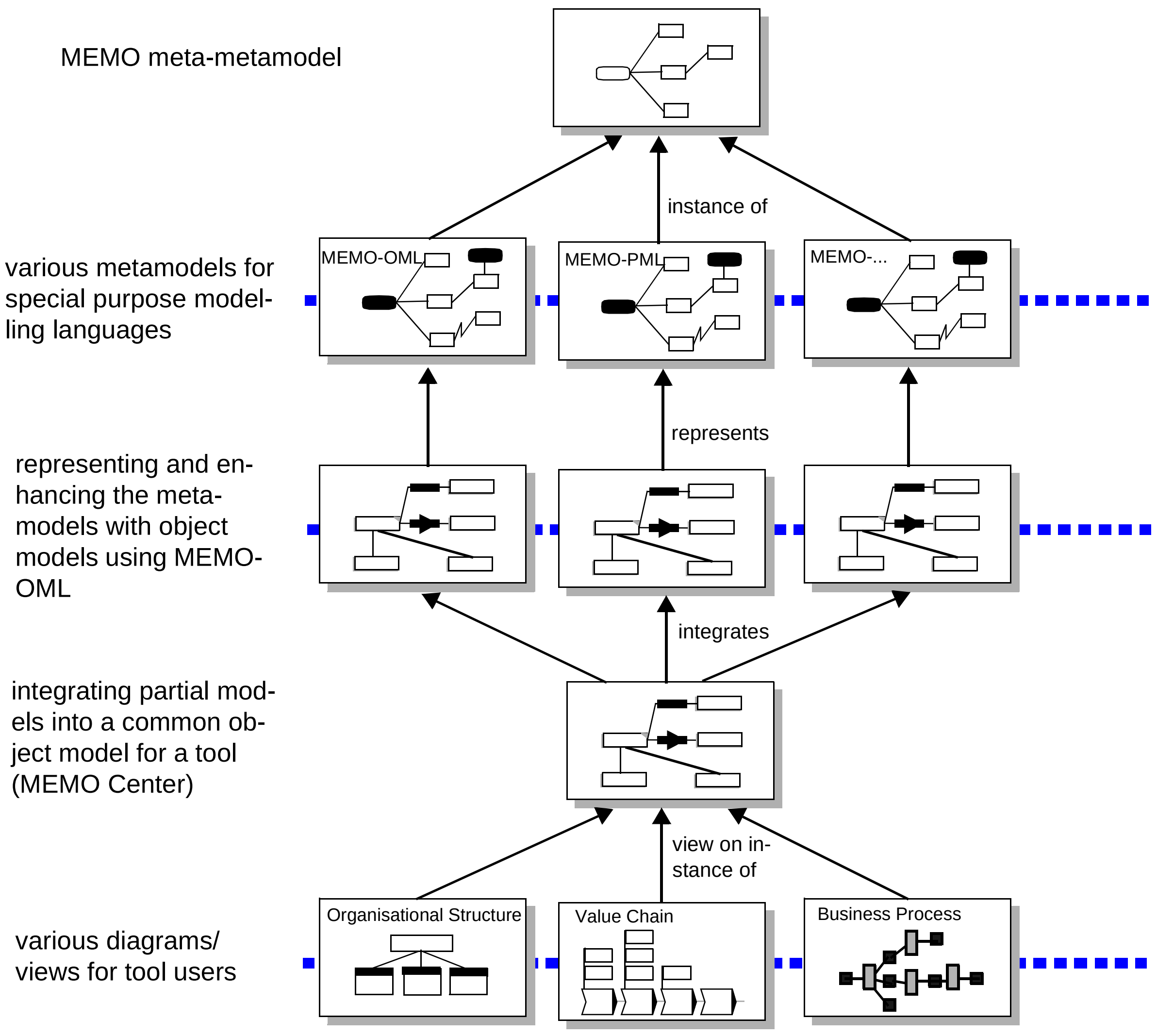

MEMO meta-metamodel
instance of
various metamodels for special purpose modelling languages
MEMO-OML
MEMO-PML
MEMO-...
represents
representing and enhancing the metamodels with object models using MEMO-OML
integrates
integrating partial models into a common object model for a tool (MEMO Center)
view on instance of
various diagrams/ views for tool users
Organisational Structure
Value Chain
Business Process

Fig. 1: Meta-Metamodel for Modelling Languages and its Relationship to Object Models for Tools

Similar to any other model, a the definition of meta-metamodel requires a set of higher order concepts - we could also speak of a meta-meta-metalanguage. In order to avoid a *regressum ad infinitum*, the concepts used for a language specification can be formalized at some point. That would require to define a set of symbols, a precise syntax and a calculus that would allow to generate all valid expressions (models). Another option to avoid an infinite series of meta-level descriptions is to use concepts with well known semantics - with an acceptable amount of ambiguity.

The meta-metamodel is defined using a small number of well known concepts. The concepts used to constitute the metamodel can be regarded as meta entity types - or within the scope of the meta-metamodel - as entity types. An entity type may have a set of attributes. An attribute is specified by one of two base types: *Integer*, *String*. Their instances are positive integers and zero, and strings. As a default, an attribute has the multiplicity 1,1. Attributes typed in *italic* have the multiplicity 0,1. At this point, *MetaName* and *MetaID* can be considered as String. They are only used to provide a higher level of abstraction which will facilitate future refinements. Furthermore, concepts of the meta-metamodel are defined via specialisation and associations.

An association between two concepts indicates that their instances must or may refer to one another. For each concept it is assigned a multiplicity, differentiated in minimum and maximum cardinality. A specialized entity type inherits all the attributes and associated entities from the superordinate entity type. Notice that specialisation is restricted to single inheritance. That implies that for the specification of the metamodels instantiated from the MEMO meta-metamodel only single inheritance is available. However, that does not exclude the definition of multiple inheritance to be used on the object level. All entity types are specialized from *MetaObject* (see fig. 2). MetaObject's sole attribute, *notation*, serves to specify how an instance of a concrete subconcept is to be rendered. While this representation can be defined in various ways, we assume at this point that there is a textual definition (which might be used to refer to a different format). In a similar way, the attribute "naming" within *MetaConcept* allows to specify naming conventions for the instances of its instances (for example: for naming attributes within an object model).

MetaEntity is the only concept within the meta-metamodel that allows for being specialized. Each instance of MetaEntity has to be assigned a name which must be unique within a metamodel. *MetaAssociation* serves to specify associations between instances of MetaEntity. Different from the object level, we do not differentiate between various kinds of associations (like interaction, aggregation). *MetaConstraint* can be associated with either instances of MetaEntity, MetaAssociation or with an instance of MetaModel. *MetaComment* allows to annotate any element of a metamodel (including instances of MetaComment) as well as the metamodel itself. MetaObject, MetaElement and MetaConcept are considered as abstract entity types. Since a graphical representation is not sufficient to express the essential semantics of the meta-metamodel, it is supplemented by a number of natural language constraints. As soon as the meta-metamodel seems to be mature enough, we will provide a formal specification.

The notation used to render the meta-metamodel corresponds to common conventions. A class is rendered as a rectangle filled with the name of the class and, if applicable, the names of attributes. Generalisation/specialisation is rendered as an arrow, leading from a class to its superclass (within the meta-metamodel we use single inheritance only). Relationships are rendered as thinner arrows which carry cardinalities and a designator to be read in the direction defined by the arrow. Abstract classes are characterized by a filled triangle in the upper left corner of the corresponding rectangle.

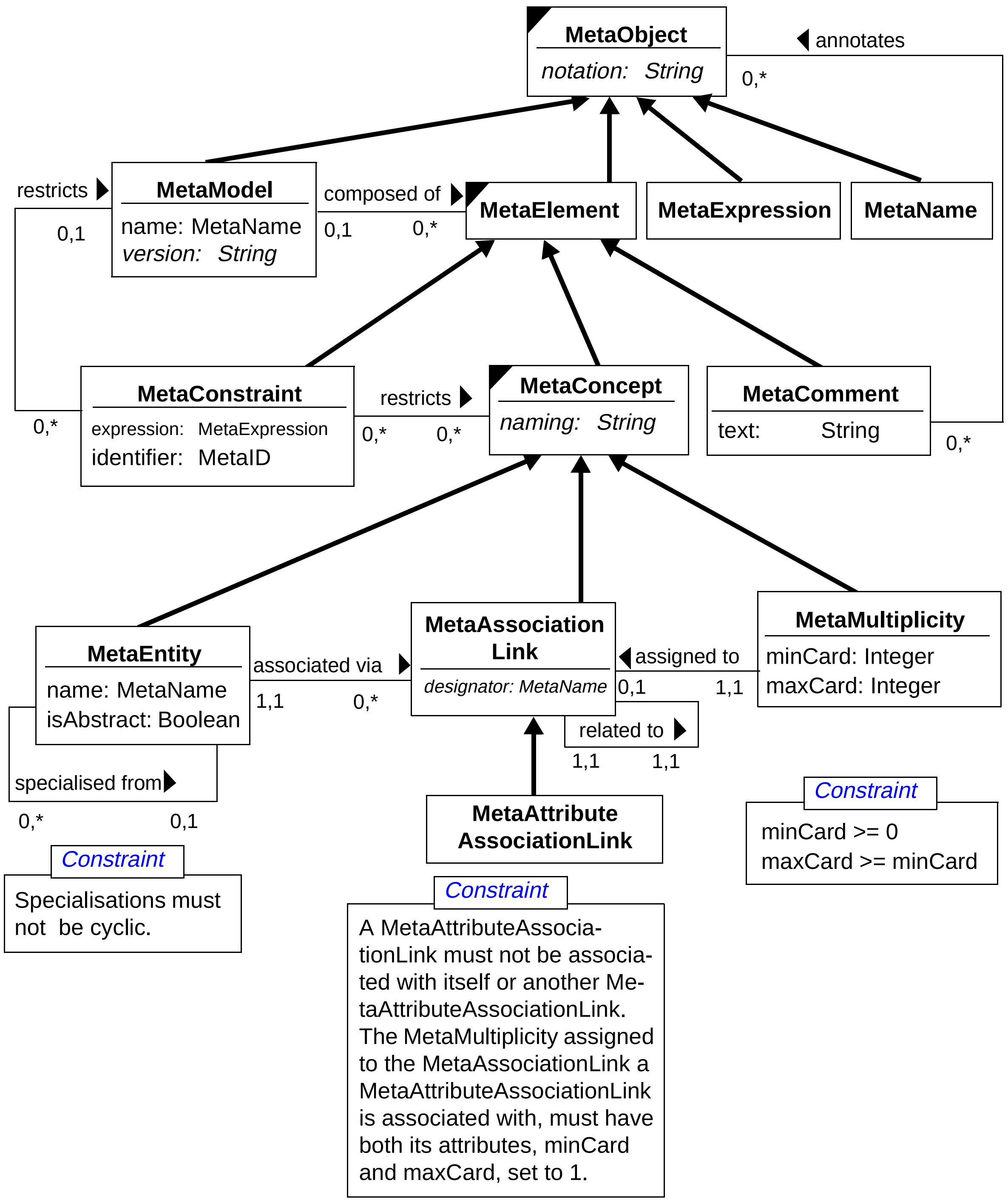


Fig. 2: The MEMO-OML Meta-Metamodel

2.2 Additional Constraints

A metamodel as an instance of the meta-metamodel serves to render the semantics and abstract syntax of a modelling language in an illustrative way. There are, however, limits to the expressive power of pure graphical models. In order to fill the semantic gaps left by the graphical representation of the metamodel, an additional specification language is required - which would be used to form expressions as instances of MetaExpression. The UML proposition currently under review by the OMG ([Rat97a], [Rat97b]) includes a formal specification language that has been developed by IBM. The Object Constraint Language (OCL, [Rat97d]) is based on a syntax which is very similar to Smalltalk. Different from Smalltalk, however, OCL uses statically typed classes (entities) only. GRAL (Graphical Specification Language, [EbWi96], [Frz97]) is another option.

From both options we favour GRAL - mainly for three reasons: GRAL is based on a solid formal foundation that resulted from intensive research. There is a Meta CASE tool that allows to instantiate specific CASE tools from metamodels specified in GRAL ([EbSü97]). Finally, GRAL was developed at Koblenz university within a team we cooperate with closely. GRAL allows to specify constraints on so called TGraphs. A TGraph is a directed graph composed of vertices and edges. Both, vertices and edges, are typed and may have attributes. GRAL comes with a library of predicates typically needed to express properties of graphs (e.g.: "isAcyclic (G)", "isNeighbourOf (G,v,w)"). Additional predicates can be defined using the specification language Z. In order to specify first order predicates on TGraphs (and/or vertices and edges) it is required to navigate the graph on any path that might be of relevance for a particular constraint. The edges used within an expression are identified by the vertices they connect and their designator(s). For a detailed specification of TGraphs see [EbFr95].

Since we do not expect the version of the meta-metamodel presented in this paper to be the final one, we will use natural language expressions to describe additional constraints.

2.3 Notation of the Metamodelling Language

The notation used for designing metamodels is much like the one already used for the meta-metamodel, except for a few additional symbols. In order to give a more precise, though not intended to be formal, specification of the notation, we will differentiate between the graphical representation and the textual designators/annotations. For the latter we use a Backus-Naur form. The bold faced non-terminal symbols are used within the graphical illustration of the notation (see fig. 3). Notice that we do not bother with specifying a few basic non-terminal symbols - like *String*, *LowercaseLetter*, *UppercaseLetter*, *LineFeed* etc. An expression that satisfies the syntactic rules imposed by the OCL is represented by the symbol *OCLExpression*, while *GRALExpression* represents propositions in GRAL.

Basic Symbols

```
<digit> ::= 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
<positiveInteger> ::= {< digit >}
<infiniteNumber> ::= ’*’
<separator> ::= ’, ’
<lowerString> ::= <LowercaseLetter> <String>
<upperString> ::= <UppercaseLetter> <String>
```

Multiplicity

<maxCardinality> ::= <PositiveInteger> | <infiniteNumber>
<minCardinality> ::= <PositiveInteger>
<multiplicity> ::= <minCardinality> separator <maxCardinality>

MetaEntity

<entityName> ::= <upperString>

Attribute

<attributeName> ::= <lowerString>

Constraint

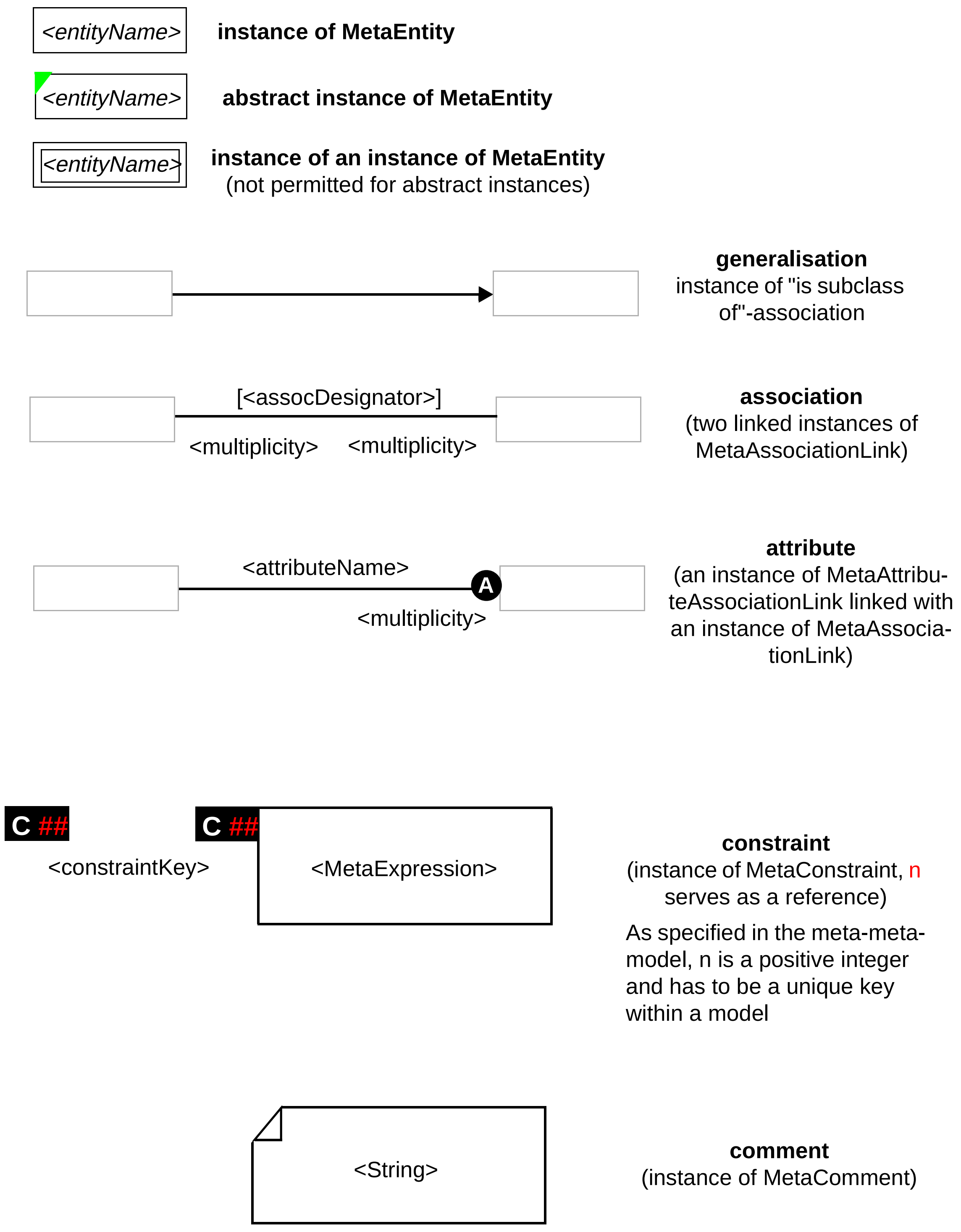
<constraintkey> ::= 'C' <number>
<MetaExpression> ::= <OCLExpression> | <GRALExpression> | <String>

Association

<backwardArrow> ::= '◀ '
<forwardArrow> ::= ' ▶ '
<designator> ::= <lowerString>
<backwardDesignator> ::= <backwardArrow> <designator>
<forwardDesignator> ::= <designator> <forwardArrow>
<forwardFirst> ::= <forwardDesignator> [< LineFeed> <backwardDesignator>]
<backwardFirst> ::= <backwardDesignator> [< LineFeed> <forwardDesignator>]
<assocDesignator> ::= forwardFirst | backwardFirst

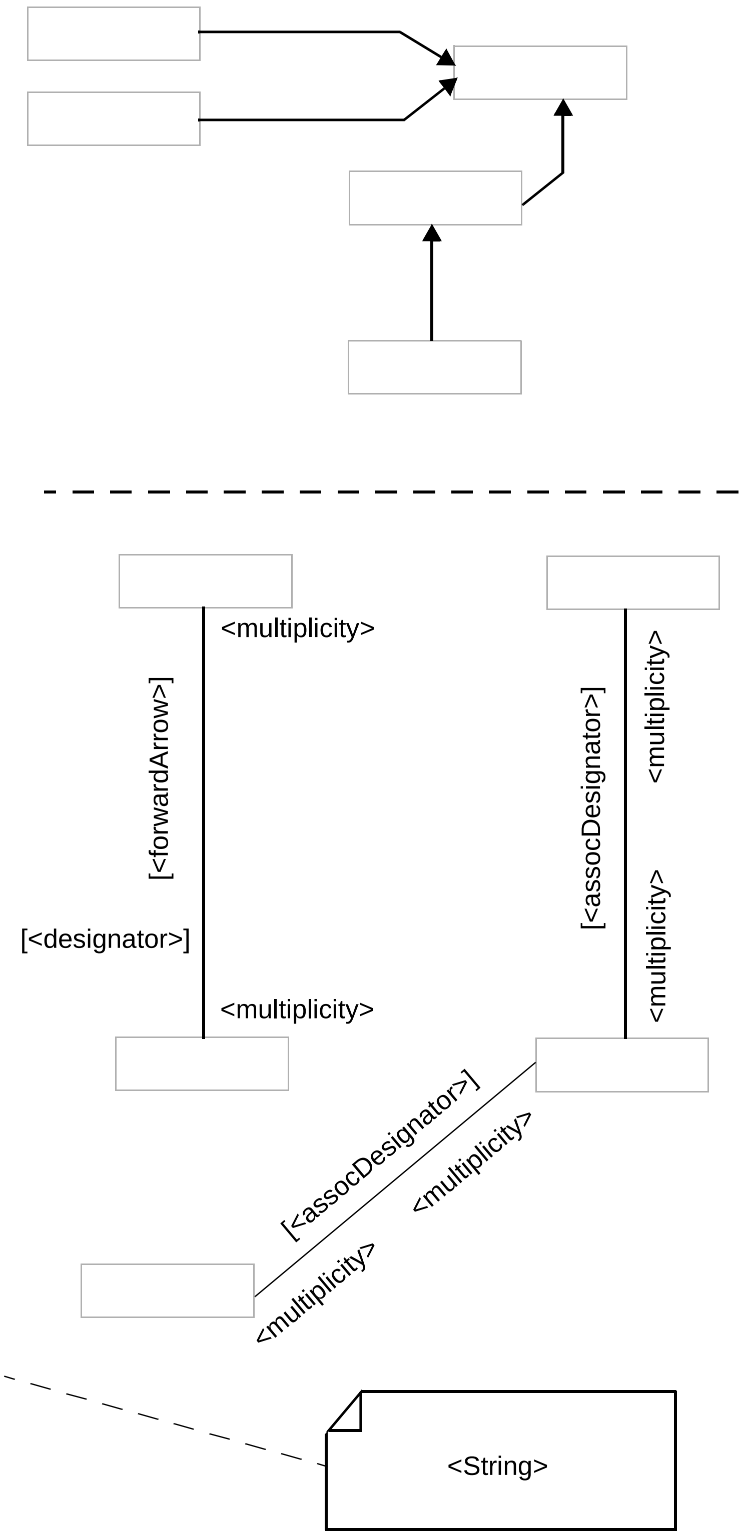
From an abtract point of view, the graphical notation of the language defined within the meta-metamodel can be regarded as a graph consisting of vertices which are connected by edges. The vertices represent instances of MetaEntity. They are rendered as rectangles. Abstract instances of MetaEntity are rendered by a filled triangle in the upper left corner of the rectangle. Sometimes it is required to use an instance of an instance of MetaEntity within a metamodel. Such an instance can be rendered by a double line rectangle. Generalisation is rendered by a line with an arrow at that end that connects to the generalized instance of MetaEntity. Associations are rendered as non directed lines. In case an associated entity is meant to be an attribute, the letter "A" has to placed at the end of the line that connects to that entity. Both, for generalisations and associations lines with different shapes are possible: Any line should be composed of a number of straight lines connecting exactly two rectangles. The lines representing associations or attributes may carry a designator (assocDesignator, attributeName). It is recommended to print the designator in parallel to the corresponding line, with the arrow indicating the direction in which the designator ought to be read. However, it may be inconvenient, if not impossible, to follow this recommendation. Therefore, alternative ways to attach a designator to a line are possible (see fig. 4). The same rule applies to multiplicities: They should, but do not have to be printed in parallel to a line representing an association or attribute. While it is recommended to print a designator above a line, multiplicities should be printed below a line.

The text that represents a constraint has to be placed in a rectangle that has an optinional identifier at the upper left corner. This identifier is used to link the constraint to the corresponding part of the model. The rectangles that contain comments may also be carry an identifier. As an alternative, a comment can also be assigned to a part of a model by using a dotted line.



<entityName>
instance of MetaEntity
<entityName>
abstract instance of MetaEntity
<entityName>
instance of an instance of MetaEntity
(not permitted for abstract instances)
generalisation
instance of "is subclass of"-association
[<assocDesignator>]
<multiplicity>
<multiplicity>
association
(two linked instances of MetaAssociationLink)
<attributeName>
A
<multiplicity>
attribute
(an instance of MetaAttributeAssociationLink linked with an instance of MetaAssociationLink)
C ##
<constraintKey>
C ##
<MetaExpression>
constraint
(instance of MetaConstraint, n serves as a reference)
As specified in the meta-meta-model, n is a positive integer and has to be a unique key within a model
<String>
comment
(instance of MetaComment)

Fig. 3: Notation of MEMO Metalanguages (1): Basic Symbols



generalisation
In principle, generalisation can be represented by any line with an arrow that points to the general entity.

association
The rule that applies for the representation of associations is similar to that for generalisation: In principle, any line that connects two entities is suitable. It is recommended, but not mandatory, to render the designators and multiplicities in parallel to a line.

comment
As an option, a comment can be assigned to a specific part of a model by a dotted line.

Fig. 4: Notation of MEMO Metalanguages (2): Additional Symbols and Illustration of Alternative Representations

3. Relationship to other Meta-Metamodels

As already mentioned above, a meta-metamodel fosters the integration of different metamodels which are instantiated from it. For this reason, it is a good idea to limit the number of meta-metamodels. At present time, two meta-metamodels have gained an outstanding popularity in the area of object-oriented modelling languages. The first one is the meta-metamodel designed for the CASE Date Interchange Format (CDIF), defined by the CDIF division within the Electronic Industries Association (EIA, ([Ern97]). It is the purpose of CDIF to support the exchange of models managed by CASE tools. Although the CDIF meta-metamodel and the metamodels instantiated from it seem to be rather elaborated, we do not think CDIF offers a satisfactory level of abstraction - like it is appropriate for the specification of an object-oriented modelling language.

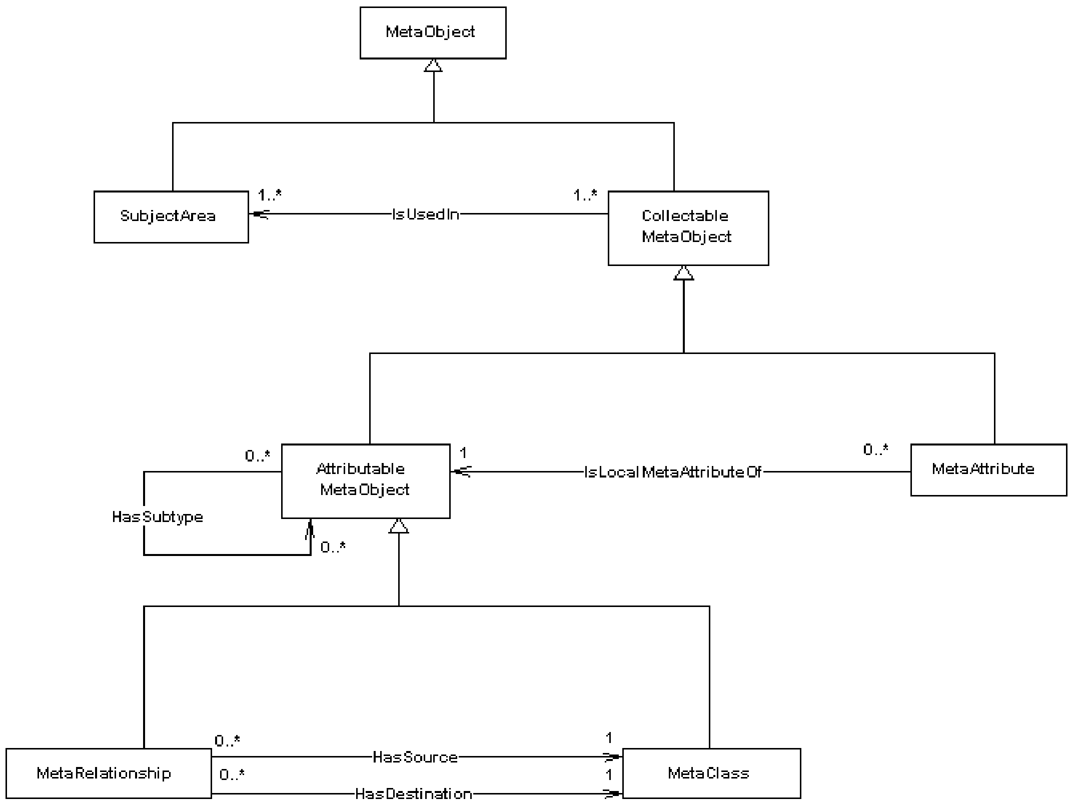


Fig. 5: The CDIF Meta-Metamodel - adapted by Platinum. The original version uses the designator "MetaEntity" instead of "MetaClass" ([Pla97], p. 23).

For instance: At first sight, the CDIF meta-metamodel does not include an entity like "Constraint" or "Comment". They are, however, included, as "MetaAttribute" ("Constraints", "Description") of "MetaObject" ([Pla97], pp. 27) - both specified by a data type. "MetaAttribute" which is used to instantiate attributes on the metalevel is also restricted to represent instances of "DataType" ([Pla97], p. 32), instead for allowing a higher level of abstraction as it would be offered by the notion of class. Furthermore, CDIF is clearly focusing on tool interoperability. For this reason, the CDIF metamodels include information that is typically required for the management of models within a tool, like "DateCreated", "DateUpdated", "TimeCreated", etc. ([Pla97], p. 52). Although these concepts are not reflected directly in the meta-metamodel, it has to be taken into account that the abstractions suggested by CDIF are based on a scope which is different from the pure definition of modelling languages.

The meta-metamodel of the UML ([Rat97a], see fig. 6) is obviously stressing a more object-oriented view than the one that comes with CDIF. For instance: Attributes may be specified by a class (or a data type as well). Nevertheless, we do not think that this meta-metamodel is satisfactory either.

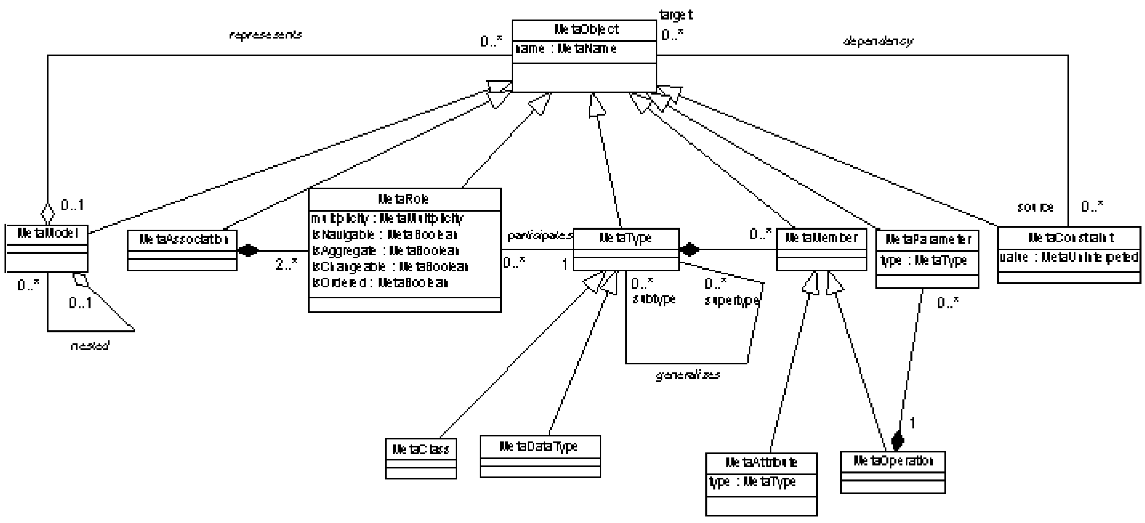


Fig. 6: The UML Meta-Metamodel ([Rat97a], p. 10)

Apart from the fact that [Rat97d] does not answer the question what it means exactly to apply a constraint to a constraint, concepts like "MetaOperation" and "MetaParameter" do not seem to be appropriate in a meta-metamodel of a modelling *language*. There is no doubt that concepts like operation and parameter are needed for object-oriented modelling languages. At that level of abstraction, they serve to describe object models which are eventually mapped to corresponding concepts of an object-oriented implementation language. On the implementation level, you will finally see objects which offer services that are executed by a machine. However, using operations in a language specification - i.e. within a metamodel - is misleading. It implies that the use of a language requires a machine - otherwise operations do not make much sense. Notice that operations may be very helpful whenever a model is managed by a tool. But that is a different context which should be clearly differentiated from metamodels of a language. We prefer the following view: Modelling languages should be described within metamodels that do not include any behaviour. On the other hand, it is a good idea to reconstruct and enhance a language description with an object model for a tool which is used to manage object models (see fig. 8).

While, for the reasons explained above, we decided to introduce a meta-metamodel especially for MEMO, it is certainly not completely different from existing ones. Although there are semantic differences, its concepts can be mapped to those of the UML and the CFIF meta-metamodels in a rather straightforward way. The following table shows how the concepts used within the three meta-metamodels correspond to each other. For a comparison of the UML meta-metamodel and the CDIF meta-metamodel see [Rat97c].

MEMO Meta-Metamodel	*UML Meta-Metamodel*	*CDIF Meta-Metamodel*
MetaObject	MetaObject	MetaObject
notation	not explicitely included; could be represented by an attribute of type MetaString on MetaObject	not explicitely included; could be represented by a meta-meta-attribute of type MetaString on MetaObject
MetaModel	MetaModel	SubjectArea
MetaEntity	MetaType	AttributableMetaObject
MetaEntity	MetaClass	MetaEntity
MetaEntity	MetaDataType	meta-meta-attribute Data Type on MetaAttribute
MetaAssociationLink	MetaAssociation	MetaRelationship
< out of scope >; could be added using instance of MetaComment	MetaRole	*< out of scope >*
MetaAssociationLink	MetaAssociation + MetaRole with isAggregate = true MetaAggregation MetaRelationship	MetaRelationship
MetaAssociationLink	MetaAssociation + MetaRole with isAggregate = true isChangeable = false source = 0..1 MetaComposition MetaRelationship	MetaRelationship
no corresponding abstraction	MetaMember	*no corresponding abstraction*
MetaAttributeAssociationLink	MetaAttribute	MetaAttribute
< out of scope >	MetaOperation	< out of scope >
< out of scope >	MetaParameter	< out of scope >
MetaConstraint	MetaConstraint	meta-meta-attribute Constraints on MetaObject
MetaComment	not explicitely included; could be represented by an attribute of type MetaString	meta-meta-attribute Description on MetaObject
< out of scope >	MetaDataType	meta-meta-attribute DataType on MetaAttribute
Boolean	MetaBoolean	DataType = Boolean
< out of scope >	MetaEnumeration	DataType = Enumerated

MetaExpression	MetaExpression	DataType = Text
MetaMultiplicity	MetaMultiplicity	DataType = String
MetaName	MetaName	DataType = Identifier

Integer	MetaNumber	DataType = Float
		DataType = Integer

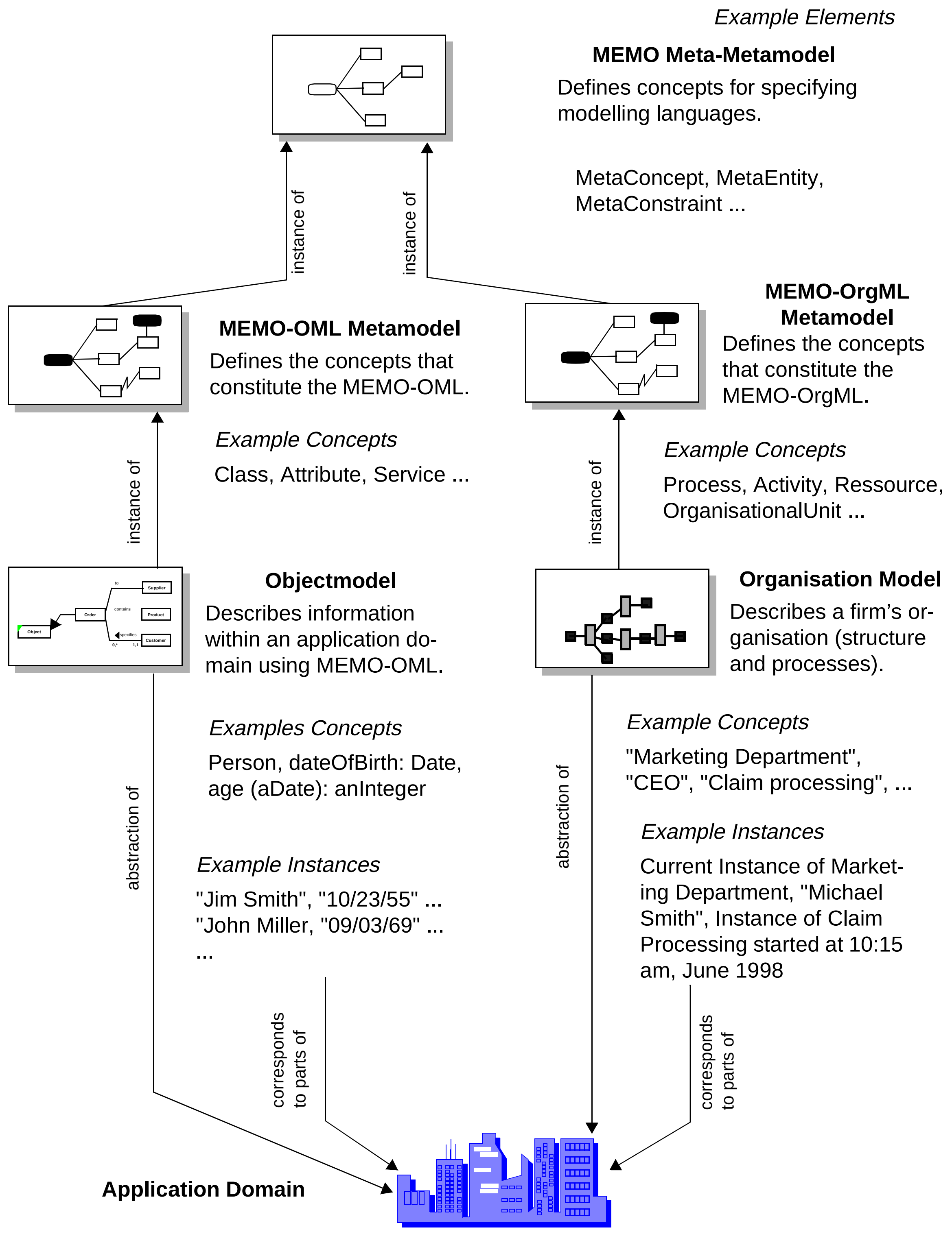
< out of scope >	MetaPoint	DataType = Point
String	MetaString	DataType = String

< out of scope >	MetaTime	DataType = Time
		DataType = Date

< out of scope >	MetaUninterpreted	DataType = Text

4. Relationship to Metamodels within MEMO

The MEMO meta-metamodel can be instantiated in various ways. Each instance is a metamodel which serves to specify a specific modelling language. Fig. 7 illustrates the different levels of abstraction: Refering to two MEMO modelling languages, the MEMO Object Modelling Language (MEMO-OML) and the MEMO Organisation Modelling Language (MEMO OrgML), it shows examples of concepts used on the various levels of abstraction between a real world domain and the meta-metamodel.



Example Elements

MEMO Meta-Metamodel

Defines concepts for specifying modelling languages.

MetaConcept, MetaEntity, MetaConstraint ...

MEMO-OML Metamodel

Defines the concepts that constitute the MEMO-OML.

Example Concepts

Class, Attribute, Service ...

MEMO-OrgML Metamodel

Defines the concepts that constitute the MEMO-OrgML.

Example Concepts

Process, Activity, Ressource, OrganisationalUnit ...

Objectmodel

Describes information within an application domain using MEMO-OML.

Examples Concepts

Person, dateOfBirth: Date, age (aDate): anInteger

Example Instances

"Jim Smith", "10/23/55" ...
"John Miller, "09/03/69" ...
...

Organisation Model

Describes a firm's organisation (structure and processes).

Example Concepts

"Marketing Department", "CEO", "Claim processing", ...

Example Instances

Current Instance of Marketing Department, "Michael Smith", Instance of Claim Processing started at 10:15 am, June 1998

Application Domain

Fig. 7: Levels of Abstraction

It is an essential goal of MEMO to allow for a tight integration of various models of an enterprise. Integration of different models is accomplished through common concepts. From our point of view, the level of integration increases with the level of semantics incorporated in those concepts. On a low level of integration, common concepts would be general basic types, like Byte, Character or Integer. MEMO fosters a higher level of integration: The various metamodels of different modelling languages share common concepts. Fig. 8 gives an example of this kind of integration.

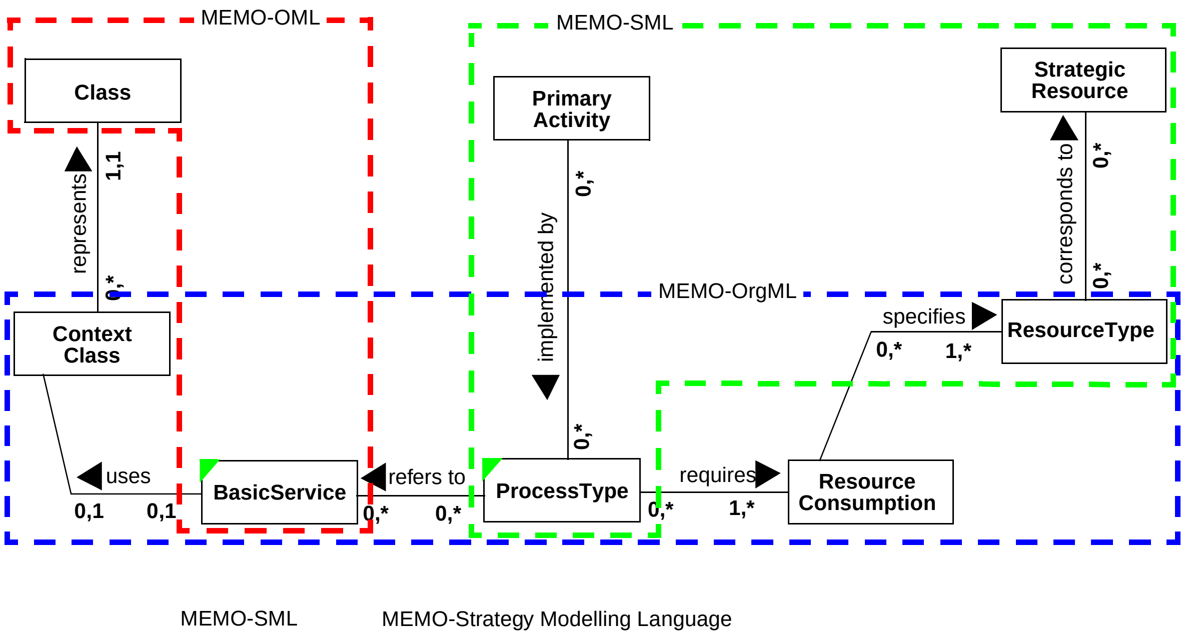


Fig. 8: Integration of Modelling Languages via common Concepts

The excerpts of the MEMO-OML metamodel shown in fig. 9 and 10 illustrate how to use the concepts defined in the meta-metamodel for the specification of a modelling language.

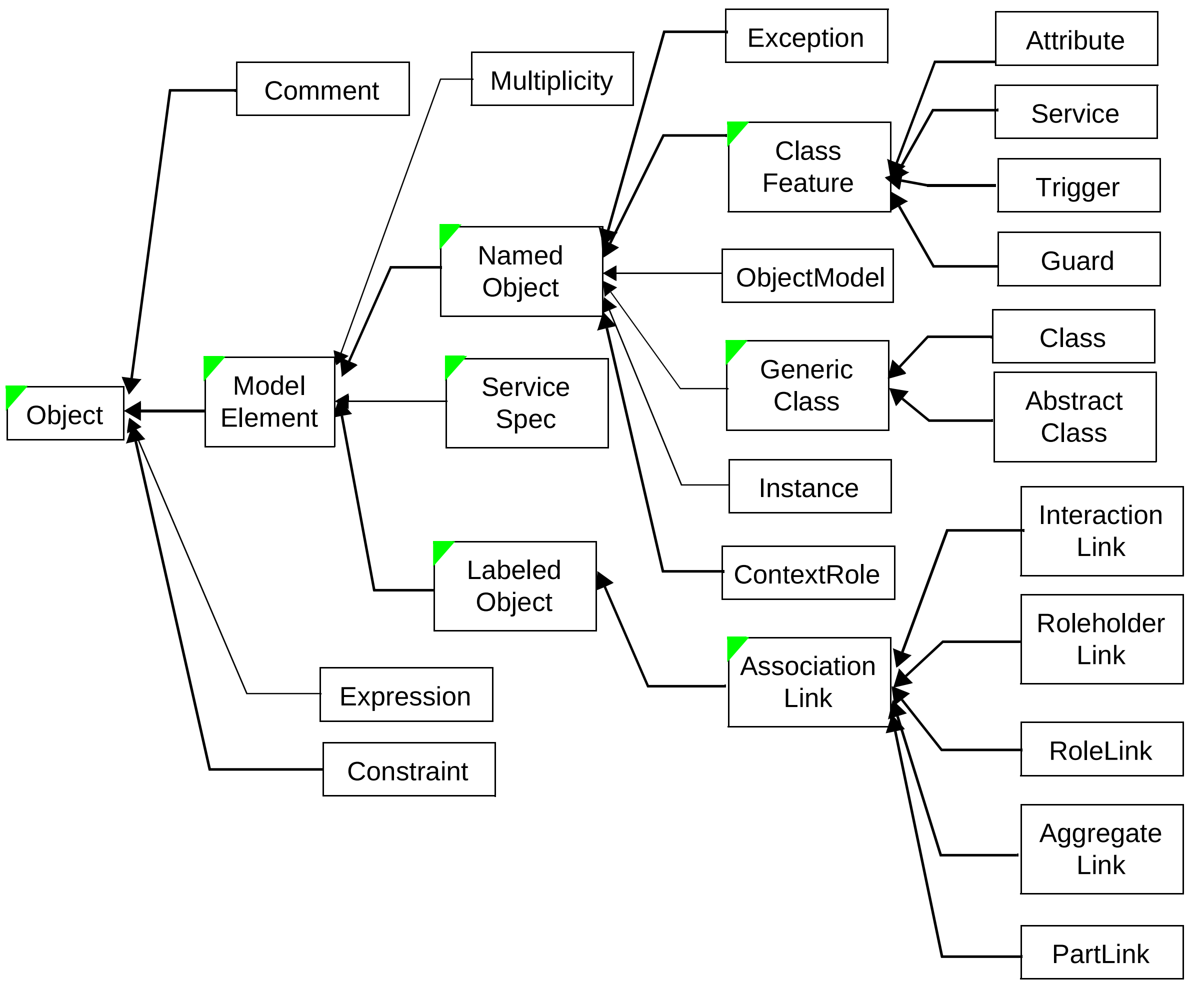
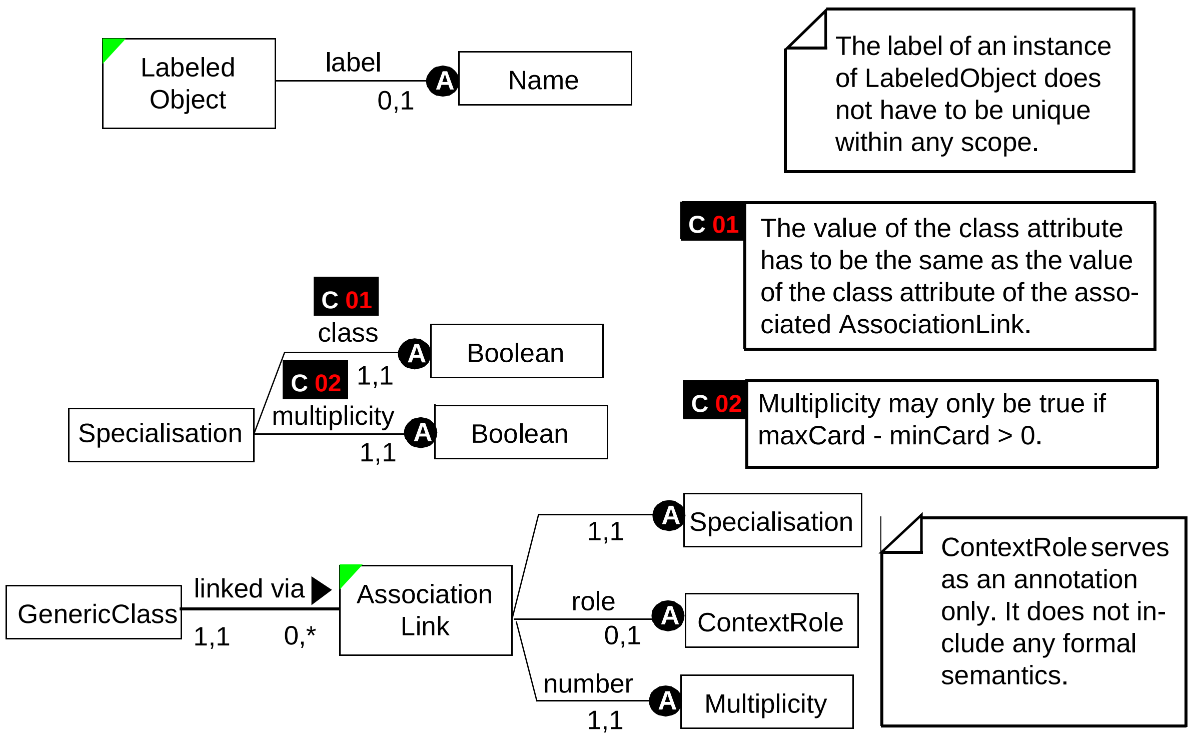


Fig. 9: MEMO-OML metamodel (1): Generalization Hierarchy



Labeled Object
label
0,1
A
Name
The label of an instance of LabeledObject does not have to be unique within any scope.
C 01
The value of the class attribute has to be the same as the value of the class attribute of the associated AssociationLink.
C 01
class
A
Boolean
C 02
1,1
multiplicity
A
Boolean
Specialisation
1,1
C 02
Multiplicity may only be true if maxCard - minCard > 0.
1,1
A
Specialisation
GenericClass
linked via
1,1
0,*
Association Link
role
A
ContextRole
0,1
number
A
Multiplicity
1,1
ContextRole serves as an annotation only. It does not include any formal semantics.

Fig. 10: MEMO-OML metamodel (2): Basic concepts to describe associations

5. Concluding Remarks

The MEMO meta-metamodel presented in this paper provides a semi-formal description of the concepts that are used to specify various graphical modelling languages within the MEMO framework. The meta-metamodel itself is specified by a small number of fairly simple concepts. While those concepts are inspired by the ontological notions of entities and associations, they are not independent from their purpose, namely the specification of specialised modelling languages. Therefore, further investigations will be necessary to confirm or refine the current meta-metamodel. We expect this to be a process with multiple levels of feedback: The ongoing specification of modelling languages will show whether or not the concepts provided by the meta-metamodel are sufficient. Applying the modelling languages for describing real world domains will help to find out whether the concepts on this level are sufficient and appropriate. At this stage, we do not regard it as necessary to formalize the meta-metamodel. However, we plan to specify a formal foundation of the meta-metamodel in order to improve the means to check the consistency of metamodels.

References

[EbFr95] Ebert, J.; Franzke, A.: A Declarative Approach to Graph Based Modeling. In: Mayr, E.; Schmidt, G.; Tinhofer, G. (Eds.): Graphtheoretic Concepts in Computer Science. Berlin, Heidelberg etc.: Springer (LNCS 903) 1995, pp. 38-50

[EbSü97] Ebert, J.; Süttenbach, R.; Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. In Olive, A., Pastor, J. A. (Eds.): Advanced Information Systems Engineering, Proceedings of the 9th International Conference, CAiSE'97. Berlin, Heidelberg etc.: Springer (LNCS 1250) 1997, pp. 203-216

[EbWi96] Ebert, J; Winter, A; Dahm, P.: Graph Based Modelling and Implementation with EER/GRAL. Fachberichte Informatik, Universität Koblenz-Landau, Heft 11, 1996

[Ern97] Ernst, J.: Introduction to CDIF. 1997 (http://www.cdif.org/)

[Fra97] Frank, U.: Enriching Object-Oriented Methods with Domain Specific Knowledge: Outline of a Method for Enterprise Modelling. Arbeitsberichte des Instituts fuer Wirtschaftsinformatik, Nr. 4, Koblenz 1997

[Frz97] Franzke, A.: GRAL 2.0: A Reference Manual. Fachberichte Informatik, Universität Koblenz-Landau, 1997

[Pla97] Platinum: Object Analysis and Design Facility Response to OMG/OA&D RFP-1. (http://www.omg.org/library/schedule/AD_RFP1.html)

[Rat97a] Rational: UML-Semantics. Version 1.1. 09/01/1997 (http://www.rational.com)

[Rat97b] Rational: UML-Notation Guide. Version 01.09/01/1997 (http://www.rational.com)

[Rat97c] Rational: Appendix M3: UML Meta-Metamodel Alignment with MOF and CDIF. Vers. 1.0, 13 January 1997 (http://www.rational.com)

[Rat97d] Rational: OCL. Version 1.1. 09/01/1997 (http://www.rational.com)