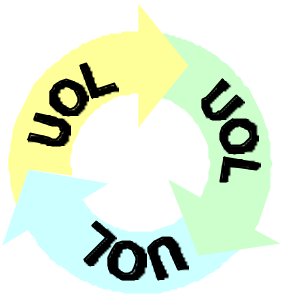


Universal Object Language 1.2 Specification

Authors

Recerca Informàtica
Daimler-Benz Research and Technology

OMG Document: ad/98-07-07
Version 1.2 /T-UOL-19980707
July 7th, 1998

The most recent updates of the Universal Object Language can be found, via worldwide web, at: http://www.recercai.com

Primary Contacts for the UOL submission:

Recerca Informàtica	Joan M. Moral	uol@arrakis.es
Daimler-Benz Research and Technology	Mario Jeckle	mario.jeckle@dbag.ulm.DaimlerBenz.COM

Table of Contents

The table of Contents contains entries for both the Specification and the Appendices.

1 Overview

1.1 Introduction

The following companies are pleased to co-submit to the OMG ad/97-12-03 RFP: Stream-based Model Interchange Format the Universal Object Language 1.2 specification (hereafter referred to as the *UOL*):

- Recerca Informàtica, SL
- Daimler-Benz Research and Technology

In accordance to the RFP the main purpose of UOL is to:

- Establish an industry standard specification for a stream-based model interchange format,
- Provide a generic format that can be used to transfer a wide variety of models,
- Demonstrate that it can be used to exchange OMG Object Analysis and Design Facility (OADF) compliant models (UML based) and models compliant to other MOF-compliant meta-models and extensions (e.g. Workflow Management Facility and Business Object Facility meta-models), and
- Leverage existing vendor-neutral transfer formats as much as possible.

This submission mainly consists of:

- The Universal Object Language Specification
- The XML-DTDs
- Mappings
  - The mapping between UOL and MOF
  - The mapping between UOL and CDIF
  - The mapping between UOL and STEP/EXPRESS
- Additional Specification
  - The mapping between UOL and UML with UML constructs
- Supporting Appendices

This submission defines these standards and provides proof of concept that covers key aspects of the RFP.

1.1.1 Rationale

UOL is a human readable format that allows representing MOF and UML models in a very compact way and easily learnable and usable by any person. The companion XML-DTDs allows publication of UOL in an industry standard format and facilitates its use by all XML supporting tools.

There are several reasons that make UOL a good transfer format:

- It is high level and thus clearly compact.
- It supports the transfer between any pair of tools in batch and real-time.
- It supports the human-tool transfer.
- It supports human processing.

The first reason is easily supported by a quick scan through the UOL grammar, it has constructs with a direct mapping to the MOF ones that avoid the assembler-like look of other transfer languages.

UOL supports the transfer between tools; in fact text is the most easily transferable format. The tool only needs to have a parser, as the one provided within this proposal, to translate the information into its particular format. UOL can also be easily generated by a simple traverse of the information contained in the repository. As a proof of concept this process has been done by Recerca Informàtica: a repository is queried for its contents, or parts of it, and the repository returns a UOL stream with the required information. Then, this stream can be introduced again in the repository, the same or a new one. Inserting UOL text into the repository is also a very easy process: the parser receives the UOL stream and populates the repository with the contained information.

The human readability and processing of UOL text is a consequence of its high level nature. The constructs of UOL evolve from analysis and design concepts and Eiffel-like syntax, therefore a software engineer should be able to read and use the UOL as easily as the graphic notation.

1.1.2 Goals and objectives

The objective of this submission is to specify the Universal Object Language (UOL) and its companion XML-DTDs as the Stream-based model interchange format.

The main goal of the UOL is to provide a format allowing for the most efficient batch or real-time transfer between tools. To allow for this efficiency the format must allow a very compact representation of the MOF and UML models and the MOF meta-data. The UOL, with is rich semantics, offers this compact representation while at the same time being human readable.

Human readability is another goal of the UOL. There are several reasons for this goal. Output of a tool, in many situations, is susceptible of being analyzed by a human before inputting it to another tool. A very special and important case of this is round-trip engineering, which we will discuss further on in this document. Humans use other tools than CASE tools such as word processors or compilers, and the transfer format should also be valid for these tools. Being that the only format accepted universally by any tool is the human readable text the UOL assumes this requirement.

Another goal of this proposal is to allow publishing in the XML format parts of a repository. Given the importance of XML, the availability of many tools and products based on XML and the reasonable need to make the repository accessible in this format we have developed the necessary DTDs for UOL.

Although the MOF allows for many meta-models, UML is the first one supported and the main meta-model CASE tools will support. Therefore, one final goal of this proposal is to simplify and optimize transfer of UML models. This simplification and optimization is obtained extending the basic UOL with UML constructs giving the most compact possible transfer format. This extension is proposed as non mandatory or compliant.

Additional benefits of this extension are having a UML textual representation and being a valid alternative to graphical UML for visually impaired individuals. These two features are two milestones in the OOA&D Task Force road map.

1.2 Structure of This Submission

This section briefly describes the major portions of the Universal Object Language (UOL) submission. The submission is made up of two documents. The first is this document titled Universal Object Language 1.2 Specification. The UOL Specification section gives the UOL specification and the XML DTDs that are being proposed for standardization. The submission also includes several mappings:

- UOL-MOF basic requirement of the SMIF RFP
- UOL-CDIF to allow transferring of repositories in CDIF format to UOL
- UOL-STEP/EXPRESS to allow transferring of STEP/EXPRESS models to MOF based repositories and vice-versa.

This submission also includes an additional specification to the SMIF RFP: direct support of the UML meta-model constructs to simplify and allow the utmost compact transfer of UML models.

The second document is the Universal Object Language Appendices, which describes the grammar of UOL in different formats (BNF format with and without UML constructions), the WSN format and the character set used in UOL mapped to the different more used representations.

1.2.1 Universal Object Language submission Overview

The UOL specification section describes the UOL, the XML-DTDs, mappings and an additional specification to the SMIF RFP.

1.2.1.1 Overview

Provides an introduction to the Universal Object Language submission. The key RFP requirements summary and how this submission addresses the RFP requirements is addressed. The key contributors to the specification are acknowledged.

1.2.1.2 Facility Purpose and Use

Viewing the SMIF as a communication language, this chapter reviews the different types of communications needs that exist. It then describes why a textual OO full life-cycle language, such as UOL, fulfills all these needs.

1.2.1.3 The Universal Object Language Specification

Together with the next section this is the main portion of the UOL submission. The lexical and syntactic specification is given. Semantics are described and examples are given of each construct.

1.2.1.4 The XML-DTDs

Together with the previous section this is the main portion of the UOL submission. This chapter justifies the need of an XML representation of UOL and describes the mapping between UOL and XML with DTDs giving several examples.

1.2.1.5 Mappings

Describes the mappings between UOL and several languages and models. More precisely, it describes the UOL-MOF mapping, basic requirement of the SMIF RFP. It also describes the UOL-CDIF to allow transferring of repositories in CDIF format to UOL to preserve previous investments of tool builders. Finally, a mapping between UOL and STEP/EXPRESS is presented following the SMP RFP requirement to allow transferring of STEP/EXPRESS models to MOF based repositories and vice-versa.

1.2.1.6 Additional Specification

An additional specification to the SMIF RFP is given to support the UML meta-model constructs to simplify and allow the utmost compact transfer of UML models. Additional advantages are being able to use UOL as a round-trip engineering language, a UML textual language and an alternative to graphics for visually impaired individuals. These last benefits are milestones in the OOA&D road map.

1.2.2 Universal Object Language (UOL) Appendices

This section describes the various appendices that support this specification. This information includes the grammar of UOL in different formats: BNF format with and without UML constructions and the WSN format. The character set used in UOL with the different more used representations is attached.

1.2.2.1 Appendix A: UOL Grammar without UML constructions

This section gives the UOL grammar in a BNF format, without the UML constructions, i.e. UOL independent on any Meta-model, only with MOF as it Meta-meta-model.

1.2.2.2 Appendix B: UOL Grammar with UML constructions

This section gives the UOL grammar in a BNF format, with the UML constructions, i.e. UOL with an extension of UML constructions for give an upward compatibility with UML based models.

1.2.2.3 Appendix C: UOL grammar in WSN format

This section gives the UOL grammar in WSN format. This format is used in the STEP/EXPRESS mapping.

1.2.2.4 Appendix D: ASCII, UNICODE and ISO 10646 Character Set

Being that the proposal shall support use of international standard code-sets, a code chart of the most important code-sets is attached.

1.3 Resolution of Requirements

This section describes how this submission meets the key requirements identified in the RFP.

1.3.1 Mandatory Requirements

RFP Requirement	How submission addresses the requirement
Use the MOF as its meta-meta-model.	UOL has MOF as its meta-meta-model and it describes all MOF concepts in addition to UML and other object engineering related concepts. **The Section 4.1 addresses this requirement by detailing the complete mapping between MOF and UOL.**

RFP Requirement	How submission addresses the requirement
Provide a complete specification of the syntax and encoding needed to export/import models and meta-model extensions included in-line as part of the transfer stream. This syntax and encoding shall have an unambiguous identification to support evolution of this technology.	UOL provides an extension mechanism based on stereotypes and tagged values. This mechanism provides the required meta-model and model extension capabilities. **The Section 4.1 addresses this requirement.**
Provide a means for unambiguous identification of any concept specified in a MOF-compliant meta-model that is referenced (but the specification is not included) in a transfer stream.	All concepts expressed in MOF are unambiguously expressed in UOL. This allows using UOL as transfer format for any MOF based solution. **The Section 4.1 addresses this requirement by detailing the complete mapping between MOF and UOL.**
Demonstrate support for import/export of UML models and the UML meta-model. This demonstration shall include demonstration of a round-trip model exchange without information loss.	The SMIF RFP requires that the interchange format should be independent of the semantic constructs in a meta-model. **The Section 4.2.1 addresses this requirement.** However, UOL also has a direct support of the UML constructs because of its important benefits. This support is proposed as non-mandatory. **Sections 4.2.2 and 8.2 address this proposal.**
Support use of international standard code-sets.	UOL supports the use of UNICODE and ASCII as its code-set. **The Section 3.2 addresses this requirement.**

1.3.2 Optional Requirements

RFP Requirement	How submission addresses the requirement
A compact data representation in addition to the text-based representation as an alternative to the interface-based representation defined in the MOF.	UOL is intended to serve as model communication language in several situations where humans may be involved. Therefore it includes full UML and other object engineering concepts support. Situations in which a more concise form is required may appear. To solve this scenario UOL includes also a more compact representation. **The Section 3.2 addresses this requirement.**
Upward-compatibility with the EIA/CDIF 1994 (CDIF94) Transfer Format standards.	UOL has been developed to express object analysis and design concepts in a seamless way. To achieve this goal we have developed a new format instead of trying to extend a non-Object-Oriented existing one. However, to protect the investments already done in CDIF, a mapping between CDIF and UOL and a conversion utility have been developed. **The Section 4.3 addresses this requirement.**
Contain an unambiguous, complete mapping of the concepts in the CDIF94 meta-meta-model to the concepts in the MOF.	To protect the investments already done in CDIF a mapping between CDIF and MOF has been developed. There is a mapping between CDIF-UOL and one between UOL and MOF, therefore, the mapping CDIF-MOF is via UOL.

Identify the impact of the proposed SMIF specification on transfer files produced using the CDIF94 Transfer Format standards. This includes identification of any changes to CDIF transfer files required to produce valid syntax and encoding per the proposed SMIF specification. This requirement may be met by providing a specification for a conversion utility for transfer files created using the CDIF94 Transfer Format standards to make them compliant with the proposed SMIF specification.	UOL does not require any change to CDIF. However, since UOL is not an extension of CDIF a mapping between CDIF and UOL and a conversion utility have been developed. **The Section 4.3 addresses this requirement.**
Provide transfer stream examples that use concepts from other industry standard meta-models.	To allow UOL to support STEP/EXPRESS a mapping and conversion utilities have been developed. Examples are included in this proposal. **The Section 4.6 addresses this requirement.**
Identify specific modeling language differences between EXPRESS and the MOF/UML and discuss ways to map between these languages	**The Section 4.6 addresses this requirement.**
Identify the impact of the proposed SMIF specification on existing schema definitions and transfer files produced using STEP EXPRESS. This may include identification of any changes to STEP EXPRESS files required to produce valid syntax and encoding per the proposed SMIF specification. Submissions may include a specification for converting STEP schemas and/or transfer files created using STEP EXPRESS standards to make them compliant with the proposed SMIF specification	**The Section 4.6 addresses this requirement.**

1.4 Resolution of RFP Issues to be Discussed

RFP Issue	How submission addresses the issue
Meta-Object Definition Language (MODL)	There is not any connection with the MODL.
Object Constraint Language (OCL)	OCL is embedded in UOL as formal constraint language. In addition free text constraints are also allowed. **The Section 3.1 addresses this requirement.**
Support semantic interoperability between tools that share and manipulate STEP schemas and STEP schema instances in addition to tools that support sharing and manipulation of OAD models.	**The Section 4.6 addresses this requirement.**

Include information on how to perform conformance tests (for checking syntax and transfer stream specific validation rules for schemas and schema instances) on transfer streams prior to import into other applications. This may include recommendations for adding additional functionality to the MOF to satisfy transfer file conformance test requirements identified by the STEP community. Proposals should discuss an approach to address this difference in problem scope. For example, proposals may describe how to use the MOF to describe STEP schemas at the same level as the UML meta-model.	**The Section 4.6 addresses this requirement.**
The connection, if any, between the proposed transfer format syntax and encoding and the Objects-by-Value syntax and encoding.	There is not any connection

1.5 Business Requirements

1.5.1 Copyright Waiver

In the event that this specification is adopted by OMG, the submitters grant to the OMG, a non-exclusive, royalty-free, paid-up, worldwide license to copy and distribute this specification document and to modify the document and distribute copies of the modified version. For more detailed information, see the disclaimer on the inside of the cover page of this submission.

1.5.2 Proof of Concept

In order to test the UOL concept it has been integrated in a CASE tool under development. The tool is capable of exporting and importing UOL code without any loss of information. The process is quite straightforward; the repository is queried for its contents in UOL format. The resulting stream can be redirected towards a file or a TCP/IP connection. The resulting model can be edited or processed in different ways (i.e. user implemented metrics or rule checkers) and then re-imported in order to keep the repository up to date. This is an example of the round-trip capacities of UOL but, of course, the stream can be presented to another repository to make a duplicate of the original one.

The implementation of such process has been done through a parser integrated within the tool. The parser receives the UOL code and translates it into the specific repository format. The generation of the UOL code has been done with a traverse utility that generates the UOL code for each element it found in the repository.

1.6 Acknowledgements

The following section lists the team members that worked on the UOL submission during the initial and revised submissions. The members of the core team that designed and influenced the UOL model are listed below. The primary contact in each company is listed first.

1.6.1 UOL Co-Submitters

Recerca Informàtica	Joan M. Moral	uol@arrakis.es
	Josep Oncins	recercai@arrakis.es
	Teresa Masot	library@arrakis.es
	Albert Sorroche	e4005868@est.fib.upc.es
	Guillem Vallès	e6745766@est.fib.upc.es
Daimler-Benz Research and Technology	Mario Jeckle	mario.jeckle@dbag.ulm.DaimlerBenz.COM

1.6.2 UOL Supporters

Telefónica I+D	J.Hierro	jhierro@tid.es
Universitat Politècnica de Catalunya	Allen Peralta	peralta@lsi.upc.es

1.6.3 Additional Contributors and Supporters

The co-submitters of the UOL submission appreciate the contributions and support of the following individuals during the UOL submission and evaluation process:

Alicia Ageno (Universitat Politècnica de Catalunya), Grady Booch (Rational, Inc.), Derek Coleman (Hewlett-Packard Laboratories), Xavier Escudero (Recerca Informàtica, SL), Antoni Gonzalez (ICT Electronics), Brian Henderson-Sellers (University of New South Wales), Ivar Jacobson (Rational, Inc.), Bertrand Meyer (ISE, Inc.), James Odell (Intellicorp, Inc.), Horacio Rodriguez (Universitat Politècnica de Catalunya), Jordi Rosell (Tao, SA), James Rumbaugh (Rational, Inc.), Joan Serras (Aceri, SA).

NOTE: In some cases, the individuals are methodologists whose writings and lectures influenced and helped the creation of this UOL submission or were reviewers of the first version of UOL (Eiffel+), developed as a PhD thesis.

2 Facility Purpose and Use

2.1 Introduction

The UOL is intended to support a wide range of usage patterns and applications. This capability comes about because UOL is a textual OO full life-cycle language. Understanding what is a textual OO full life-cycle language will allow us to understand is usefulness in a wide range of scenarios.

2.2 What is a Textual OO Full Life-cycle Language?

We define a textual OO full life-cycle language as an object engineering language that is capable of describing all OOAD constructs and concepts and conceptually being executable.

Naturally, when we say when we say *all OOAD constructs and concepts* we should refer to who's definition of OOAD. Happily, OMG's initiative to standardize an OOAD modeling language allows us to define UOL based on OMG's UML 1.1 standard.

In 1994, the UOL co-author, Allen Peralta, developed an OO full life-cycle language based on Eiffel[1] as his thesis. This language was called Eiffel+[2] and was finished in March 1995 and reviewed by Bertrand Meyer that same year. The initial idea was to have a textual language that would allow describing all OOAD elements, generally accepted at that time, and to obtain different products at each development phase. It also assumed that graphical languages where neither adequate for all tasks nor for all people. Accordingly, a textual language that gave support to OOAD graphical languages could complement them in such a way that at each moment one could choose the best representation paradigm.

Being that Eiffel was, and is, considered one of the best-engineered languages, if not the best and certainly the most complete and easiest to learn, it was chosen as the basis on which to develop the new language. Developing a new language from scratch was discarded because of the difficulty of introducing a new language to the OO community and because it would seem that we already had a surplus of OO languages.

Most of the elements, defined in UML, were already present in 1995, and included in Eiffel+. Having much of the work already done and being that readability of programs is a must in any language the decision to further expand Eiffel+ into UOL supporting UML was immediate.

The transition from Eiffel+ to UOL has been very simple and most of the efforts have been in trying to make the language as simple as possible. Even though we have tried, to the utmost, to emulate Bertrand Meyer, we recognize we have added many more new keywords than we would like but it has seemed necessary for readability purposes.

[1] Eiffel is a registered trademark of NICE.
[2] The name Eiffel+ was used exclusively for the thesis.

2.3 Why we Need a Full Life-cycle Language

When a software engineer develops an object-oriented system he must describe a model using OO analysis, design and programming concepts. Models are, essentially, a way of communicating solutions to a problem. There are three types of communications that are necessary:

- person to person
- person to tool
- tool to tool

Person to person communication can be done verbally or through documentation. Verbal communication, although essential, resides outside the purpose of UOL.

Communication through documents requires maximum formalization to reduce misunderstandings to the least possible. Analysis, design and programming languages are a way of formalizing communications. This formalization is especially efficient if the languages are standardized and universally known. In this sense UML-MOF is an important step in this direction.

The models we create must be represented graphically and/or textually with tools, which may vary from a text editor to a CASE tool. If the tool supports the same concepts that must be used to describe the model the software engineer's task is much easier, allowing concentration on the problem instead of the means to make the description. Therefore, we need tools that support the standard OO analysis, design and programming concepts.

One of the important features of OO is its support of what is called "seamless transition". That is, that the same concepts are used throughout the whole life-cycle. However, to obtain this seamless transition it is not only necessary to use the same concepts at every stage but also to use at every stage tools that support the same concepts. The problem arises, naturally, that at some stages we are forced to use, to communicate with, tools that do not support OO concepts.

A full life-cycle language is therefore a language that allows us to use and to communicate always with the same OO constructs at any stage and with any tool or between any pair of tools.

Let us review the two types of communications in which a full life-cycle language can be used.

2.4 Person to Tool Communication

There are many tools a software engineer may use: compilers, editors, CASE, GUI builders, etc. In some cases it is possible for a person to communicate directly with an OO tool based on UML (e.g. a CASE tool) but in many other situations this is not the case.

Two examples of this might be:

- input to a compiler, even if a CASE tool has generated it, must be manipulated during debugging with a program editor
- creating analysis documentation extracted from the repository to a word processor

In this situation the seamless transition is not maintained unless we can continue to use our OO concepts even with a tool that does not support them. If we have maintained consistently our OO representation, once we finish working with our non-OO tool, we will be able to import the result of our work "seamlessly" to a tool supporting the OO concepts. To maintain this consistent view

and given that most of our work with non-CASE tools is done textually, what we need is to be able to have all our texts embedded in OO constructs and this is possible with a textual OO full life-cycle language such as UOL.

There is, however, an absolute requirement that UOL or any other textual OO full life-cycle must comply with to be effective in this situation: simplicity and ease of learning and use. Any software engineer should be able to learn and use the language in a few days.

There is one very special and important case of this need of tool-to-person and person-to-tool communication. We refer to round-trip engineering and the OO auxiliary tools industry, such as GUI builders and the component and framework industry.

Round-trip engineering will be explained in the next chapter.

The component and framework builders are, in our opinion, at least as important as tool builders (CASE or other). If software development is to be an engineering profession it requires the existence of a component industry. There is no engineering profession (mechanic, electronic, etc.) that relies on developing in-house all their pieces or materials. The existence of a component industry in inherent with the concept of an engineering discipline. There are two aspects that we must consider with respect to these tools.

The first is that, even though one may buy a component(s) or a framework without source code, it is the exception rather than the norm, at least at present. If we need to investigate how these components are built it is necessary to view them with all the enrichment that UML allows us, instead of looking only at the source code and separately at the AD models in a manual. If we require learning from them more than what we can see through programming language constructs it is necessary to embed in the code analysis and design constructs.

The second is that we need to connect the output of these tools with the rest of the models we are developing within the main CASE tool. Therefore, we need to allow importing the source code, and its corresponding OOAD model, generated by their products (GUI builders, component libraries, etc.) easily into the repository by CASE tools supporting UML.

Both reasons reflect that we have the need of round-trip engineering the source code with embedded AD constructs developed from these tools.

2.5 UOL as a Round-Trip Engineering Language

2.5.1 Why Round-Trip Engineering is Necessary

Software engineers consider using CASE tools for several reasons. One of the main reasons is to maintain only one description of their systems. That is to have, at all times, both models and source programs synchronized.

When developing a system, software engineers start by working with CASE tools to describe graphically and document the analysis and design. With a full life-cycle CASE tool they then proceed to generate code from their design. At this point they start doing testing and debugging with programming language compilers and text editors. Usually they will introduce changes that must be reflected in the designs stored in the CASE tool repository. Therefore, if they want to have both model and source synchronized, they must either re-import the source code restructuring the model or, at least, reflect in the model manually the changes that have been produced during the testing phase.

Source code can also be produced by other tools such as screen designers/painters, 4GLs, etc. This code or the part of the model that it represents should also be imported to the repository to reflect the complete design of the system.

Ideally importing the code and restructuring the model or reflecting the changes done to the model (if the CASE tool does not store the source code) should be done automatically. Doing this manually is very error prone and especially tedious. What happens, when not done automatically, in real-life stress situations, is that it is not done and there is a mismatch between model and program. Being the consequence of this situation evident for all and sufficiently documented we will not further describe it.

Although this situation is the one most discussed in the industry, one should also consider the inverse, especially in an iterative life-cycle as we usually apply in OO development. When we start a new iteration of the system being developed, our main efforts will start again at analysis and design. Naturally, when we increase our system with new aspects, we will introduce changes to the previous model. Since this model has been translated to source code and debugged, it will be necessary to reflect in the previous version of the source code the changes in the model. That is the precise inverse situation that we have previously described and it is mandatory that the model reflects exactly what is implemented in the program to be able to generate the necessary changes to the program or a new version of the program.

In order to fulfil the obvious need of facilitating synchronicity between model and code, it is necessary that CASE tool builders offer the following functionality:

- Generate source code
- Export source code changes from a modified model
- Import source code generated from other tools and the model it represents
- Import modified source code reconstructing the model

Only the first of these four tasks is trivial for CASE tool builders, generating code from the model. The second, exporting source code, is also trivial if the CASE tool stores the source code or is integrated with a Version/Configuration Management tool. If it isn't it may be more or less difficult depending on the environment the programmer uses.

The last two however, are inherently difficult for all CASE tool builders, no matter what language they support. Proof of this can be observed from CASE tool advertisements. Even though most companies offer support for many languages, they do so only exporting or generating. Importing or round-trip is offered only for a subset of them.

There are many reasons for this difficulty. In the first place, constructing an efficient parser is always a difficult task. If the parser is not fast the programmer will desist in using it. The second reason is that, we not only have to parse the source code but we also have to analyze it semantically, even if the code is correct from a compiler's point of view. When analyzing the source code, for re-import purposes, we must have some way of distinguishing features that are not supported by programming languages. Some of these features may be distinguishing attributes implementing associations from those of aggregations, or where have patterns been applied, or what parts of the system are in a module (in most languages), etc. Finally, interpreting the changes with respect to the original model can be also quite complex.

What most CASE tool builders do, is to enrich the generated source code with comments (mark-up code) that assist the round-trip tool in analyzing, from an OOAD point of view, the source code, to be able to modify the model stored in the repository. Although this way of focusing the solution

may be helpful, it does not totally solve the problem. The reason is that there are several types of changes that the programmer may introduce:

- Firstly, the programmer may delete something previously written. This type of change is the easiest of all and does not have any special difficulties.

- The second is that he may modify something written previously. This is much more difficult. How can we know that he has changed the cardinality of a relation, or that he now treats it as a different kind of relation or with a different constraint?

- The third is adding some new element. Detecting an attribute may be easy but not so much if it is used to implement a relation. And what if he modifies something affecting a use-case?

- Lastly, what happens if during testing he inadvertently changes or deletes some of the comments generated by the tool? How can we interpret the source code then?

And with respect to source code, generated by GUI builders, component/library manufacturers, pre-CASE systems, what can we do? There is no mechanism to facilitate importing such code.

We can say the same with respect to changing code from designs changed from within the CASE tool as mentioned previously. How can we inform a version/configuration management tool of what has been changed in a meaningful way? How can we reflect the model inside the code, in such a way that the programmer, when debugging/testing the program, may know how it affects the model?

The only way to solve all these problems is with a language (a “round-trip engineering language”) that can be used to enrich the source code in any language. By this we mean, that it is able to describe all OOAD constructs to facilitate round-trip engineering and that is sufficiently simple that any programmer can learn in a few days.

We believe UOL is a solution to the above problems or considerations.

2.5.2 What is a Round-Trip Engineering Language?

A Round-Trip Engineering Language is a Textual OO Full Life-Cycle Language that is capable of being embedded any OO programming language program.

OOAD modeling languages are richer than OO programming languages. They are capable of describing more concepts and in more detail than any OO programming language. In fact, there is not even uniformity of concepts among OO programming languages. Some as Delphi[3] have modules expressed syntactically, others as C++ can express them indirectly or maybe not even that, as in the case of Visual Basic[4].

Transforming an OOAD model to code implies, therefore, a loss of information in all cases. The only way to avoid this is embedding in the source code exported/generated from the CASE tool additional information that, enriching the code, describes what was originally documented.

Can we consider a round-trip engineering language a new category or a new idea? Yes and no. Enriching source code with comments has been done traditionally by most CASE tool builders but in an informal and proprietary way, many times specifically defined for each language. What UOL brings to the OO community is a formal language that can be embedded in any OO programming

[3] Delphi is a registered trademark of Borland
[4] VisualBasic is a registered trademark of Microsoft.

language (much in the same way that SQL is embedded) and that is capable of describing all the elements that the modeler has documented.

A round-trip engineering language must be especially simple and easy to learn by any programmer. It is expected that he can learn it in a few days and that when he maintains a program, he will simultaneously maintain both the source code as well as the UOL code and with the minimum burden possible.

There are two alternatives when generating UOL embedded code. The first is to generate a complete description of the program with UOL and the second is to generate only those constructs not supported by the language. In the appendix, we have shown both alternatives. We believe that the first alternative makes somewhat more readable the generated code. In any case, both can be make available to the programmer by the CASE tool. For a CASE tool builder, it only affects the control flow of the reverse engineering tool. In the first case, the UOL parser can always start the collaborative compilation and it will remain in control of the process at all times. In the second case, control of the compilation process will depend on the target language. This will be further explained in the chapter 6.

Although some UOL constructs (i.e. use cases, sequence diagrams, deployment, etc.) may seem unnecessary from a strict reverse engineering source code task, they are, however, necessary from the point of view, that they describe the full system and that any part of it may be changed by the programmer. This will, at the same time, enhance her/his role as a *system* developer. Using the full expressive power of UOL reinforces the concept that when we change code we change design, that the distinction among tasks (analysis/design/programming) in OO is blurred and that we are all responsible of the system as a whole.

There are also some secondary benefits in using UOL. For example, it easily allows the reverse engineering tool to analyze syntactically and semantically the reengineered model *before* importing and restructuring the repository model. In this way, the tool can detect not only syntactic errors but also integrity or consistency errors (ex. services used in sequence diagrams and not developed in the objects because of changes, etc.). It also facilitates version control of the model because the matching process between the repository and the imported versions (detecting changes, additions and deletions) is simplified not having to do “automatic deduction” from the source code by the importing tool.

2.6 Tool to Tool Communication

The second type of communication is the one that must exist for a direct dialog between tools. For this type of communication the OMG has made the SMIF RFP. Let us now consider what this type of communication implies.

Communication between tools may have to be done in real-time or in batch.

Real-time communication requires standardizing the language. MOF is already a form of real-time communication language proposed by OMG. However, there are several situations and/or aspects that must be considered in this case. If all tools supported MOF the SMIF RFP would not be necessary.

For many reasons not everybody will accept MOF and CORBA, which is inherent to MOF. A very special case of this is Microsoft that has assumed UML but not MOF for its repository.

If the language used by a tool is not OMG's standard and the tool builder wants to either connect their tool to another's or to allow another company to connect to their tool, it will be necessary to agree between tool constructors what will be the valid language and it will only be valid for the two

partners (and others that may also accept it). Any other tool constructor that does not accept and/or have access to this private/proprietary language is blocked from a market, which may be very important.

MOF is based on CORBA and has its many benefits but it also has, however, an important drawback: it requires working with ORBs even though in many cases it may be unnecessary and, therefore, expensive and resource consuming.

Finally, a general opinion on CASE tools, which comes from structured method CASE tools, is that the best tools are those that support the whole life-cycle: the Integrated CASE. That is to say, tools that allow doing everything from within the tool. Integrated tools have usually been developed by large companies, which are the only ones with sufficient resources to develop all the required modules to be considered a full life-cycle tool.

However, although it is positive to be able to carry out every task with only one tool, it is not necessarily true that the best tool is the one developed entirely by one constructor. In many cases these tools have some of the modules excellent but others are inadequate in some cases or for some users. On the other hand, the existence of these "complete" tools does not precisely motivate smaller companies that can offer interesting or innovative ideas in some aspects but are unable to develop a full CASE tool.

A CASE tool can have many modules or components integrated: repository, graphical designer, screen designer, code generator, metrics and standards validator, etc. The central component is, naturally, the repository but conceptually there is no reason for which any pair of tools should not be able to communicate if they are based on the same model: UML. We should be able to represent graphically the design from the source code or obtain metrics or generate code from a graphical design without having to go through the repository.

From the UML 1.1 (Summary Document, page 8) we read:

> "Standardizing a language is necessarily the foundation for tools and process. The Object Management Group’s RFP (OADTF RFP-1) was a key driver in motivating the UML definition. The primary goal of the RFP was to enable tool interoperability. However, tools and their interoperability are very dependent on a solid semantic and notation definition, such as the UML provides. The UML defines a *semantic* meta-model, not a tool *interface*, *storage*, or *run-time* model, although these should be fairly close to one another. ".

In the same way that UML-MOF standard will incentive the CASE tool industry, if a real-time tool communication standard language existed, it would be possible to create a CASE component industry allowing users to build tailored CASE tools adapted to their precise needs.

A language, such as UOL, would permit communication between any two tools used in the construction of OO systems. That is, it would be a Tool Interface Language.

Batch communication requires also standardizing the language. The OMG's SMIF RFP hopes to define a standard for batch communication. Naturally, this RFP mentions CDIF to protect tool builder's previous investment in this technology. Even though this argument is out of the question and must be completely accepted, there are other aspects that must be considered and that make an extension of CDIF inadequate as the SMIF solution.

Even if an extension to CDIF is developed CDIF it is pre-OO and not OO. CDIF was developed to port full models between tools and It was based on pre-OO concepts that were not standardized (data modeling, data-flow diagrams, etc.). CDIF has no semantics and given that the communication between tools is batch it easily allows for errors during the port.

In many situations two tools must communicate in batch form (CASE tools generating code and compiler) but it is necessary for the software engineer to understand the port. There is one very special case of this situation. We refer to round-trip engineering as we have previously mentioned: code with embedded AD constructs. In this respect we must consider two aspects.

First that being CDIF non-OO and developed exclusively to communicate between tools, it is complex and cryptic for the programmer if it were possible to embed in their code. It would also require working with two different paradigms.

And second, that in some cases the software engineer must work with two tools, in which one or the other (or both) does not support the full standard: UML does not support all OO concepts and programs, certainly, do not support full UML

A textual OO full life-cycle language, such as UOL is a solution to all these considerations.

2.7 Conclusions

Now that we have reviewed what a textual OO full life-cycle language is and its need let us summarize the main requirements that any proposal in response to the SMIF RFP should comply with:

- Stream-based Model Interchange Format

  From the SMIF RFP we extract that its specific objectives are:

  - Establish an industry standard specification for a stream-based model interchange format,
  - Provide a generic format that can be used to transfer a wide variety of models,
  - Demonstrate that it can be used to exchange OMG Object Analysis and Design Facility (OADF) compliant models (UML based) and models compliant to other MOF-compliant meta-models and extensions (e.g. Workflow Management Facility and Business Object Facility meta-models), and
  - Leverage existing vendor-neutral transfer formats as much as possible.

- Generic Communication and Interchange

  From the previous discussion we obtain the following requirements:

  - It must be a textual, human readable, format,
  - It must be specially simple, easy to learn and use,
  - It must provide maximum support for all UML constructs to allow for seamlessness,
  - It must be adequate for tool to tool communication both in real-time and batch,
  - It must be independent of CORBA although compatible with it and
  - It must be adequate to support Round-Trip Engineering.

3 The Universal Object Language Specification

3.1 UOL Syntax

To describe this part, we use an Extended BNF grammar for its reading simplicity. Please see appendix 7.1 for the BNF syntax of UOL

3.1.1 Lexical Specification

The lexical part of UOL consists of a large number of tokens because of the many definitions and concepts that in UML are described. They are:

action	**branch**	**deferred**	**final**	**instance**	**package**	**simple**	**true**
actions	**by**	**diagrams**	**flow**	**interface**	**partitioned**	**state**	**undefine**
activity	**call**	**else**	**fork**	**is**	**postcondition**	**static**	**unique**
actor	**class**	**end**	**from**	**join**	**precondition**	**stereotype**	**use**
adaptation	**collaboration**	**entry**	**frozen**	**like**	**prefix**	**stereotyped**	**usecase**
addonly	**component**	**event**	**history**	**link**	**raise**	**subactivity**	**values**
after	**composite**	**exception**	**implements**	**machine**	**redefine**	**submachine**	**viewed**
all	**concurrent**	**exit**	**implies**	**model**	**relation**	**subsystem**	**when**
alternative	**constrained**	**expanded**	**import**	**node**	**rename**	**synchronous**	**with**
and	**constraint**	**export**	**in**	**none**	**request**	**tag**	**xor**
any	**course**	**extend**	**infix**	**not**	**result**	**then**	
as	**creates**	**extension**	**inherit**	**of**	**select**	**to**	
attached	**current**	**false**	**initial**	**or**	**shallow**	**transition**	
BIT	**deep**	**feature**	**inout**	**out**	**signal**	**trigger**	

There are some tokens that must be included for the inclusion of OCL, they are:

Bag	**Collection**	**else**	**endif**	**enum**	**if**	**Set**	**Sequence**
then							

The regular expressions defined in UOL are:

OCLtypeName: [A-Z][a-zA-Z_0-9]+
OCLname: [a-z][a-zA-Z_0-9]+
Integer_constant: ([1-9][0-9]*|0)
Character_constant: "([^\t\n] | (\\[^\t\n]))"
Range: [1-9][0-9]* {mdot} ([1-9][0-9]*| *)
Float_constant: ([0-9]+.[0-9]*([eE][+|-]?[0-9]+)?
Comment: -- [^(--)\n]
String: ' ([^'\\] | \\\\[ntbrf\\\\"\"]|[0-7][0-7]?|[0-3][0-7][0-7] | [\n\r])* '
TextMultiline: text " [^"] "
CommentMultiline: **comment** " [^"] "

Anonymous ?[0-9]*

3.1.2 Syntax Specification

3.1.2.1 Start production

In UOL the transmission unit, and thus the codification unit, are the model and the package. The model construction allows the interchange of models, and the package construction allows the auxiliary industry to give the design of frameworks or class libraries.

```
Start_production                    -> ( Model_declaration | Package_declaration )
```

3.1.2.1.1 Examples

```
-- Can be a model
model anExample
       -- body ommited
end -- model anExample

-- Can be a package
package UML_UOL
       -- body ommited
end -- package UML_UOL
```

3.1.2.2 Model declaration

In UOL we leave most of the constructs as optional. In this way we can obtain correct UOL from incomplete models.

```
Model_declaration                   -> model Model_name
                                         ( Package_or_subsystem_declaration )*
                                         ( View_element_decl_list )? end
Model_name                          -> identifier
```

In this production we declare the diagrams that compose the model.

```
View_element_decl_list              -> diagrams View_element_declaration ( ';'
                                         View_element_declaration )* end
View_element_declaration            -> Identifier_list ':' View_element_kind
View_element_name                   -> identifier
View_element_kind                   -> identifier Extension_use (Invariant)?
```

The view_element_kind is the name of a kind of diagram (i.e. static diagram or use case). We leave the name as an identifier instead of providing a closed list of diagram names to allow the use of a large list of diagrams from different methodologies.

```
Package_or_subsystem_declaration    -> ( Subsystem_declaration
                                         | Package_element_decl | Use_of_tagged_value
                                         | Use_of_constraint | Use_of_stereotype )
```

A model is also the top most package and thus can declare all the elements that can be found in a package. In addition it can declare also subsystems.

3.1.2.2.1 Example

```
model anExample
       -- Element declarations (package, subsystem, ...)
       -- Declaration of the diagrams used in the model
       diagrams
            MainD,SecondD:StaticDiagram
            -- Declaration of the stereotypes, tag values and
       -- constraints applied to the MainD and SecondD
       -- If there are more diagrams, we must put a ';'
       -- else the end token.
       end -- diagrams
end -- model anExample
```

3.1.2.3 Package

```
Package_declaration              -> package Package_name
                                      ( viewed with View_element_name_list )?
                                      Extension_use
                                      ( inherit Package_name_list )?
                                      ( import Package_import_list )?
                                      ( Package_element_decl_list )?
                                      ( Use_of_constraint )? end;
```

```
Package_name                     -> identifier
View_element_name_list           -> View_element_name Position ( ',' View_element_name
                                      Position )*;
Position                         -> ( '(' Dec ',' Dec Third_dimension ')' )?;
Third_dimension                  -> ( ',' Dec )?;
Package_name_list                -> ( Use_of_constraint )? Package_name ('('Name')')?
                                      ( ',' Package_name ( '(' Name ')' )? )*;
Package_import_list              -> Package_import_elem ( ',' Package_import_elem )*;
Package_import_elem              -> ( '{' Visibility '}' )? ( Element_name
                                      ( '::' Element_name )* As_alias )?
                                      from Package_name;
As_alias                         -> ( as Alias )?;
Element_path                     -> Element_name ( '::' Element_name )* ;
Alias                            -> Element_name;
```

The package construction can declare all the other constructions except model and subsystem. Each element can give the list of diagrams in which it appears. Also, most of the elements can inherit from compatible elements. A package can optionally import elements from other packages. This import can specify a list of elements to import or can import the whole package. The import of an element is conditioned by the element's visibility. An imported element may receive an alias and change its visibility by a more restrictive one. These changes only affect the element as a member of the importing package.

Of course the main use of a package is to group elements together. Following is the declaration of such elements.

```
Package_element_decl_list        -> is ( ('{' Visibility '}')? Package_element_decl)+;
Package_element_decl             -> Package_declaration | Interface_declaration
                                      | Class_declaration | Relation_declaration
                                      | Extension_declaration | Usecase_abstraction
                                      | Activity_model | Comment_definition
                                      | Object_declaration
                                      | ( actor | exception ) Light_body
                                      | ( component | node ) Ultra_light_body
```

```
                                        | Collaboration_declaration
```

This construction reflects a comment in the model. It is usually attached to an element. Standalone comments are also allowed and then they need to declare in which diagram(s) they are shown.

```
Comment_definition              -> CommentMultiline (attached to Element_name
                                      | viewed with View_element_name_list)
Light_body                      -> Name ( Formal_generics )? Extension_use
                                      ( viewed with View_element_name_list )?
                                      ( Inheritance)? Features ( Use_of_constraint )?
                                      end
Ultra_light_body                -> Name ( Formal_generics )? Extension_use
                                      ( viewed with View_element_name_list )?
                                      ( Inheritance )? Name_list (
Use_of_constraint)?
                                      end
```

3.1.2.3.1 Example

```
package PMain viewed with MainD -- Position for the viewed
       inherit constrained with { aConstraint } UOL_UML
       import from UML_UOL
       -- is keyword must be put if there is at least one element.
       -- declaration of the elements of a package
end -- pachage PMain
```

3.1.2.4 Subsystem

```
Subsystem_declaration           -> Subsystem_header ( Formal_generics )?
                                      Extension_use
                                      (viewed with View_element_name_list )?
                                      (Inheritance )?
                                      (import Package_import_list )?
                                      (feature '{'Visibility }' Operation_list end )*
                                      (Package_element_decl_list )?
                                      (Use_of_constraint )?
                                      end
```

A subsystem is 'a package with behavior'. It declares elements, like a package, but also declares operations and, optionally can be instantiated (if is not marked as 'deferred').

```
Subsystem_header                -> ( deferred )? subsystem Subsystem_name
Subsystem_name                  -> identifier
```

3.1.2.4.1 Example

```
subsystem aSubsystem -- can not be deferred if it is final.
       -- subsystem body (Use of extensions, inheritance,...)
end -- subsystem aSubsystem
```

3.1.2.5 Features

Features share the semantics of UML and most of the syntax with Eiffel. It is a block beginning with the keyword '**feature**' then a visibility and list of operations, methods and attributes. Not all the elements that can have features can declare all the kinds of features.

```
Features                        -> ( feature '{' Visibility '}'  Feature_rest end )*
```

The a là Eiffel visibility declares which element can access a marked element. However the mapping to the UML's visibility's is straightforward.

```
Visibility                      -> any | none | Classifier_list
Classifier_list                 -> Classifier_name ( ',' Classifier_name )*

Feature_rest                    -> Use_of_stereotype
                                     | Use_of_stereotype ';' Feature_list
                                     | Feature_list
Feature_list                    -> Feature_declaration ( ';' Feature_declaration )*

Feature_declaration             -> Use_of_tagged_value
                                     | Operation_declaration
                                     | Method_declaration
                                     | Attribute_declaration
```

The rest are needed to allow describing all the ways in which a feature can be declared.

```
Operation_rest                  -> is Specification
Method_rest                     -> Specification is Routine
                                     | is Routine | like identifier is Routine
Attribute_rest                  -> ',' Identifier_list Type_mark
                                     |(Use_of_constraint)? Type_mark is
Initial_value
                                     |(Use_of_constraint)? Type_mark
                                     |(Use_of_constraint)? ':' unique '{'
                                     Identifier_list '}'
Signature_rest                  -> static identifier | identifier
```

3.1.2.5.1 Attributes

The attributes can have cardinality, invariants, type, initial values, stereotypes and tagged values.

```
Attribute_declaration           -> Attribute_signature Attribute_single_or_multi
                                     Extension_use
Attribute_single_or_multi       -> Attribute_rest | Cardinality2
                                     ( Use_of_constraint )? Type_mark
                                     ( is '{' Expression_list '}' )?
Attribute_signature             -> Signature_rest
                                     | frozen Signature_rest
                                     | addonly Signature_rest
Initial_value                   -> Expression;
```

The type mark includes a delegation mechanism. The type can be the same that the one of the element that appears after the 'like' keyword.

```
Type_mark                       -> ':' identifier | ':' like identifier
```

3.1.2.5.2 Operations

Operations are only specifications and thus never instantiable. They can have a full signature, pre and post conditions, and a specification.

```
Operation_declaration           -> Signature Operation_body;
Operation_body                  -> OM_body Operation_rest;
Entity_declaration_list         -> Entity_declaration_group
                                     ( ';' Entity_declaration_group )*
Entity_declaration_group        -> ( Parameter_kind )? Parameter_name_list Type_mark
                                     ( is Initial_value )?;
Parameter_name_list             -> identifier (Cardinality)?;
Parameter_name_list             -> Parameter_name_list ',' identifier (Cardinality)?;
```

```
Identifier_list                 -> identifier ( ',' identifier )*;
Parameter_kind                  -> in | out | inout;
Specification                   -> TextMultiline;
```

3.1.2.5.3 Methods

A method can have the same components that an operation can have plus an implementation.

```
Method_declaration              -> Method_header Method_body;
Method_header                   -> Signature | Signature_rest;
Method_body                     -> OM_body Method_rest;
Routine                         -> TextMultiline;
```

The following is part of the body of the method and operations.

```
OM_body                         -> '('( Entity_declaration_list )? ')' ( Type_mark )?
                                     ( '{' PrePost )?
                                     Extension_use ( Use_of_constraint )?;
Signature                       -> deferred Signature_rest
PrePost                         -> precondition ':' Constraint_expression Post_opt
                                     | postcondition ':' Constraint_expression '}'
Post_opt                        -> '}' MorePost
MorePost                        -> ('{' postcondition':' Constraint_expression '}' )?
```

3.1.2.5.4 Features with only Attributes and Operations

These features, are declared separatly from the others, to have a more readable grammar. These features are used in the elements that can not declare methods (usecases,...)

```
Features_attrib_or_Oper         -> ( feature '{' Visibility '}'
                                     Feature_rest_attrib_or_Oper end )*
Feature_rest_attrib_or_Oper     -> Use_of_stereotype
                                     | Use_of_stereotype ';'
                                     Feature_list_attrib_or_Oper
                                     | Feature_list_attrib_or_Oper
Feature_list_attrib_or_Oper     -> Feature_declaration_attrib_or_Oper
                                     ( ';' Feature_declaration_attrib_or_Oper )*
Feature_declaration_attrib_or_Oper -> Use_of_tagged_value
                                     | Operation_declaration | Attribute_declaration
```

3.1.2.5.5 Example

```
feature {any}
       isMarried, isUnemployed:Boolean;
       birthDate:Date;
       age:Integer;
       firstName,lastName:String;
       sex: unique { male,female };
       deferred income(d:Date):Integer is text""
end -- feature
```

Note that the semicolon '**;**' is used as a concatenator (as in Eiffel) and not as a final sentence (as in C or Java), therefore, the following declaration is correct:

```
feature
       anAttrib1:aType1;
       anAttrib2:aType2
end
```

But these others are incorrect:

```
   feature
       anAttrib1:aType1
       anAttrib2:aType2
      -- ^ ; expected
   end
```

or

```
   feature
       anAttrib1:aType1;
   end
-- ^ an attribute, operation or method expected
```

3.1.2.6 Classes

A class may declare and use extensions, can be a template, can inherit, can have any kind of feature and it can also use invariants. It must be marked as deferred if any of its methods is deferred.

```
Class_declaration              -> Class_header ( Formal_generics )?
                                     ( viewed with View_element_name_list )?
                                     Extension_declaration_list
                                     Extension_use
                                     Class_body end
Class_body                     -> Inheritance Rest ( Use_of_constraint )?
                                     | Features State_machine ( Use_of_constraint )?
                                     | Features feature '{' Visibility '}'
                                     Feature_rest end ( Use_of_constraint )?
Rest                           -> Features ( State_machine )?
Class_header                   -> ( deferred )? class Class_name
Class_name                     -> identifier
```

3.1.2.6.1 Example

```
class Person viewed with MainD
       feature {any}
            isMarried, isUnemployed:Boolean;
            birthDate:Date;
            age:Integer;
            firstName,lastName:String;
            sex: unique { male,female };
            deferred income(d:Date):Integer is text"Incoming operation"
       end
       -- State machine for the class Person
       -- declaration of an invariant
       constrained by
       { self.age>=0 }
       -- rest of constraints ommited
end -- Class Person
```

3.1.2.7 Instances

The instances will follow the definition of its base class, giving values to its attributes, using extensions such as stereotypes and tag values and/or defining invariants.

```
Object_declaration             -> Object_name [Formal_generics] instance of
                                     Element_path Extension_use [Viewed_with]
                                     [Object_body] [Invariant] end;
```

```
Object_name                    -> identifier
Object_body                    -> is Attribute_value (';' Attribute_value)*
Attribute_value                -> identifier is Expression
```

3.1.2.7.1 Example

```
CloseObject instance of Usecase
       is
       annotation is '(a) The system will load the current object that
            is referenced (b) ask to the actor for its username update
            the username in the document (c), finally save the document
            (d)';
       name is CloseObject;
       extension_point is <<'a','b','c','d'>>
end
```

3.1.2.8 Interfaces

An interface is very similar to a class but can not have methods or attributes.

```
Interface_declaration          -> Interface_header ( Formal_generics )?
                                       ( viewed with View_element_name_list )?
                                       Extension_declaration_list
                                       Extension_use ( Inheritance )?
                                       (feature '{' Visibility '}' Operation_list
end)*
                                       ( Use_of_constraint )? end
Interface_header               -> interface identifier;
Operation_list                 -> Operation_declaration (';'Operation_declaration )*
```

3.1.2.8.1 Example

```
interface anInterface[Param1 constrained by {self.Param1>=0}, Param2]
       viewed with aDiagram
       feature {any}
            -- only operations
            deferred static anOperation(aParam:aType):aReturnType
                 {precondition: aConstraint}
                 {postcondition: aConstraint}
                 constrained by {aConstraint}
                 is text "Specification"
       end
       -- constrained by...
end -- interface anInterface
```

3.1.2.9 Declaration and use of extensions

Some extensions (tag values or stereotype) must be declared before their use. Constraints may be declared also but are not mandatory.

```
Extension_declaration_list     -> ( Extension_declaration )*
Extension_declaration          -> Constraint | Tagged_values | Stereotype
```

The following productions allow using stereotypes and tagged values. Note that only one stereotype use is permited.

```
Extension_use                  -> ( Use_of_stereotype )? ( Use_of_tagged_value )*
```

3.1.2.9.1 Declaration of constraints

The declaration of a constraint is the mechanism to reuse constraints. A name is tied to a constraint expression and this name can be used after in any constraint.

```
Constraint                      -> constraint Constraint_def_list end
Constraint_def_list             -> ( identifier is '{' Constraint_expression '}')+
```

A constraint can be either an OCL_expression or a TextMultiline. A TextMultiline is just a free text description. The OCL_expression is a valid OCL expression defined in the OMG document number ad970808. The OCL grammar has been merged with the UOL grammar, exactly as it is described in the document mentioned previously. This improves the use of constraints (preconditions, postconditions, invariants,...).

```
Constraint_expression           -> OCLexpression | TextMultiline
```

3.1.2.9.2 Use of constraints

```
Use_of_constraint               -> constrained by '{' Constraint_expression '}'
```

3.1.2.9.3 Tagged values declaration

The declaration of a tag value allows assigning a default value.

```
Tagged_values                   -> tag values Tagged_values_def_list end
Tagged_values_def_list          -> ( Tagged_values_def )+
Tagged_values_def               -> identifier ( is Initial_value )?
```

3.1.2.9.4 Use of tagged values

The use of tag values is a list of properties of the form <tag,value>. A tag value with default value can appear simply as <tag> if the default value is suitable.

```
Use_of_tagged_value             -> with tag values '(' Property_list ')'
Property_list                   -> Property ( ',' Property )*
Property                        -> '<' identifier ( ',' Expression )? '>'
```

3.1.2.9.5 Stereotypes declaration

The stereotype declaration gives a name to the stereotype, declares its base class (the class that can be stereotyped with this stereotype) and declares the tagged values that act as attributes for the stereotype. A stereotype can also inherit from other stereotypes with compatible base class and declares constraints.

```
Stereotype                      -> stereotype identifier of Base_class ( Icon )?
                                      ( inherit Stereotype_parent_list )?
                                      Stereotype_extension_dec end
Stereotype_extension_dec        -> ( Constraint | Tagged_values )*
Base_class                      -> identifier;
Icon                            -> viewed as String;
Stereotype_parent_list          -> identifier ( '(' Name ')' )? ( ',' identifier,
                                      ( '(' Name ')' )? )*
```

3.1.2.9.6 Use of stereotypes

```
Use_of_stereotype               -> stereotyped with identifier
```

3.1.2.10 Identifiers

Identifier is divided in OCLtypeName or OCLname. This is to give the maximum support to the OCL grammar. Therefore, there will be a production where an identifier will be an OCLtypeName or an OCLname, where an OCLtypeName is an identifier that must begin with an uppercase letter and an OCLname is an identifier that must begin with a lowercase letter. Another branch that can be taken is expressing that the name of an element is no name (in UML exists a difference between an element with the name null and an element without name). For this reason, we include a 'Anonymous' token expressed as a question mark '**?**'.

```
identifier                      -> OCLtypeOrName | Anonymous
```

3.1.2.10.1 Example

```
aValidName
AValidTypeName
anIdentifier_1
AnIdentifier_2
?
InvalidOne? -- The question mark not included as a letter
7NotCorrect -- It must begin with a letter.
_NotCorrect -- It must begin with a letter.
```

3.1.2.11 State machine

State machines are defined as in UML. They must contain a composite state in which there all the states of the state machines are declared. A composite state that is the *top state* must not end with the keyword end, because it uses the same end that the state machine. This is defined in this way for readability purposes.

```
State_machine                   -> state machine Name
                                      ( viewed with View_element_name_list )?
                                      ( Constraint_use_def )?
                                      Machine_body
                                      end
Name                            -> identifier
Path_name                       -> Name (Path Path_name)*
Constraint_use_def              -> Constraint ( Use_of_constraint )?
                                      | Use_of_constraint
```

In the next production we define the *top most state*. It can be seen that there is no end.

```
Machine_body                    -> Composite_state Transition_list Action_def_list;
State_definition                -> state Name ( viewed with View_element_name_list )?
                                      ( Constraint_use_def )?
                                      ( Action )?
                                      Internal_transition_list
                                      ( deferred ( event Name )+ )?
Action                          -> actions (entry Action_list)? (exit Action_list)?
```

States defined as substates (i.e. not the *top most state*) must be all of the same kind, concurrent or not, but they can not be mixed.

```
Composite_state                 -> composite State_definition Concurrent_state_list
                                      | composite State_definition State_list
Concurrent_state_list           -> ( concurrent State_list )+
State_list                      -> ( State_kind )+
```

In the next production there is a branch that can match with Machine_body, this is the composite state. UML defines that the state machine contains all the transitions and the action definitions, but

in UOL those are defined in the concurrent state in which they are included. If it is not defined in this way, it implies putting path references for all the states, and this is completely unreadable.

```
State_kind                      -> ( Simple_state | Pseudostate | Submachine
                                     | Machine_body ) end
Simple_state                    -> simple State_definition
Pseudostate                     -> Pseudostate_kind Name ( Constraint_use_def )?
                                     ( Action )?
Pseudostate_kind                -> ( deep | shallow ) history
                                     | initial
                                     | final
                                     | join
                                     | fork
                                     | branch
Submachine                      -> submachine Name
                                     ( viewed with View_element_name_list )?
                                     ( Constraint_use_def )? Machine_body
Internal_transition_list        -> ( transition When_or_after ( Trigger_expression )?
                                     ( Action_sequence )? )*
When_or_after                   -> when Guard_expression
                                     | after Time_expression ;
Time_expression                 -> Integer_constant String;
Guard_expression                -> Expression | TextMultiline;
Trigger_expression              -> call Operation_use
                                     | trigger Signal_or_time_or_change
Signal_or_time_or_change        -> Signal_definition
                                     | after Time_expression
                                     | when Boolean_expression;
Boolean_expression              -> TextMultiline;
Transition_definition           -> transition Name from ( Path_name | initial ) to
                                     ( Path_name | final )
                                     Guard_expression_opt ( Trigger_expression )?
                                     ( Action_sequence )?
Transition_list                 -> ( Transition_definition )*
Guard_expression_opt            -> When_or_after
Action_sequence                 -> actions Action_list
Action_list                     -> identifier ( ',' identifier )*
Action_def_list                 -> ( Action_definition )*
Action_definition               -> ( synchronous )? action Name ( Recurrence )?
                                     ( Script )?
                                     Object_set_expression_opt
                                     ( request Operation_or_signal )?
                                     Action_kind
Script                          -> String
Recurrence                      -> '(' Expression ')'
Object_set_expression_opt       -> ( to Object_set_expression )?
Object_set_expression           -> Name ( ',' Name )*
Operation_or_signal             -> Operation_use | Signal_definition
Signal_definition               -> signal Name ( Reception )? ( Exception )?
Reception                       -> to Name_comma_list
Name_comma_list                 -> Name ( ',' Name )*
Exception                       -> Exception_list
Exception_list                  -> ( raise Exception_use )+
Exception_use                   -> Name from Name_comma_list
Action_kind                     -> ( call Operation_use
                                     | creates identifier
                                     | TextMultiline )?
Operation_use                   -> Name '.' Name '(' Expression_list ')'
```

3.1.2.11.1 Example

```
state machine SPerson viewed with PersonD
       composite state SCPerson
            -- States definitions
            concurrent composite state CivilStatus
                 simple state Single viewed with PersonD end
                 simple state Married viewed with PersonD end
                 transition ? from initial to Single when not isMarried
```

```
                transition ? from initial to Married when isMarried
                transition ? from Single to Married when isMarried
                transition ? from Married to Single when not isMarried
           end
           concurrent composite state JobStatus
                simple state Unemployed viewed with PersonD end
                simple state Employer viewed with PersonD end
                transition ? from initial to Unemployed
                     when isUnemployed
                transition ? from initial to Employer
                     when not isUnemployed
                transition ? from Employer to Unemployed
                     when isUnemployed
                transition ? from Unemployer to Employer
                     when not isUnemployed
           end
       -- Transitions definitions
       transition ? from initial to CivilStatus
       transition ? from initial to Job
end -- State machine SPerson
```

3.1.2.12 Activity model

Activity models are exactly as the state machines, but they add new features such as the partition in which a state belongs and the object flow state.

```
Activity_model                  -> activity Name
                                     ( viewed with View_element_name_list )?
                                     ( Constraint_use_def )?
                                     Activity_body
                                     end
Activity_body                   -> Activity_state Transition_list Action_def_list
Activity_state                  -> composite State_definition partitioned in Name
                                     Act_concurrent_state_list
                                     | composite State_definition partitioned in
Name
                                     Act_state_list
Act_concurrent_state_list       -> ( concurrent Act_state_list )+
Act_state_list                  -> ( Act_state_kind partitioned in Name end )+
Act_state_kind                  -> Activity_body
                                     | Act_simple_state
                                     | Pseudostate
                                     | Subactivity
Act_simple_state                -> Action_state | Object_flow_state
Action_state                    -> state Name ( viewed with View_element_name_list )?
                                     ( Constraint_use_def )?
                                     ( Action )?
Object_flow_state               -> State_definition flow Name '[' Name ']'
                                     ( Use_of_constraint )?
Subactivity                     -> subactivity Name
                                     ( viewed with View_element_name_list )?
                                     ( Constraint_use_def )? Activity_body
```

3.1.2.12.1 Example

```
activity anActivity
       viewed with anActivityD
       -- constraint def
       -- constraint use
       composite state aSComposite
            -- States definitions
```

```
            state anState1
                 viewed with aDiagram
                 partitioned in aPartition1
            end
            state anState2
                 viewed with PersonD
                 partitioned in aPartition1
            end
            transition init from initial to anState1 after 3 'sec'
            transition ? from anState1 to anState2 when anExpression
            transition ending from anState2 to final
                 when not anExpression
       -- There is no end for this composite state, because it is the
       -- end of the state machine
end -- activity anActivity
```

3.1.2.13 Usecases

There are three kinds of usecases: the declaration, the extension and the instance of a usecase.

```
Usecase_abstraction              -> Usecase_definition
                                      | Usecase_instance
                                      | Usecase_extension
```

This is the declaration of a usecase, defined in UML as UseCase

```
Usecase_definition               -> usecase Name ( Formal_generics )?
                                      ( inherit Name_inherit_list )?
                                      ( use Name_list )?
                                      ( actor Name_list )? Features_attrib_or_Oper
                                      ( TextMultiline )?
                                      ( alternative course TextMultiline )?
                                      (extension in Extension_point_list)? end
```

The instance of a usecase has a type mark that defines the usecase that is instantiated. The instantiation consists of a list of arguments instantiating the formal generics and a list of attributes with their values, all defined in the usecase definition.

```
Usecase_instance                 -> usecase Name '(' ( Entity_declaration_list )? ')'
                                      ( Type_mark )? is Usecase_method_list end
```

The extension of a usecase is defined separately from the usecase, because there can be many extensions of the same point of a given usecase.

```
Usecase_extension                -> extend Usecase_path with Usecase_path_list
                                      in Extension_point
Usecase_method_list              -> ( Usecase_method )+
Usecase_method                   -> identifier is Expression
Extension_point                  -> String
Extension_point_list             -> String_list
Name_inherit_list                -> Name ('(' Name ')')? (',' Name ('(' Name ')')? )*
Name_list                        -> Name ( ',' Name )*
Usecase_path                     -> Name ( Path Name )*
Usecase_path_list                -> Usecase_path ( ',' Usecase_path )*
String_list                      -> String ( ',' String )*
```

3.1.2.13.1 Examples

```
-- Usecase definition
usecase aUsecase [aUsecase1,aUsecase2]
       inherit aUsecase1(aDiscriminator),aUsecase2
       use aUsecase3,aUsecase4
       actor anActor1,anActor2
       feature {any}
            -- Only attributes or operations
            anAttribute:Integer;
            deferred anOperation() is text "Operation"
       end
       text "(1) This is the description of the usecase
             (2) using text multiline"
       alternative course
       text "This decribes the exceptions in the usecase"
             extension in '(1)','(2)'
end -- usecase aUsecase

-- Usecase instance
usecase anInstance(aParam1,aParam2:aType1;aParam2:aType2):aUsecaseDef
is
       aName1 is anExpression
       aName2 is anExpression
end -- usecase anInstance

-- Usecase extensions
extend aPackage::aUsecase with aUsecase7 in '(1)'
extend aPackage::aUsecase with aUsecase8 in '(2)'
```

3.1.2.14 Collaborations

A collaboration is a set of elements (classes and relations) that provides the implementations of a classifier or operation. Therefor, it describes required classifiers (with features) and interaction between them to achieve the desired goal.

```
Collaboration_declaration          -> collaboration Collaboration_name
                                        ( Formal_generics )?
                                        ( viewed with View_element_name_list )?
                                        ( implements Classifier_or_operation )?
                                       Class_or_intf_or_rel_decl_list
                                       Action_def_list ( Message_list )? end
Collaboration_name                 -> identifier
Classifier_or_operation            -> identifier
Class_or_intf_or_rel_decl_list     -> ( Class_declaration
                                        | Element_name
                                        | Interface_declaration
                                        | Relation_declaration )*
Message_list                       -> ( Message )+
Message                            -> actions Action_list to Classifier_name from
                                        Classifier_name
Classifier_name                    -> Element_name
Formal_generics                    -> '[' Formal_generic_list ']'
Formal_generic_list                -> Formal_generic ( ',' Formal_generic )*
Formal_generic                     -> Element_name ( Use_of_constraint )?
Element_name                       -> identifier
```

3.1.2.14.1 Example

```
collaboration aCollaboration
       -- Formal generics
```

```
        -- Viewed with
        implements aClassOrOp
             class aClass
                  -- body ommited
             end
             relation aRelation
                  -- body ommited
             end
             interface anInterface
                  -- body ommited
             end
        -- action def list
        actions anAction1, anAction2 to aClassName1 from aClassName2
        actions anAction3, anAction4 to aClassName3 from aClassName4
end -- collaboration aCollaboration
```

3.1.2.15 Expressions

```
Expression                      -> Call
                                     | Operator_expression
                                     | Equality
                                     | Manifest_constant
                                     | Manifest_array
Call                            -> ( Parenthesized_qualifier )? Call_chain
Call_chain                      -> Unqualified_call ( '.' Unqualified_call )*
Parenthesized_qualifier         -> Parenthesized '.'
Parenthesized                   -> '(' Expression ')'
Unqualified_call                -> Entity ( Actuals )?
Entity                          -> identifier
                                     | result
                                     | current
Actuals                         -> '(' Actual_list ')'
Actual_list                     -> Actual ( ',' Actual )*
Actual                          -> Expression
Operator_expression             -> Parenthesized
                                     | Unary_expression
                                     | Binary_expression;
Unary_expression                -> Unary Expression
Unary                           -> not
                                     | '+'
                                     | '-'
Binary_expression               -> Expression Binary Expression
Binary                          -> '+' | '-' | '*' | '/' | '<' | '>' | '<=' | '>='
                                     | '\\' | '//' | '^' | and | or | xor | implies
Manifest_constant               -> Boolean_constant | Character_constant
                                     | Integer_constant
                                     | Float_constant | String
Boolean_constant                -> true | false
Float_constant                  -> 'FLOATINGconstant0'
Manifest_array                  -> ANGLEBL Expression_list ANGLERR
Expression_list                 -> Expression ( ',' Expression )*
Comparision                     -> '=' | '/='
Equality                        -> Expression Comparision Expression
```

3.1.2.16 Inheritance

```
Inheritance                     -> inherit Parent_list
Parent                          -> (Use_of_constraint)? Class_type ( '(' Name ')' )?
Parent_list                     -> Parent ( ';' Parent )*
Class_type                      -> Class_name ( Actual_generics )?
Actual_generics                 -> '[' Type_list ']'
Type_list                       -> Type ( ',' Type )*
Type                            -> Class_type | Class_type_expanded | Anchored
                                     | Bit_type
Class_type_expanded             -> expanded Class_type
Anchored                        -> like Anchor
```

```
Anchor                          -> identifier | current
Bit_type                        -> BIT Constant
Constant                        -> Manifest_constant | Entity
Rename                          -> rename Rename_list
Rename_list                     -> Rename_pair ( ',' Rename_pair )*
Rename_pair                     -> Feature_name as Feature_name
Feature_name                    -> identifier | Prefix | Infix
Infix                           -> infix '(' Infix_operator ')'
Infix_operator                  -> Binary | identifier
Prefix                          -> prefix '(' Prefix_operator ')'
Prefix_operator                 -> Unary identifier
New_export_item                 -> Clients Feature_set
New_export_list                 -> New_export_item ( ';' New_export_item )*
New_exports                     -> export New_export_list
Class_list                      -> Class_name ( ',' Class_name )*
Clients                         -> '{' Class_list '}'
Feature_set                     -> Name_list | all
Undefine                        -> undefine Feature_list
Select                          -> select Feature_list
```

3.1.2.17 Relations

We reify the association into a class-like construct. It can have all the items a class can have except methods and adding a link list that declares the names of the elements linked by the association.

```
Relation_declaration            -> relation Relation_name
                                      Extension_declaration_list
                                      Extension_use
                                      ( Relation_inheritance )?
                                      ( Link_list )?
                                      Features_attrib_or_Oper
                                      ( Use_of_constraint )?
                                      end
Relation_name                   -> identifier
Relation_inheritance            -> inherit Parent_relation_list
Parent_relation_list            -> Parent_relation ( ';' Parent_relation )*
Parent_relation                 -> (Use_of_constraint)? Relation_type
                                      ( Relation_feature_adaptation )?
Relation_type                   -> Relation_path
Relation_path                   -> Element_path
Relation_feature_adaptation     -> adaptation ( Rename )? ( New_exports )?
                                      ( Undefine )? Relation_redefine
                                      ( Select )? end
Relation_redefine               -> redefine Feature_or_redef
Feature_or_redef                -> Feature_list | Redefine_with_list
Redefine_with_list              -> Redefine_pair ( ',' Redefine_pair )*
Redefine_pair                   -> Feature_name with Feature_name
```

Here we have the main differences with classes. The link clause provides a list of elements joined by the relationship. The link list may have two forms: plain list (a, b, c, d) or list of pairs (a to b, c to d). The first corresponds to an association, the second to a dependency (the pairs have 'direction'). Each entity declared in the link list may have an association end. Grammatically, an association end is a feature clause with visibility to the class which it is attached.

```
Link_list                       -> link Type_or_dependency
Type_or_dependency              -> Type_link_two_list ( with Classifier_name )?
                                      | Dependency_list ( Dependency_description )?
Type_link_two_list              -> Type_link ( Cardinality2 )? ',' Type_link_list
Cardinality2                    -> '[' Cardinality ']'
Cardinality                     -> Range_list Range_last
Range_last                      -> Range | Int_or_star
Range_mid                       -> Range ',' | Integer_constant ','
Range_list                      -> ( Range_mid )*
Int_or_star                     -> Integer_constant | '*'
```

```
Type_link_list                  -> Type_link ( ',' Type_link )*
Type_link                       -> Classifier_name ( Cardinality2 )?
Dependency_list                 -> Dependency ( ',' Dependency)*
Dependency                      -> Element_path to Element_path
Dependency_description          -> ( is TextMultiline )?
```

3.1.2.18 OCL

Finally, we introduce the grammar for the OCL, taken from the OMG document ad970808. The most recent updates on the Unified Modeling Language are available via the worldwide web:

http://www.rational.com/uml.

A free OCL Parser and the most recent information on the Object Constraint Language are available via the worldwide web: http://www.software.ibm.com/ad/ocl.

All the rules' names are changed adding the word OCL at the beginning; therefore it is easier to read and differentiates between the UOL grammar and the OCL grammar

```
OCLexpression                   -> OCLlogicalExpression
OCLifExpression                 -> if OCLexpression
                                      then OCLexpression
                                      else OCLexpression
                                      endif
OCLlogicalExpression            -> OCLrelationalExpression
                                      (OCLlogicalOperator OCLrelationalExpression )*
OCLrelationalExpression         -> OCLadditiveExpression
                                      (OCLrelationalOperator OCLadditiveExpression )?
OCLadditiveExpression           -> OCLmultiplicativeExpression
                                      (OCLaddOperator OCLmultiplicativeExpression )*
OCLmultiplicativeExpression     -> OCLunaryExpression
                                      (OCLmultiplyOperator OCLunaryExpression )*
OCLunaryExpression              -> (OCLunaryOperator OCLpostfixExpression )
                                      | OCLpostfixExpression
OCLpostfixExpression            -> OCLprimaryExpression (('.'|'->') OCLfeatureCall )*
OCLprimaryExpression            -> OCLliteralCollection
                                      | OCLliteral
                                      | OCLpathName OCLtimeExpression? OCLqualifier?
                                      OCLfeatureCallParameters?
                                      | '(' OCLexpression ')'
                                      | OCLifExpression
OCLfeatureCallParameters        -> '('(OCLdeclarator )? (OCLactualParameterList )?')'
OCLliteral                      -> OCLstring | OCLnumber | '#' OCLname
OCLenumerationType              -> enum '{' '#' OCLname ( ',' '#' OCLname )* '}'
OCLsimpleTypeSpecifier          -> OCLpathTypeName
                                      | OCLenumerationType
OCLliteralCollection            -> OCLcollectionKind '{'OCLexpressionListOrRange?'}'
OCLexpressionListOrRange        -> OCLexpression
                                      (( ',' OCLexpression )+ |('..' OCLexpression
))?
OCLfeatureCall                  -> OCLpathName OCLtimeExpression? OCLqualifiers?
                                      OCLfeatureCallParameters?
OCLqualifiers                   -> '[' OCLactualParameterList ']'
OCLdeclarator                   -> OCLname ( ',' OCLname )*
                                      ( ':' OCLsimpleTypeSpecifier )? '|'
OCLpathTypeName                 -> OCLtypeName ( '::' OCLtypeName )*
OCLpathName                     -> (OCLtypeName | OCLname )
                                      ( '::' (OCLtypeName | OCLname ) )*
OCLtimeExpression               -> '@' OCLname
OCLactualParameterList          -> OCLexpression ( ',' OCLexpression )*
OCLlogicalOperator              -> and | or | xor | implies
OCLcollectionKind               -> Set | Bag | Sequence | Collection
OCLrelationalOperator           -> '=' | '>' | '<' | '>=' | '<=' | '<>'
OCLaddOperator                  -> '+' | '-'
OCLmultiplyOperator             -> '*' | '/'
OCLunaryOperator                -> '-' | not
OCLnumber                       -> Integer_constant
```

```
OCLstring                          -> String
```

3.1.2.19 Example

This example is taken from the OMG document ad970808

```
( context: Company::hireEmployee(p : Person)
not employee->includes(p)and
employees->includes(p) and stockprice() = stockprice@pre() + 10
```

3.1.3 Encoding, tokenizing

The encoding is the one used in text-based files.
The tokenizing is 1 to 1 with the keywords.

3.1.4 UNICODE

Being that it is text-based format, the UNICODE can be used with no problems. As a proof of concept, in the parser that we developed, we read from ASCII format or UNICODE format. A table of properties (ftp://ftp.unicode.org/Public/UNIDATA/) for Unicode characters is provided on the Unicode ftp site, and complete information on the processes involved in proper Unicode rendering (such as the bidi algorithm or Indic reordering) can be found in The Unicode Standard, Version 2.0. (http://www.unicode.org/unicode/ uni2book/ u2.html). These algorithms are easy to implement, and we use it for the Unicode-based UOL files.

4 The XML-DTDs

4.1 The Mapping between UOL and XML

4.1.1 Justification

One could believe that XML would be a valid alternative for the SMIF RFP. Advantages of XML as a SMIF RFP proposal would be:

- XML is a standard defined by W3C and, therefore will be supported by Internet tools
- XML is textual and easily comprehensible and usable
- XML is object-oriented and could easily be mapped with MOF and UML
- XML carries meta-data and is extensible
- XML offers the possibility of performing structural validations using the Document Type Definition (DTD)

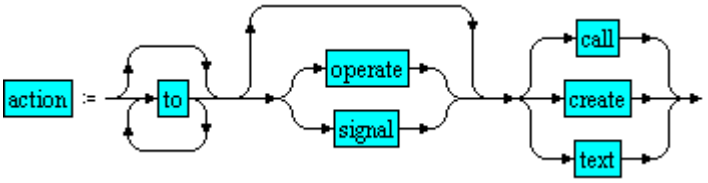
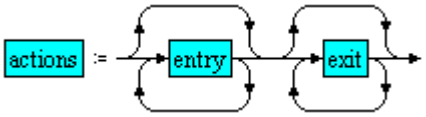
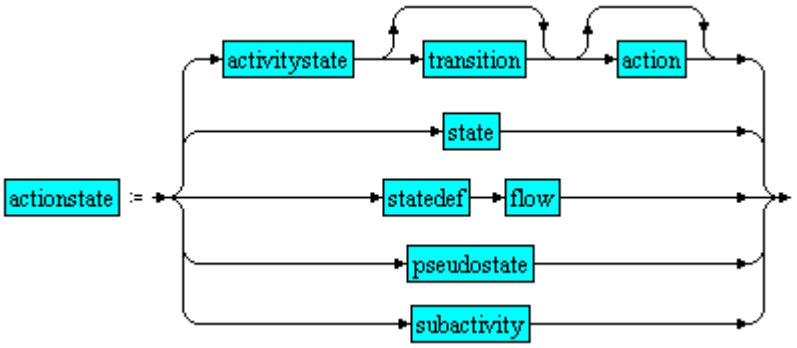
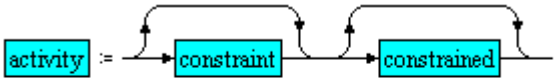
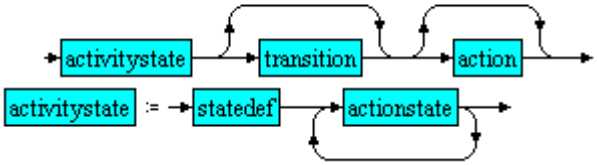
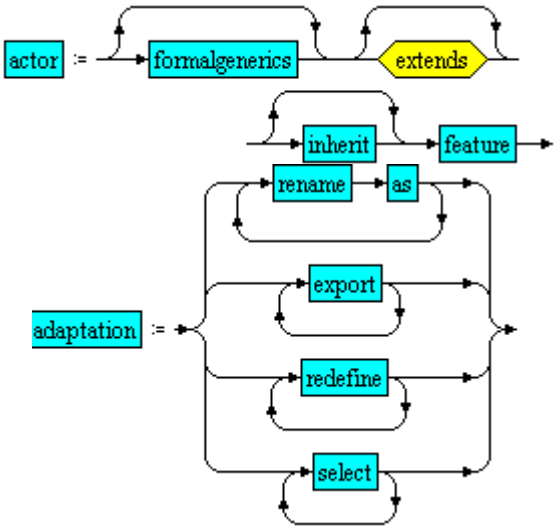
Then, if XML has all these advantages, why not propose XML to the SMIF RFP instead of UOL? Two of the main reasons are:

- XML has not been developed to be used heavily by humans without specific tools and, therefore, is not adequate for generic human to tool and human to human communication
    - Software engineers would not accept XML as a round-trip engineering language because it is too cumbersome to use without syntactic editors
    - XML obscures the embedded documents if read without a XML viewer
- XML only offers structural control, in the same sense as BNF, and, for example, is not susceptible to be extended as a procedural specification language or many object-oriented other requirements.

Therefore, we believe and, as such propose in this proposal the combination of both UOL and XML. This combination is presented in the following chapter.

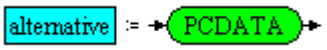
4.1.2 The mapping between UOL and XML

4.1.2.1 Graphical Definition of the Tags

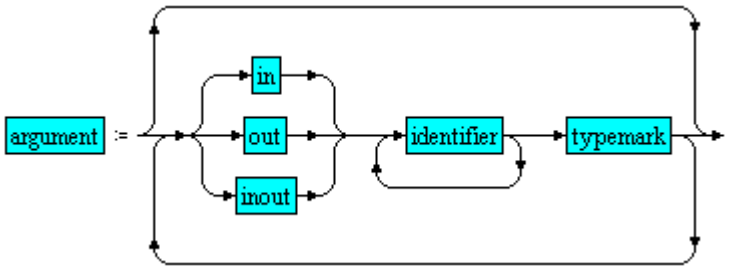
TAG	ATTRIBUTES	TAG'S STRUCTURE FOR AN XML UOL DOCUMENT
act		
action	name synchronous expression	
actions		
action state		
activity	name viewed	
activity state	partitioned	
actor	name viewed	
adaptation		

after

alternative



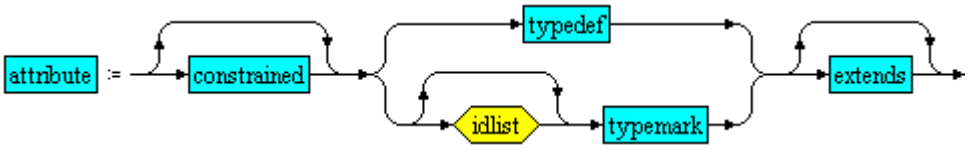
argument



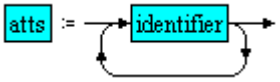
as

attribute

name
static
changes

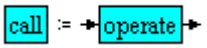


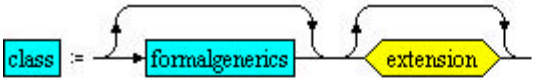
atts



BIT

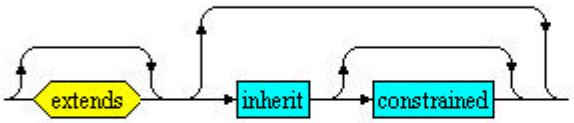
call

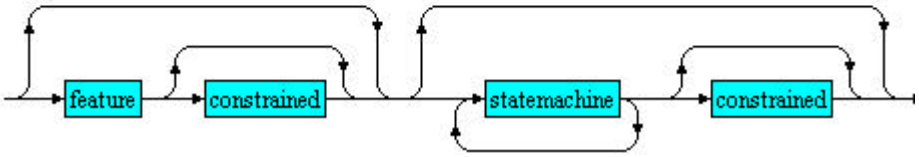




class

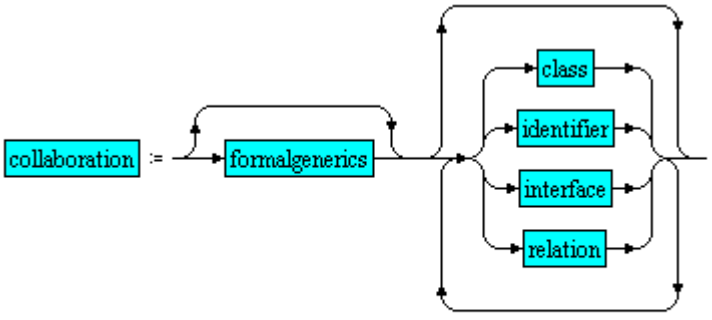
name
deferred
viewed

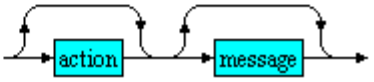




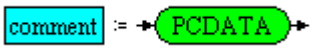
collaboration

name
viewed
implements

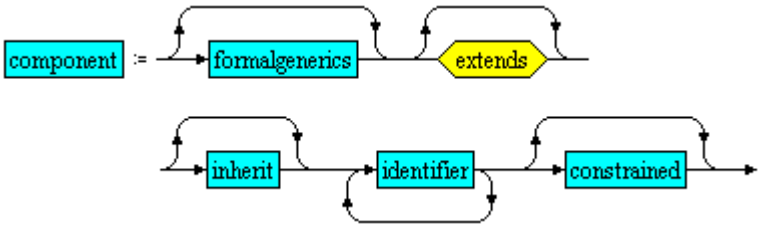




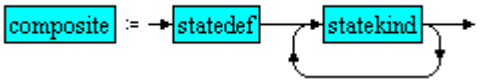
comment



component name viewed

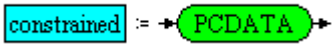


composite



constant

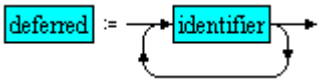
constrained



constraint name

create name

deferred



dependency

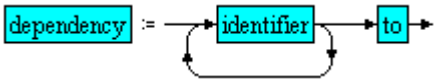
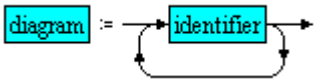
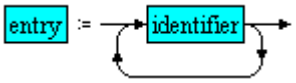


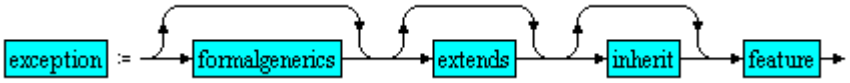
diagram type



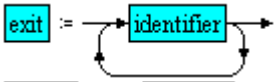
entry name



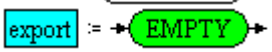
exception name viewed



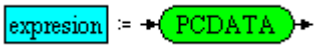
exit name



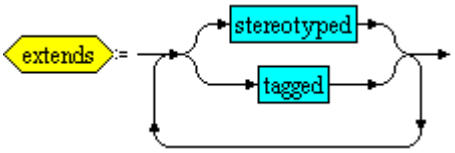
export name



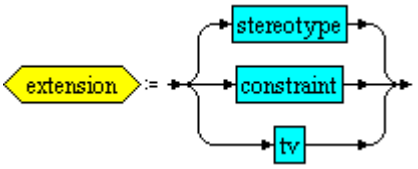
expresion



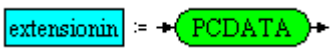
ENTITY extends



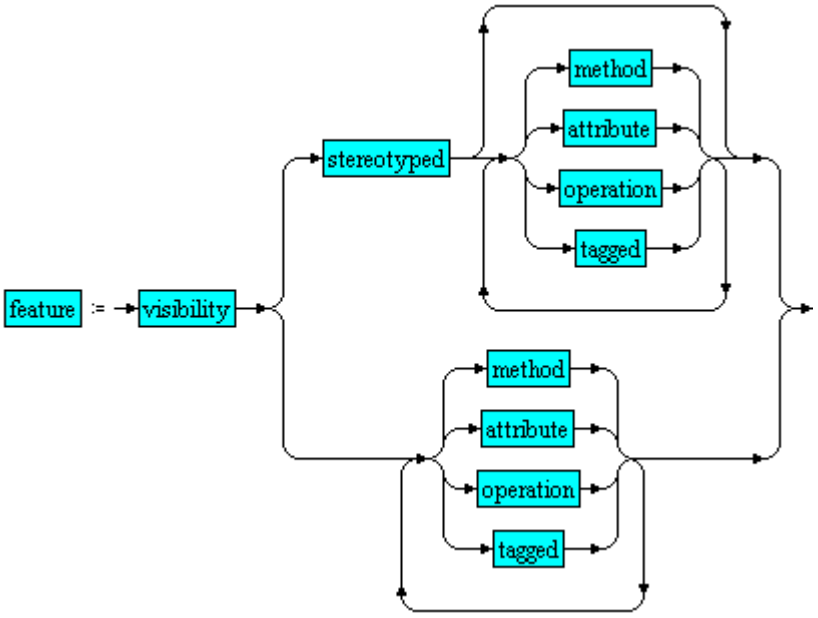
ENTITY extension



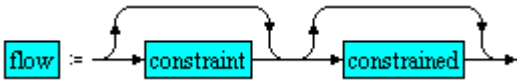
extensionin



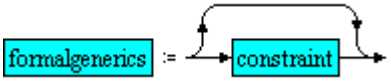
feature



flow classname statename



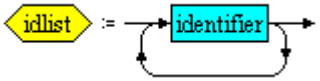
formal generics name



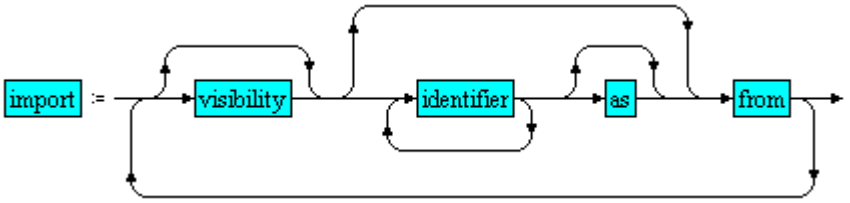
from

identifier name

ENTITTY idlist

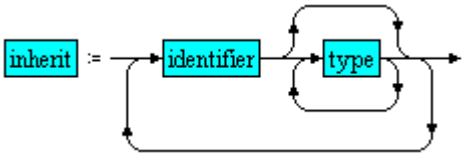


import

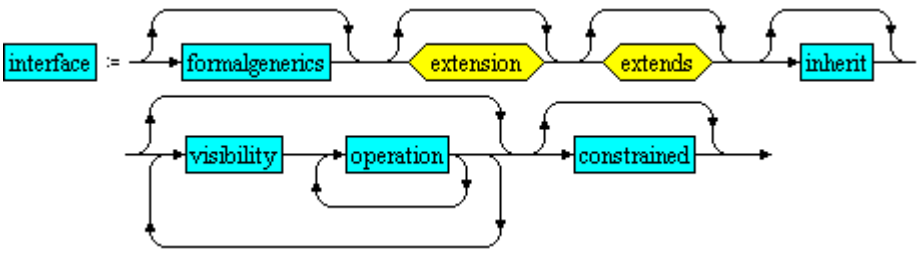


in

inherit name

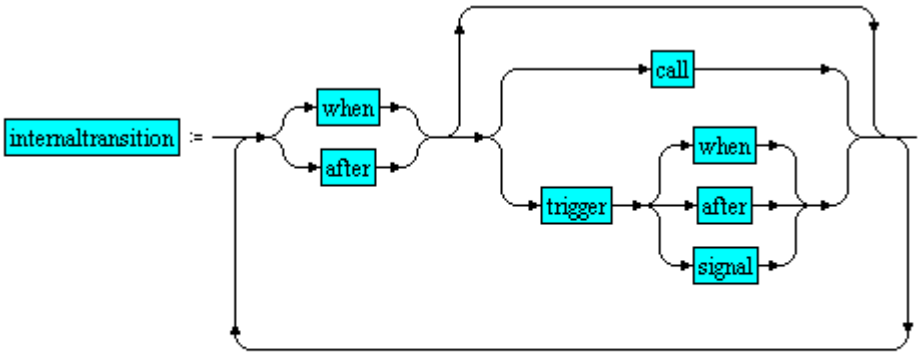


interface name viewed

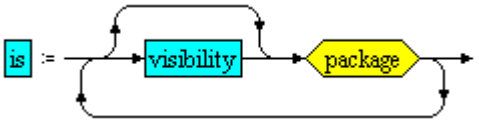


inout

internal transition

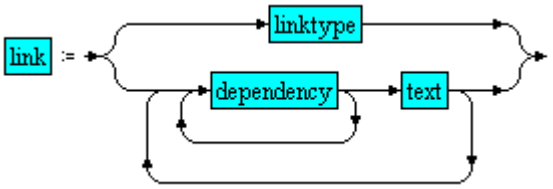


is



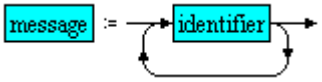
like

link `with`

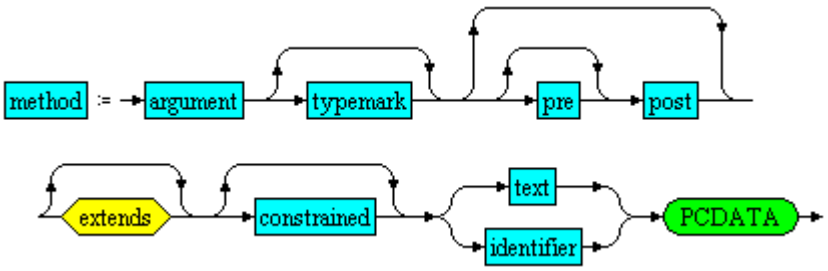


linktype `roleA` `roleB` `cardA` `cardB`

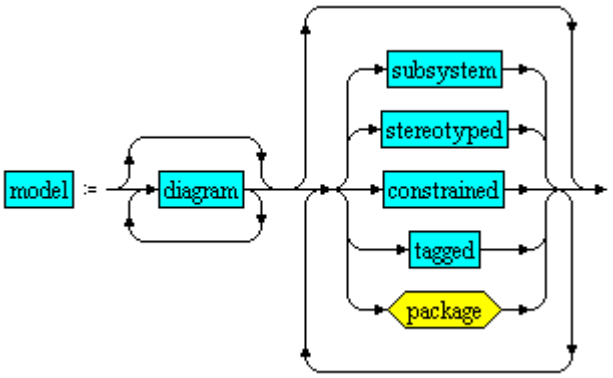
message



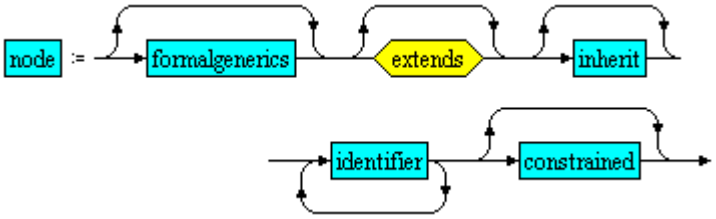
method `name` `deferred` `static`



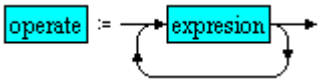
model name



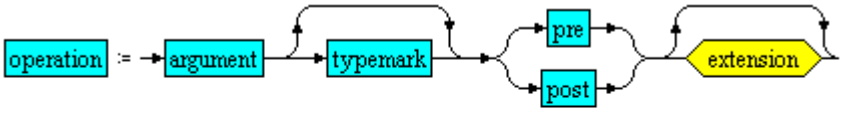
node name viewed



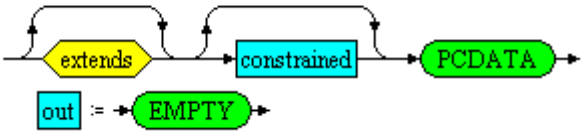
operate class operation



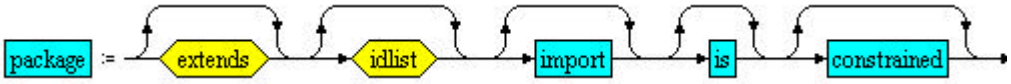
operation name static deferred



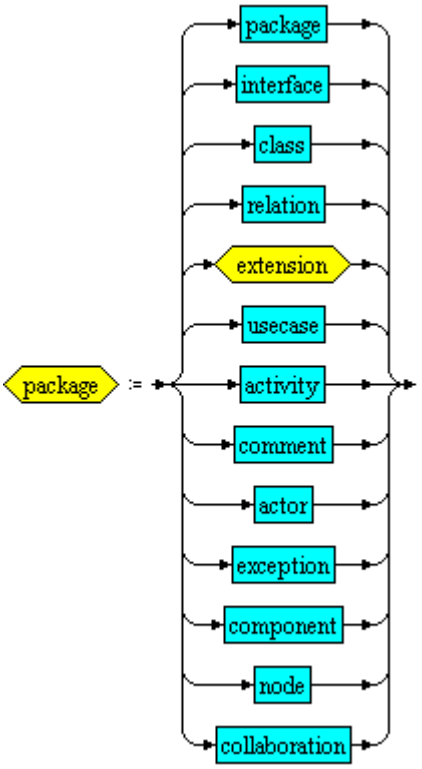
out



package name viewed



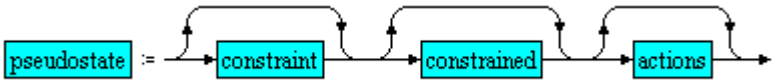
ENTITY
package



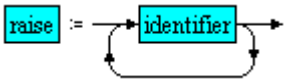
post

pre

pseudo state name type

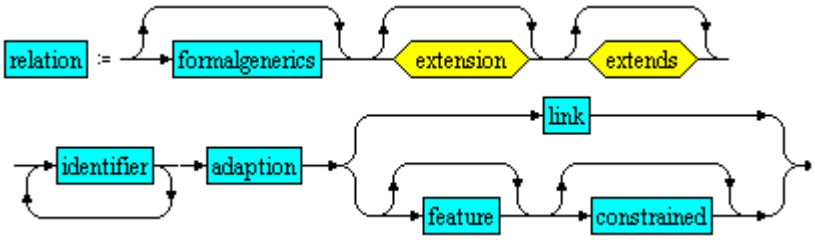


raise name



redefine name

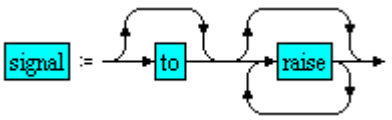
relation



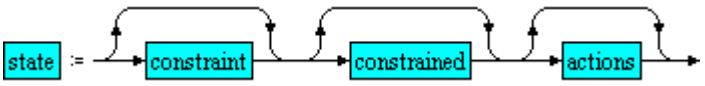
rename name

select name

signal name

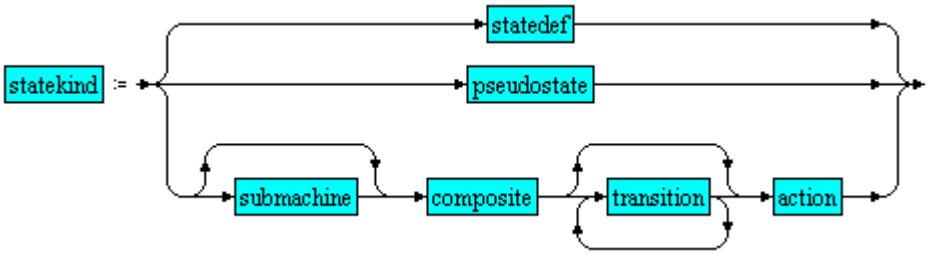


state name viewed



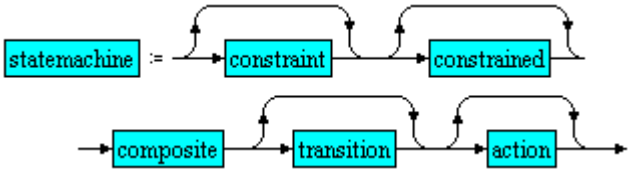
state def

state kind



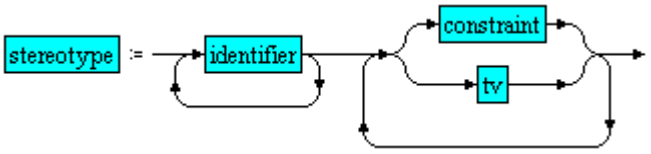
state machine

name
viewed

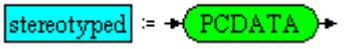


sterotype

name
of

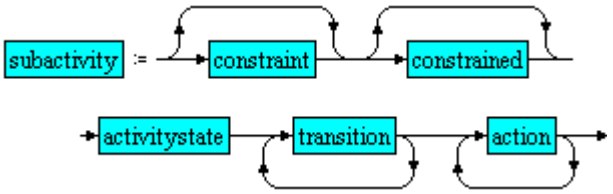


stereo typed



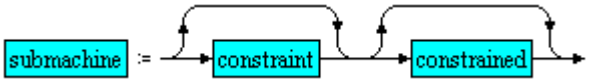
subactivity

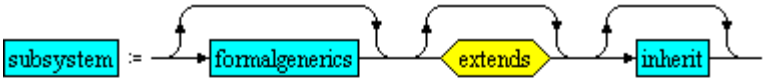
name
viewed



submachine

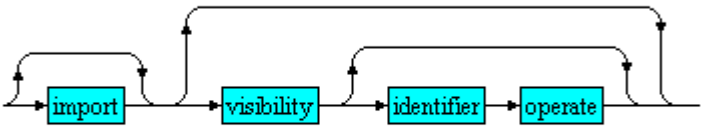
name
viewed

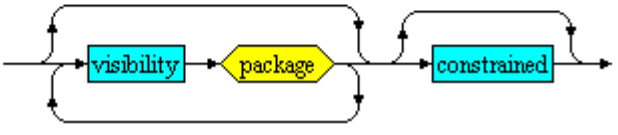




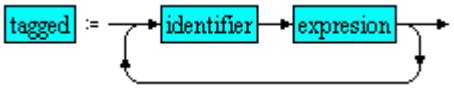
subsystem

name
viewed
deferred

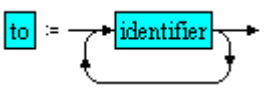




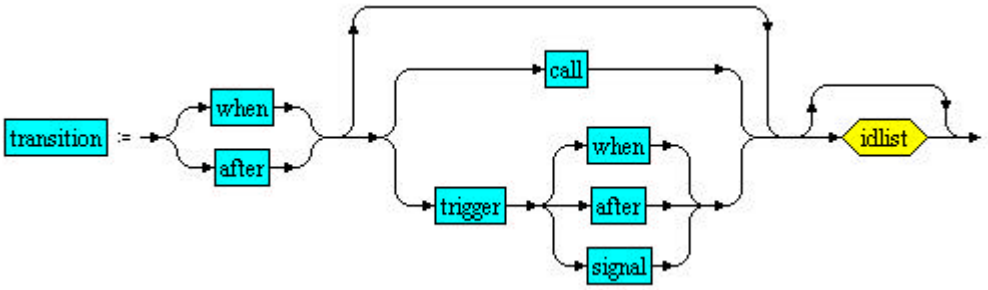
tagged



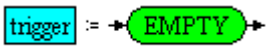
text



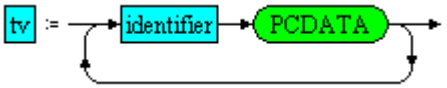
transition `from`
`to`

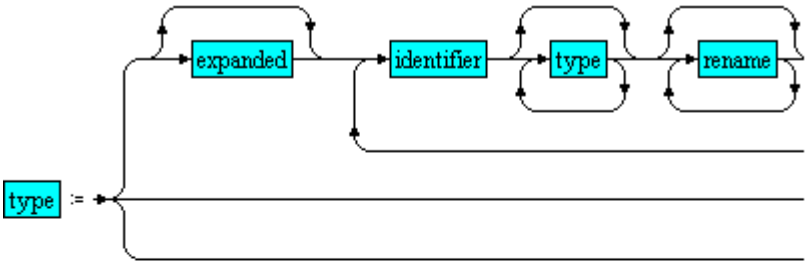


trigger

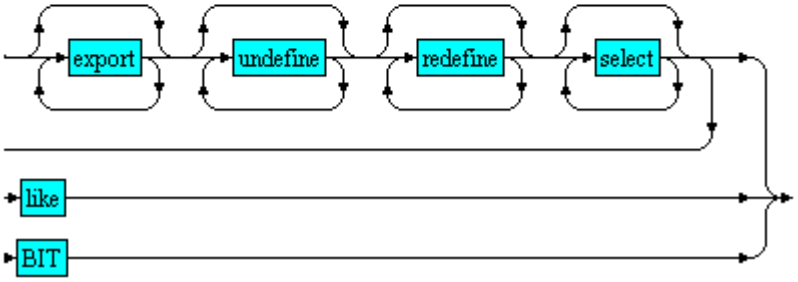


tv

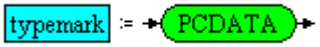




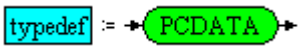
type

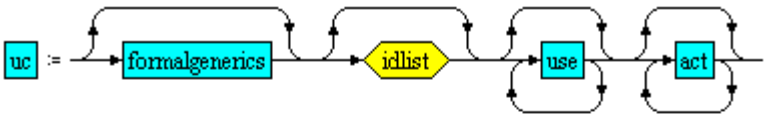


typemark `like`

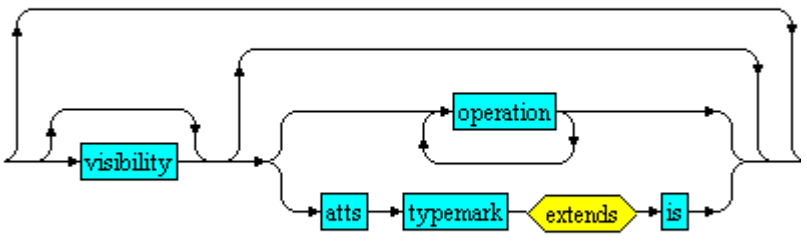


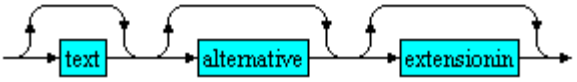
typedef



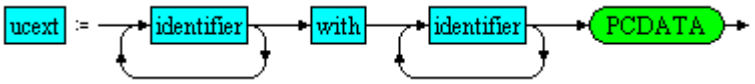


uc

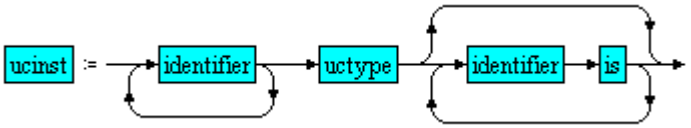




ucext



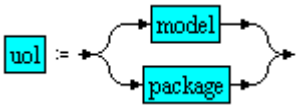
ucinst name



uctype name

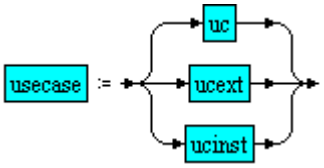
undefine

uol

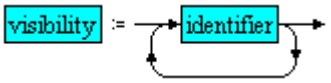


use

usecase name



visibility



with

when

4.1.2.2 Document Type Definition (DTD) of UOL

```
<!ENTITY % extends " (stereotyped | tagged)+ " >
<!ENTITY % extension " stereotype | constraint | tv " >
<!ENTITY % package " (package | interface |class | relation | stereotype |
          constraint | tv | usecase | activity | comment | actor | exception |
          component | node | collaboration) " >
<!ENTITY % idlist " identifier+ " >
<!ELEMENT act  (#PCDATA) >
<!ELEMENT action  (to* , (operate | signal)? ,(call | create | text)) >
<!ATTLIST action
          name      ID          #REQUIRED
          synchronous    (YES|NO)    #IMPLIED
          expression     CDATA #IMPLIED>
<!ELEMENT actions  (entry* , exit*) >
<!ELEMENT actionstate  ((activitystate , transition? , actions?) | state |
          (statedef , flow) | pseudostate | subactivity) >
<!ELEMENT activity  (constraint? , constrained? , activitystate , transition?
          , action?) >
<!ATTLIST activity
          name     ID          #REQUIRED
          viewed   CDATA       #IMPLIED>
<!ELEMENT activitystate  (statedef , actionstate+) >
```

(UOL 1.2)

```
<!ATTLIST activitystate
          partitioned   CDATA       #IMPLIED>
<!ELEMENT actor  (formalgenerics? , (%extends;)? , inherit? , feature) >
<!ATTLIST actor
          name          ID          #REQUIRED
          viewed  CDATA       #IMPLIED>
<!ELEMENT adaptation  ((rename , as)+ | export+ | redefine+ | select+) >
<!ELEMENT after  (#PCDATA) >
<!ELEMENT alternative  (#PCDATA) >
<!ELEMENT argument  ((in | out | inout) , identifier+ , typemark)* >
<!ELEMENT as  (#PCDATA) >
<!ELEMENT attribute  (constrained? ,(typedef | ((%idlist;)? , typemark)) ,
          (%extends;)?) >
<!ATTLIST attribute
          name ID #REQUIRED
          static (NO|YES) #FIXED "NO"
          changes (addonly|frozen) #IMPLIED>
<!ELEMENT atts  (identifier)+ >
<!ELEMENT BIT  (#PCDATA) >
<!ELEMENT call  (operate) >

<!ELEMENT class (formalgenerics? , (%extension;)? , (%extends;)? , ((inherit ,
          constrained?)? , (feature , constrained?)? , (statemachine+ ,
          constrained?)?)?) >
<!ATTLIST class
          name ID #REQUIRED
          deferred (no|yes) #IMPLIED
          viewed CDATA #IMPLIED >
<!ELEMENT collaboration  (formalgenerics? , (class | identifier | interface |
          relation)* , action? , message?) >
<!ATTLIST collaboration
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED
          implements    ID          #IMPLIED>
<!ELEMENT comment  (#PCDATA) >
<!ELEMENT component  (formalgenerics? , (%extends;)? , inherit? , identifier+
          ,constrained?) >
<!ATTLIST component
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED>
<!ELEMENT composite  (statedef , statekind+) >
<!ELEMENT constant  (#PCDATA) >
<!ELEMENT constrained  (#PCDATA) >
<!ELEMENT constraint  (#PCDATA) >
<!ATTLIST constraint
          name          ID          #REQUIRED>
<!ELEMENT create  EMPTY >
<!ATTLIST create
          name          ID          #REQUIRED>
<!ELEMENT deferred  (identifier)+ >
<!ELEMENT dependency  (identifier+ , to) >
<!ELEMENT diagram  (identifier)+ >
<!ATTLIST diagram
          type          ID          #REQUIRED>
<!ELEMENT entry  (identifier)+ >
<!ATTLIST entry
```

```
          name          ID          #REQUIRED>
<!ELEMENT exception  (formalgenerics? , (%extends;)? , inherit? , feature) >
<!ATTLIST exception
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED>

<!ELEMENT exit  (identifier)+ >
<!ATTLIST exit
          name          ID          #REQUIRED>
<!ELEMENT export  EMPTY >
<!ATTLIST export
          name          CDATA       #REQUIRED>
<!ELEMENT expresion  (#PCDATA) >
<!ELEMENT extensionin  (#PCDATA) >
<!ELEMENT feature  (visibility , ((stereotyped , (method | attribute |
          operation | tagged)*) | (method | attribute | operation | tagged)+))
          >
<!ELEMENT flow  (constraint? , constrained?) >
<!ATTLIST flow
          classname           ID          #REQUIRED
          statename           CDATA       #REQUIRED>
<!ELEMENT formalgenerics  (constraint)? >
<!ATTLIST formalgenerics
          name          ID          #REQUIRED>
<!ELEMENT from  (#PCDATA) >
<!ELEMENT identifier  EMPTY >
<!ATTLIST identifier
          name          ID          #REQUIRED>
<!ELEMENT import  (visibility? , (identifier+ , as?)? , from)+ >
<!ELEMENT in  EMPTY>
<!ELEMENT inherit  (identifier , type*)+ >
<!ATTLIST inherit
          name          ID          #REQUIRED>
<!ELEMENT inout  EMPTY >
<!ELEMENT interface  (formalgenerics? , (%extension;)? , (%extends;)? ,
          inherit? , ((visibility , operation*)* , constrained?)) >
<!ATTLIST interface
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED>
<!ELEMENT internaltransition  (((when | after) , (call | (trigger , (when |
          after | signal)))?)+ , (%idlist;)?) >
<!ELEMENT is  (visibility? , %package;)+ >
<!ELEMENT like  (#PCDATA) >

<!ELEMENT link  (linktype | (dependency+ , text)+) >
<!ATTLIST link
          with          ID          #REQUIRED>
<!ELEMENT linktype  EMPTY >
<!ATTLIST linktype
          roleA         ID          #REQUIRED
          roleB         ID          #REQUIRED
          cardA         CDATA       #IMPLIED
          cardBed       CDATA       #IMPLIED>
<!ELEMENT message  (identifier)+ >
<!ELEMENT method  (argument , typemark? , (pre? , post)? , (%extends;)? ,
          constrained? , (text | identifier) , PCDATA) >
```

(UOL 1.2)

```
<!ATTLIST method
          name          ID          #REQUIRED
          deferred(YES|NO)    #IMPLIED
          static        (YES|NO)    #IMPLIED>
<!ELEMENT model  (diagram* , (subsystem | stereotyped | constrained | tagged |
          %package;)*) >
<!ATTLIST model
          name          ID          #REQUIRED>
<!ELEMENT node  (formalgenerics? , (%extends;)? , inherit? , identifier+ ,
          constrained?) >
<!ATTLIST node
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED>
<!ELEMENT operate  (expresion)+ >
<!ATTLIST operate
          class         ID          #REQUIRED
          operation     ID          #REQUIRED>
<!ELEMENT operation  (argument , typemark? , (pre | post) , (%extension;)? ,
          (%extends;)? , constrained? , PCDATA) >
<!ATTLIST operation
          name          ID          #REQUIRED
          static        (YES|NO)    #IMPLIED
          deferred(YES|NO)    #IMPLIED>
<!ELEMENT out  EMPTY >
<!ELEMENT package  ((%extends;)? , (%idlist;)? , import? , is? , constrained?)
          >
<!ATTLIST package
          name          ID          #REQUIRED
          viewed        ID          #IMPLIED>
<!ELEMENT post  (#PCDATA) >
<!ELEMENT pre  (#PCDATA) >

<!ELEMENT pseudostate  (constraint? , constrained? , actions?) >
<!ATTLIST pseudostate
          type    (deephistory|shallowhistory|initial|final|join|fork|branch)
          #REQUIRED
          name          ID          #REQUIRED>
<!ELEMENT raise  (identifier)+ >
<!ATTLIST raise
          name          ID          #REQUIRED>
<!ELEMENT redefine  EMPTY >
<!ATTLIST redefine
          name          CDATA       #REQUIRED>
<!ELEMENT relation  (formalgenerics? , (%extension;)? , (%extends;)? ,
          identifier+ , adaptation , (link | (feature? , constrained?))) >
<!ELEMENT rename  EMPTY >
<!ATTLIST rename
          name          CDATA       #REQUIRED>
<!ELEMENT select  EMPTY >
<!ATTLIST select
          name          CDATA       #REQUIRED>
<!ELEMENT signal  (to? , raise*) >
<!ATTLIST signal
          name          ID          #REQUIRED>
<!ELEMENT state  (constraint? , constrained? , actions?) >
<!ATTLIST state
```

```
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED>
<!ELEMENT statedef  (state , internaltransition* , deferred?) >
<!ELEMENT statekind  (statedef | pseudostate | (submachine? , composite ,
          transition* , action)) >
<!ELEMENT statemachine  (constraint? , constrained? , composite , transition?
          , action?) >
<!ATTLIST statemachine
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED>
<!ELEMENT stereotype  (identifier+ , (constraint | tv)+) >
<!ATTLIST stereotype
          name          ID          #REQUIRED
          of      ID          #REQUIRED>
<!ELEMENT stereotyped  (#PCDATA) >


<!ELEMENT subactivity  (constraint? , constrained? , activitystate ,
          transition+ , action+) >
<!ATTLIST subactivity
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED>
<!ELEMENT submachine  (constraint? , constrained?) >
<!ATTLIST submachine
          name          ID          #REQUIRED
          viewed        CDATA       #IMPLIED>
<!ELEMENT subsystem  (formalgenerics? , (%extends;)? , inherit? , import? ,
          (visibility , (identifier , operate)?)? , (visibility , %package;)*
          , constrained?) >
<!ATTLIST subsystem
          name          ID          #REQUIRED
          deferred(YES|NO)    #IMPLIED
          viewed        CDATA       #IMPLIED>
<!ELEMENT tagged  (identifier , expresion)+ >
<!ELEMENT text  (#PCDATA) >
<!ELEMENT to  (identifier)+ >
<!ELEMENT transition  ((when | after) , (call | (trigger , (when | after |
          signal)))? , (%idlist;)?) >
<!ATTLIST transition
          from          ID          #REQUIRED
          to      ID          #REQUIRED>
<!ELEMENT trigger  (#PCDATA) >
<!ELEMENT type  ((expanded? , (identifier , type* , rename* , export* ,
          undefine* , redefine* , select*)+) | like | BIT) >
<!ELEMENT typedef  (#PCDATA) >
<!ELEMENT typemark  (#PCDATA) >
<!ATTLIST typemark
          like (NO|YES) #FIXED "NO">
<!ELEMENT tv  (identifier , #PCDATA)+ >
<!ELEMENT uc  (formalgenerics? , (%idlist;)? , use* , act* , (visibility? ,
            (operation+ | (atts , typemark , (%extends;) , is))?)? , text? ,
            alternative? , extensionin?) >
<!ELEMENT ucext  (identifier+ , with , identifier+ , #PCDATA) >
<!ELEMENT ucinst  (identifier+ , uctype , (identifier , is)*) >
<!ATTLIST ucinst
          name          ID          #REQUIRED>
```

```
<!ELEMENT uctype  EMPTY >
<!ATTLIST uctype
          name          ID          #REQUIRED>
<!ELEMENT undefine  (#PCDATA) >
<!ELEMENT uol  (model | package) >
<!ELEMENT use  (#PCDATA) >
<!ELEMENT usecase  (uc | ucext | ucinst) >
<!ATTLIST usecase
          name          ID          #REQUIRED>
<!ELEMENT visibility  (identifier)+ >
<!ELEMENT when  (#PCDATA) >
<!ELEMENT with  (#PCDATA) >
```

4.1.2.3 Examples

example1.uol

```
model Example0
    diagrams
        MainD,MainE:StaticDiagram;
        PersonD:StateDiagram
    end -- diagrams
end -- model
```

example1.xml

```
<?XML VERSION="1.0" ?>
<!DOCTYPE uol SYSTEM "uol.dtd">
<uol>
      <model name="Example0">
           <diagram type="StaticDiagram">
                 <identifier name="MainD"/>
                 <identifier name="MainE"/>
           </diagram>
            <diagram type="StateDiagram">
                 <identifier name="PersonD"/>
           </diagram>
      </model>
</uol>
```

example2.uol

```
model aModel
     package aPMain is {any}
           class SCHEMA
                 -- features of schema ommited
           end
           class aClass
                 -- features of aClass ommited
           end
           stereotype EXPRESS_SCHEMA of SCHEMA end
```

```
                stereotype aStereoType of aClass end
                stereotype typeA of aClass end
                class Schema_Class
                      stereotyped with EXPRESS_SCHEMA
                      feature {any}
                            a:integer;
                            b:integer is 3;
                            c [1..3,6..*] constrained by {aConstraint}
                                  :integer is {2,3,4}
                                  stereotyped with aStereoType
                      end -- feature
                end -- class
                class aClass2
                      feature {any}
                            stereotyped with typeA
                      end -- feature
                end -- class
          end -- package
     end -- model
```

example2.xml

```
     <?XML VERSION="1.0" ?>
     <!DOCTYPE uol SYSTEM "uol.dtd">
     <uol>
     <model name="aModel">
          <package name="pMain">
                <visibility> <identifier name="ANY"/>
                </visibility>
                <class name="SCHEMA"> </class>
                <class name="aClass"> </class>
                <stereotype name="EXPRESS_SCHEMA" of="SCHEMA">
                </stereotype>
                <stereotype name="aStereoType" of="aClass">
                </stereotype>
                <stereotype name="typeA" of="aClass"> </stereotype>
                <class name="Schema_Class">
                      <stereotyped>EXPRESS_SCHEMA</stereotyped>
                      <feature>
                            <visibility> <identifier name="ANY"/>
                            </visibility>
                            <attribute name="a">
                                  <typemark>integer</typemark>
                            </attribute>
                            <attribute name="b">
                                  <typedef>integer is 3</typedef>
                            </attribute>
                            <attribute name="c">
                                  <constrained>{aConstraint}
                                  </constrained>
                                  <typedef>
                                        <typemark>integer
                                        </typemark>
                                  [1..3,6..*] is {2,3,4}
                                  </typedef>
                                  <stereotyped>aStereoType
```

(UOL 1.2)

```
                                    </stereotyped>
                              </attribute>
                        </feature>
                  </class>
                  <class name="aClass2">
                        <feature>
                              <visibility>
                                    <identifier name="ANY"/>
                              </visibility>
                              <stereotyped>typeA</stereotyped>
                        </feature>
                  </class>
            </package>
      </model>
      </uol>
```

example3.uol

```
model Example3
    -- Subsystem declaration
    deferred subsystem adSubsystem
        -- extension use
        -- inheritance
        import
            {any} anElement as thisElement from aPackage,
            anotherElement from aPackage2,
            from aPackage3
    end -- subsystem adSubsystem
    package AllElements is
        -- Package declaration
        package aPackage
        end -- package aPackage
        -- Interface declaration
        interface anInterface
           feature {any}
                -- only operations
               deferred static anOperation(aParam:aType):aReturnType
                   {precondition: aConstraint}
                   {postcondition: aConstraint}
                   constrained by {aConstraint}
                   is text "Specification"
               end
        end – interface anInterface
        -- Class declaration
        class Person viewed with MainD
            feature {any}
                isMarried, isUnemployed:Boolean;
                birthDate:Date;
                age:Integer;
                firstName,lastName:String;
                sex: unique { male,female };
                    deferred income(d:Date):Integer
                is text"Incoming operation"
             end
           constrained by { self.age>=0 }
        end – Class Person
```

```
            -- Relation declaration
            relation job
                link Person[0..*], Company[0..*] with Marriage
                feature {Person}
                    stereotyped with UMLAssociationEnd;
                    with tag values (<AssociationEndName , employee>)
                end
                feature {Company}
                    stereotyped with UMLAssociationEnd;
                    with tag values (<AssociationEndName , employer>);
                    deferred jobAmount(accountNumber:Integer)
                    is text "Amounting count"
                end
                constrained by
                   { self.employee->size <=50 }
                    -- rest of constraints ommited
            end -- Relation job
            -- Stereotype declaration
            stereotype aStereotype of aBaseClass viewed as 'anIcon.gif'
                inherit aName1(aDiscriminator),aName2
                tag values
                    AssociationEndName
                    IsNavigable is true
                    IsOrdered is false
                    Aggregation is 'none'
                  end -- tag values
                constraint
                    aConstraint1 is { text "This is a constraint" }
                    aConstraint2 is { aConstraintDef }
                end -- constraint
            end -- stereotype aStereotype
        end -- package AllElements
    end -- model Example3
```

example3.xml

```
    <?XML VERSION="1.0" ?>
    <uol>
    <model name="Example3">
         <subsystem name="adSubsystem" deferred="yes">
               <import>
                     <visibility>any</visibility>
                     <identifier name="anElement"/>
                     <as>thisElement</as>
                     <from>aPackage</from>
               </import>
               <import>
                     <identifier name="anotherElement"/>
                     <from>aPackage2</from>
               </import>
               <import><from>aPackage3</from></import>
       </subsystem>
         <package name="aPackage">
         </package>
       <package name="AllElements">
```

```
            </package>
          <interface name="anInterface">
              <visibility>any</visibility>
                <operation name="anOperation" static="yes">
                   <argument>
                      <out/><identifier name="aReturnType"/>
                      </argument>
                      <argument>
                            <in/><identifier name="aParam"/>
                            <typemark>aType</typemark>
                         </argument>
                    <pre>aConstraint</pre>
                    <post>aConstraint2</post>
                          <constraint>{aConstraint3}</constraint>
                     Specification
                    </operation>
          </interface>

            <class name="Person" viewed="MainD">
                <feature>
                   <visibility>any</visibility>
                   <attribute name="isMarried">
                         <typemark>boolean</typemark>
                           </attribute>
                   <attribute name="isUnemployed">
                              <typemark>boolean</typemark>
                            </attribute>
                   <attribute name="birthDate">
                              <typemark>date</typemark>
                          </attribute>
                   <attribute name="age">
                              <typemark>boolean</typemark>
                          </attribute>
                   <attribute name="firstName">
                              <typemark>string</typemark>
                          </attribute>
                   <attribute name="lastName">
                              <typemark>string</typemark>
                           </attribute>
                   <attribute name="sex">
                              <typedef>unique enum={male,female}</typedef>
                          </attribute>
                   <operation name="income" deferred="Yes" type="Integer">
                       <parameter name="d" type="Date"/>
                       Incoming operation
                   </operation>
              </feature>

              <constraint> {self.age>=0} </constraint>
          </class>
          <relation name="job">
              <feature>
                   <visibility>Person</visibility>
                 <stereotyped>UMLAssociationEnd</stereotyped>
                 <tagged>
                     <identifier name="AssociationEndName"/>
```

```
                husband
              </tagged>
            </feature>
          <feature>
                  <visibility>Company</visibility>
             <stereotyped>UMLAssociationEnd</stereotyped>
             <tagged>
                 <identifier name="AssociationEndName"/>
                 employer
             </tagged>
           </feature>
          <constraint>{self.employee->size %le=50}</constraint>
              <link roleA="Person" cardA="[0..*]" roleB="Company"
              cardb="[0..*]" with="Marriage"/>
    </relation>
      <stereotype name="aStereotype" of="aBaseClass">
           <identifier name="anIcon.gif"/>
        <tv>
            <identifier name="AssociationEndName"/>
            husband
        </tv>

        <tv>
            <identifier name="IsNavigable"/>
            true
        </tv>
          <tv>
            <identifier name="IsOrdered"/>
            false
        </tv>
             <tv>
            <identifier name="Aggregation"/>
            %quot none %quot
        </tv>
             <constraint name="aConstraint1">
            text \"This is a constraint\"
        </constraint>
             <constraint name="aConstraint2">
             aConstrainDef
        </constraint>
     </stereotype>
  </model>
  </uol>
```

5 Mappings

5.1 The mapping between UOL and MOF

5.1.1 Direct mapping

MOF Meta-metamodel	UOL
Association (binary)	Association (n-ary)
NA	AssociationClass
AssociationEnd	AssociationEnd
Attribute	Attribute
BehavioralFeature	BehavioralFeature
Class	Class
Classifier	Classifier
Constraint	Constraint
DataType	DataType
/ dependsOn (association)	Dependency (class)
Exception	Exception
Feature	Feature
GeneralizableElement	GeneralizableElement
generalizes (association)	Generalization (class)
Generalization	Generalization
Interface	Interface Class (as Interface)
ModelElement	ModelElement
Reference	NA
Constant	Attribute
Namespace	Namespace
Operation	Operation
Package	Package
Parameter	Parameter
StructuralFeature	StructuralFeature
Class (as Type)	Type (stereotype)

MOF META-METAMODEL	UOL
AggregationKind	AggregationKind
CORBA Boolean	Boolean
CORBA Enum	unique
NameType	Expression
CORBA Short, Long, Unsigned Short, Unsigned Long, Double,	Integer

Octet, Float	
List, Set	List
MultiplicityKind	Multiplicity
NameKind	Name
DirectionKind	OperationDirectionKind
DependencyKind (enum)	dependencies (reified as classes)
ScopeKind	ScopeKind
CORBA String, Char	String
CORBA Time Service Data Types	Time
TypeDef	NA
CORBA Any	TextMultiline
VisibilityKind	VisibilityKind

5.1.2 Support for meta-model extensions

MOF can be extended via descendents of the MOF class and UOL can be extended via tagged values and stereotypes.

The reason for the way MOF is extended is that a meta-model describes such concepts as instances of the concepts defined in the meta-meta-model. Therefore, new concepts will be descendents in MOF of the only suitable MOF entity, the MOF class.

Being that MOF's extension mechanism is different from the concepts defined in UOL, these concepts will be mapped to a MOF class through the UOL's extension mechanisms. The exact mapping is pending.

5.2 The mapping between UOL and CDIF

5.2.1 Introduction

5.2.1.1 Document structure

This document explains the translation from a CDIF Transfer to UOL code. The CDIF structures are presented in the same order as they appear in the EIA/IS-109 document standard from Electronic Industries Association. For every CDIF element, we present an explanation of it according to the standard referenced followed by its UOL mapping, where we justify the concrete mapping. Then, we give the part of the CDIF grammar corresponding to the CDIF element and a mapping example extracted from the standards examples. These examples are part of the complete CDIF Transfer presented at the final chapter.

Finally, we translate a complete CDIF Transfer to UOL, giving also its UML graphical representation.

5.2.1.2 Structure of a CDIF Transfer

A CDIF Transfer consists of two elements, the TransferEnvelope and the TransferContents. They are detailed in each of the corresponding chapters.

The general syntax of a CDIF Transfer **is** as follows:

```
<CDIFTransfer>::=         <TransferEnvelope>
                  <TransferContents>
```

[extracted from EIA/IS-109,page 8]

5.2.2 Transfer Envelope

5.2.2.1 Introduction

The Transfer Envelope consists of the CDIF Signature, the Syntax Identifier and the Encoding Identifier.

[extracted from EIA/IS-109,page 8]

5.2.2.2 UOL mapping

The Transfer Envelope maps to UOL as a comment.

The information of the Transfer Envelope has no meaning in an UOL source code because it refers to the syntax standard and the encoding standard used in the source CDIF file.

5.2.2.3 Grammar

```
<TransferEnvelope> ::=    <CDIFSignature> , <SyntaxIdentifier> ,
                          <EncodingIdentifier>
<SyntaxIdentifier> ::=    SYNTAX <TransferEnvelopeSpace> <SyntaxId>
                          <TransferEnvelopeSpace> <SyntaxVersion>
<EncodingIdentifier> ::=  ENCODING <TransferEnvelopeSpace>
                          <EncodingId> <TransferEnvelopeSpace>
                          <EncodingVersion>
<CDIFSignature> ::=       CDIF
<SyntaxId> ::=            <TransferEnvelopeString>
<SyntaxVersion> ::=       <TransferEnvelopeString>
<EncodingId> ::=                  <TransferEnvelopeString>
<EncodingVersion>         ::=     <TransferEnvelopeString>
```

[extracted from EIA/IS-109,page 35]

5.2.2.4 Example

CDIF Source Example:

```
 CDIF , SYNTAX “SYNTAX.1” “02.00.00” , ENCODING “ENCODING.1” “02.00.00”
```

[extracted from EIA/IS-110 Extract,page 26]

UOL Mapping Example:

```
-- CDIF,SYNTAX “SYNTAX.1” “02.00.00”,ENCODING “ENCODING.1” "02.00.00"
-- Transfer Contents
```

5.2.3 Transfer Contents

5.2.3.1 Introduction

The first level of the grammar of the Transfer Contents is the same for any CDIF Transfer, its structure is as follows:

```
<TransferContents>::=             <HeaderSection >
                                  <MetaModelSection>
                                  [<ModelSectionClause>]
```

[extracted from EIA/IS-109,page 8]

5.2.3.2 Header Section

5.2.3.2.1 Introduction

The Header Section defines information that applies to the whole transfer.
[extracted from EIA/IS-109, page 9]

5.2.3.2.2 UOL mapping

The Header Section mappes to UOL as a comment.

The information of the Header Section has no meaning in an UOL source code because it specifies summary information about the transfer, in the form of a number of items.

5.2.3.2.3 Grammar

```
<HeaderSection> ::=       <OpenScope> <HeaderKeyword>
                          <SummaryClause> <CloseScope>
<SummaryClause> ::=       <OpenScope> <SummaryKeyword>
                          [<IdentifierValuePair>]...
                          <CloseScope>
<IdentifierValuePair> ::= <OpenScope> <SummaryIdentifier>
                          <StringValue> <CloseScope>
<SummaryIdentifier> ::=   <Identifier>
<StringValue> ::=                 <String>
```

[extracted from EIA/IS-109, pages 9,10]

5.2.3.2.4 Example

CDIF Source Example:

```
(: HEADER
       (:SUMMARY
            (ExporterName      “CASE Genius”)
            (ExporterVersion   “01.00.00”)
            (ExportDate        “1991/04/01”)
            (ExporterTime      “07:00:00”)
            (PublisherName     “Mary Lomas”)
       )
)
```

[extracted from EIA/IS-110 Extract,page 26]

UOL Mapping Example :

```
--SUMMARY
--     (ExporterName      “CASE Genius”)
--     (ExporterVersion   “01.00.00”)
--     (ExportDate        “1991/04/01”)
--     (ExporterTime      “07:00:00”)
--     (PublisherName     “Mary Lomas”)
```

5.2.3.3 Meta-model Section

5.2.3.3.1 Introduction

The Meta-model Section of the Transfer consists of references to standardized Subject Areas, followed by extensions to the Meta-Model. The Meta-model for the transfer, known as the

"Working Meta-model", is defined by the set of meta-meta-entity and meta-meta-relationship instances that are used in any of the referenced Subject Areas, plus those added by extensions.

Its grammar is as follows:

```
<MetaModelSectionClause>::=        <OpenScope> <MetaModelKeyword>
                                   <CDIFSubjectAreaReferenceClause>...
                                   [<MetaModelExtensionClause>]...
                                   <CloseScope>
```

[extracted from EIA/IS-109,pages 11,12]

5.2.3.3.2 CDIF Subject Area Reference Clause

5.2.3.3.2.1 Introduction

This section identifies the standardized CDIF Subject Areas that should be used by the importer when interpreting model data. The appropriate version of each of these Subject Areas is also identified (as defined in the relevant Subject Area standard).

[extracted from EIA/IS-108,page 14]

5.2.3.3.2.2 UOL mapping

Subject Area References are mapped as an UOL import statement. Subject Areas should be mapped as UOL packages.

Each package defined from a SubjectAreaReference, contains a non-instantiable class, called "VersionNumber<SubjectAreaName>", with one attribute, "VersionNumber", that contains the version number of the imported Subject Area.

Mapping Subject Areas as UOL packages allows the model to import them and gives support to the underlying structure, grouping the model and the meta-model in a package each. The meta-model contains a number of packages, corresponding to the referenced Subject areas, and the package containing the extensions.

5.2.3.3.2.3 Grammar

```
<CDIFSubjectAreaReferenceClause>::=          <OpenScope>
                                             <SubjectAreaReferenceKeyword>
                                             <SubjectAreaName>
                                             <OpenScope>
                                             <VersionNumberKeyword>
                                             <SubjectAreaVersionNumber>
                                             <CloseScope>
                                             <CloseScope>
<SubjectAreaName>::=                         <MetaObjectName>
<SubjectAreaVersionNumber>::=                <String>
```

[extracted from EIA/IS-109, page 12]

5.2.3.3.2.4 Example

CDIF Source Example :

```
(:SUBJECTAREAREFERENCE DataModeling
       (:VERSIONNUMBER     "01.00")
)
(:SUBJECTAREAREFERENCE DataDefinition
       (:VERSIONNUMBER     "01.00")
)
```

[extracted from EIA/IS-110 Extract,page 27]

UOL Mapping Example :

```
-- code in the Model package
import from OwnMetaModel::DataModeling
import from OwnMetaModel::DataDefinition

-- code in the Meta-model package
-- code in DataModeling Meta-model package

class VersionNumberDataModeling
       feature {any}
            VersionNumber: String is '01.00'
       end
end

-- code in DataDefinition Meta-model package
class VersionNumberDataDefinition
       feature {any}
            VersionNumber: String is '01.00'
       end
end
```

5.2.3.3.3 Meta-model Extension Clause

5.2.3.3.3.1 Introduction

This section contains meta-model extension information that must be communicated to the importer before it encounters model data. This section must be empty if importers and exporters use only the standardized CDIF Subject Areas.

When an exporter needs to extend the standardized CDIF meta-model or to provide its own meta-model definition(s), it places these extensions in this section. All Syntaxes shall provide mechanisms for extension.

The syntax of the Meta-model Extension Clause:

```
<MetaModelExtensionClause>::=
                              <MetaMetaEntityInstance>
                            | <MetaMetaRelationshipInstance>
                            | <EnumeratedMetaAttributeExtension>
```

[extracted from EIA/IS-108,page 14 and EIA/IS-109,page 13]

5.2.3.3.3.2 Meta-meta-entity Instance

5.2.3.3.3.2.1 Introduction

A meta-meta-entity is the definition of the behaviour and structure of meta-entities, meta-relationships, meta-attributes, or subject areas (i.e., a definition of the meta-object definitions used to describe information in models).

[extracted from EIA/IS-109,page 50]

5.2.3.3.3.2.2 UOL Mapping

The extensions are mapped into a package containing all the extensions. This package is included in our own meta-model package. These extensions are mapped as a class stereotyped with Utility.

The use of a Utility stereotype allows expressing all the meaning of the Meta-meta-entity in a simple way resulting in a very straightforward mapping.

5.2.3.3.3.2.3 Grammar

```
<MetaMetaEntityInstance>  ::=      <OpenScope> <MetaMetaEntityName>
                                   <CDIFMetaIdentifier>
                                   [<MetaMetaAttributeInstance>]...
                                   <CloseScope>
<MetaMetaEntityName>         ::=   <MetaMetaObjectName>
<CDIFMetaIdentifier>         ::=   <Identifier>
```

[extracted from EIA/IS-109, page 13]

5.2.3.3.3.2.4 Example

CDIF Source Example:

```
(MetaEntity ME001    [MetaMetaAttributeInstance]...)
```

[extracted from EIA/IS-110 Extract,pages 38-39]

UOL Mapping Example:

```
class ME001
       stereotyped with Utility
       feature {any}
            -- Translation of MetaMetaAttributeInstance
       end
end
```

5.2.3.3.3.3 Meta-meta-attribute Instance

5.2.3.3.3.3.1 Introduction

A meta-meta-attribute instance clause is used to represent each of the meta-meta-attributes (other than the CDIF MetaIdentifier meta-meta-attribute) of the meta-meta-entity.

[extracted from EIA/IS-109,page 14]

5.2.3.3.3.3.2 UOL Mapping

Meta-meta-attribute instances are mapped as features of the UOL class they belong to.

The mapping of a meta-meta-attribute instance as a feature is a consequence of having translated its meta-meta-entity as a class stereotype. This is because a feature is the mean we have of adding characteristics to a class.

5.2.3.3.3.3.3 Grammar

```
<MetaMetaAttributeInstance>        ::=      <OpenScope>
                                   <MetaMetaAttributeName>
                                   <MetaMetaAttributeValue>
                                   <CloseScope>
<MetaMetaAttributeName>   ::=      <MetaMetaObjectName>
<MetaMetaAttributeValue>  ::=      <MetaAttributeValue>
```

[extracted from EIA/IS-109, page 14]

5.2.3.3.3.3.4 Example

CDIF Source Example:

```
(MetaEntity ME001
(Name *SecurityClassification*)
            ...
)
```

[extracted from EIA/IS-110 Extract,page 38]

UOL Mapping Example:

```
class ME001
       stereotyped with Utility
       feature {any}
            Name: String is '*SecurityClassification*'
       ...
       end
end
```

5.2.3.3.3.4 Meta-meta-relationship Instance

5.2.3.3.3.4.1 Introduction

A meta-meta-relationship is a definition of a type of data object that occurs in CDIF meta-models. Specifically, a meta-meta-relationship represents the definition of a relationship between instances of meta-meta-entities.

[extracted from EIA/IS-109,page 50]

5.2.3.3.3.4.2 UOL Mapping

Meta-meta-relationship instances are mapped as UOL classes stereotyped with UMLAssociation, in the meta-model extension package.

The relationships are mapped as classes instead of relations because, at this level, a relationship does not connect classes. It is at the model level when a relationship takes its role as connector between classes. In the meta-model, we only define the concept of connector, but we do not connect classes actually.

5.2.3.3.3.4.3 Grammar

```
<MetaMetaRelationshipInstance> ::=
                <OpenScope>
                <FullMetaMetaRelationshipName>
                <SourceMetaMetaEntityCDIFMetaIdentifier>
                <DestinationMetaMetaEntityCDIFMetaIdentifier>
                <CloseScope>
<FullMetaMetaRelationshipName> ::=
                <SourceMetaMetaEntityName> <Dot>
                <MetaMetaRelationshipName> <Dot>
                <DestinationMetaMetaEntityName>
<SourceMetaMetaEntityName>          ::=
<MetaMetaObjectName>
<MetaMetaRelationShipName>          ::=
<MetaMetaObjectName>
<DestinationMetaMetaEntityName>::=              <MetaMetaObjectName>
<SourceMetaMetaEntityCDIFMetaIdentifier>:=      <CDIFMetaIdentifier>
<DestinationMetaMetaEntityCDIFMetaIdentifier>::= <CDIFMetaIdentifier>
<CDIFMetaIdentifier>::=                         <Identifier>
```

[extracted from EIA/IS-109, page 15]

5.2.3.3.3.4.4 Example

CDIF Source Example:

```
(MetaRelationship.HasDestination.MetaEntity MR001 ME001)
```

[extracted from EIA/IS-110 Extract, page 39]

UOL Mapping Example:

```
class HasDestination
       stereotyped with UMLAssociation
       feature {any}
            from: String is MR001
            to: String is ME001
       end
end
```

5.2.3.3.3.5 Enumerated Meta-attribute Extension

5.2.3.3.3.5.1 Introduction

This construction extends an enumerated meta-attribute, adding values that are appended to those values that have already been defined for the meta-attribute.

5.2.3.3.3.5.2 UOL mapping

Enumerated Meta-attributes are mapped as "unique <List-of-values>". On the list of values there are all the values of the Meta-attribute. The UOL mapping only appears when an attribute of a (Meta-)entity or relationship is defined with it.

To extend an enumerated value, we redefine the enumerated value adding the new values with the UOL equivalent construction for enumerated values, unique <List-of-values>.

5.2.3.3.3.5.3 Grammar

```
<EnumeratedMetaAttributeExtension>::=
       <OpenScope>
       <ExtendMetaAttributeKeyword>
       <CDIFMetaIdentifier>
       <OpenScope>
       <EnumeratedIdentifierValue>
       [<EnumeratedSeparator><EnumeratedIdentifierValue>]...
       <CloseScope>
       <CloseScope>
<CDIFMetaIdentifier>::=            <Identifier>
<EnumeratedIdentifierValue>::=     <Identifier>
```

[extracted from EIA/IS-109, page 16]

5.2.3.3.3.5.4 Example

CDIF Source Example:

```
(:EXTENDMETA-ATTRIBUTE MA001 (Encrypted,Nonencrypted))
```

[extracted from EIA/IS-109, page 16]

UOL mapping Example: (appears when MA001 is used as a Type)

```
MA001: unique {<Before defined values (if necessary)>,
```

```
       Encrypted,Nonencrypted};
<AttributeName> : MA001
```

5.2.3.4 Model Section

5.2.3.4.1 Introduction

The Model Section contains references to attributable meta-objects (object types) and actual model data. The object types referenced here are instances of MetaEntities and MetaRelationships that were defined in the Meta-model Section as part of the CDIF Subject Area references or in the Meta-model Extensions clauses. The model data are in the form of meta-entity instances, meta-relationships instances and meta-attribute instances.

[extracted from EIA/IS-109,page 16]

Its Grammar is as follows:

```
<ModelSectionClause>::=   <OpenScope>
                          <ModelKeyword> <ObjectClause>...
                          <CloseScope>
<ObjectClause>::=                  <MetaEntityInstance>
                          | <MetaRelationshipInstance>
```

[extracted from EIA/IS-109, page 17]

An example of the Model Section Clause is

```
(:MODEL <ObjectClause> ...)
```

[extracted from EIA/IS-109, page 17]

5.2.3.4.2 Meta-entity Instance

5.2.3.4.2.1 Introduction

Meta-entity is a definition of a type of data object that occurs in CDIF models. Specifically, a meta-entity represents a set of zero or more meta-attributes, stored together to represent a thing, event or concept that has instances in a model.

[extracted from EIA/IS-109,page 49]

5.2.3.4.2.2 UOL mapping

Meta-entity instances are mapped as classes stereotyped with the Meta-meta-entity they are instances of.

The decision of mapping a meta-entity instance as a stereotype of the meta-meta-entity it is related to allows keeping all the characteristics of the meta-entity while adding necessary details for a model entity.

5.2.3.4.2.3 Grammar

```
<MetaEntityInstance> ::=  <OpenScope> <MetaEntityName>
                          <CDIFIdentifier>
                          [<MetaAttributeInstance>]...
                          <CloseScope>
<MetaEntityName> ::=      <MetaObjectName>
<CDIFIdentifier> ::=      <Identifier>
```

[extracted from EIA/IS-109, page 17]

5.2.3.4.2.4 Example

CDIF Source Example:

```
(DataModel MOD01
       [MetaAttributeInstance]...
)
```

[extracted from EIA/IS-110 Extract,page 27]

UOL Mapping Example:

```
class MOD01
       stereotyped with DataModel
       feature {any}
       -- Translation of MetaAttributeInstance
       end
end
```

5.2.3.4.3 Meta-relationship Instance

5.2.3.4.3.1 Introduction

A meta-relationship is a definition of a type of data object that occurs in CDIF models. Specifically, a meta-relationship represents the definition of a relationship between meta-entities that has instances in a model. A meta-relationship may also define a set of zero or more meta-attributes, stored together to represent characteristics of a relationship between meta-entities.

[extracted from EIA/IS-109, page 50]

5.2.3.4.3.2 UOL mapping

Meta-relationship instances are mapped as relations stereotyped with the Meta-meta-relationship they are instances of.

A meta-relationship instance is mapped as a relation because, at the model level, it links classes. In UOL, this is accomplished with relations stereotyped with the meta-meta-relationship it is related to.

5.2.3.4.3.3 Grammar

```
<MetaRelationshipInstance>::=
                          <OpenScope>
                          <FullMetaRelationshipName>
                          <MetaRelationshipCDIFIdentifier>
                          <SourceMetaEntityCDIFIdentifier>
                          <DestinationMetaEntityCDIFIdentifier>
                          [MetaAttributeInstance]...
                          <CloseScope>
FullMetaRelationshipName::=
       <SourceMetaEntityName><Dot><MetaRelationshipName>
       <Dot><DestinationMetaEntityName>
SourceMetaEntityName::=            <MetaObjectName>
MetaRelationshipName::=            <MetaObjectName>
DestinationMetaEntityName::=       <MetaObjectName>
MetaRelationshipCDIFIdentifier::=  <CDIFIdentifier>
SourceMetaEntityCDIFIdentifier::=  <CDIFIdentifier>
DestinationMetaEntityCDIFIdentifier ::=
                          <CDIFIdentifier>
CDIFIdentifier::=         <Identifier>
```

[extracted from EIA/IS-109, page 18]

5.2.3.4.3.4 Example

CDIF Source Example:

```
(DataModel.IsCollectionOf.DataModelObject R001 MOD01 ENT02)
```

[extracted from EIA/IS-110 Extract, page 30]

UOL Mapping Example:

```
relation MOD01_IsCollectionOf_ENT02
       stereotyped with IsCollectionOf
       link MOD01[1], ENT02[1]
end
```

5.2.3.4.4 Meta-attribute Instance

5.2.3.4.4.1 Introduction

A meta-attribute is a definition of a characteristic of a meta-entity or a meta-relationship. Instances of a meta-attribute occur in a model as data values.

[extracted from EIA/IS-109,page 49]

5.2.3.4.4.2 UOL mapping

Meta-attribute instances are mapped as features of the UOL class/relation they belong to.

The use of features is the mean UOL has to specify characteristics of a class, which is what an attribute express.

5.2.3.4.4.3 Grammar

```
<MetaAttributeInstance>::=          <OpenScope> <MetaAttributeName>
                           <MetaAttributeValue>
                           <CloseScope>
<MetaAttributeName>::=     <MetaObjectName>
```

[extracted from EIA/IS-109, page 19]

5.2.3.4.4.4 Example

CDIF Source Example:

```
(DataModel MOD01
       (Name “Example2”)
       ...
)
```

[extracted from EIA/IS-110 Extract, page 27]

UOL Mapping Example:

```
class MOD01
       stereotyped with DataModel
       feature {any}
            Name: String is ‘Example2’
            ...
       end
end
```

5.2.3.4.5 Meta-attribute Value

(UOL 1.2)

5.2.3.4.5.1 Introduction

Different values supported by CDIF.

5.2.3.4.5.2 UOL mapping

All values are mapped as “is <Value>” in the definition of the attribute in the Meta-entity instance in the ownmodel_package.

Values directly supported are mapped as they are (its type and its value), and values non-directly supported are mapped as strings.

There are some constructions with a direct equivalence in UOL for which we can do a direct mapping. The constructions with no direct equivalence can be translated using UOL standard elements.

5.2.3.4.5.3 Grammar

```
<MetaAttributeValue>::=    <BitmapValue>
                         | <BooleanValue>
                         | <DateValue>
                         | <EnumeratedValue>
                         | <FloatValue>
                         | <IdentifierValue>
                         | <IntegerValue>
                         | <IntegerListValue>
                         | <PointValue>
                         | <PointListValue>
                         | <StringValue>
                         | <TextValue>
                         | <TimeValue>
<BitmapValue>::=         <BitmapKeyword><Height><Width>
                         <OpenScope><Bitmap><CloseScope>
<Height>::=              <HeightKeyword><PositiveInteger>
<Width>::=               <WidthKeyword><PositiveInteger>
<Bitmap>::=              <PixelValue> [<ListSeparator>
                         <PixelValue>] ...
<PixelValue>::=          <OpenScope>
                         <PixelRedIntensity> <PixelSeparator>
                         <PixelGreenIntensitiy><PixelSeparator>
                         <PixelBlueIntensity>
<CloseScope>
<PixelRedIntensity>::=   <PixelIntensity>
<PixelGreenIntensity>::= <PixelIntensity>
<PixelBlueIntensity>::=  <PixelIntensity>
<BooleanValue>::=                <TrueValue> | <FalseValue>
<DateValue>::=           <DateKeyword><Date><DateClassValue>
<DateClassValue>::=      <Absolute> | <RelativePositive> |
<RelativeNegative>
<IntegerValue>::=                <DecimalIntegerValue> |
<BinaryValue>
                         | <HexaDecimalValue>
                         | <OctalValue>
<IntegerListValue>::=    <IntegerListKeyword><OpenScope>
                         <IntegerValue> [<ListSeparator>
                         <IntegerValue>]...
                         <CloseScope>
<PointValue>::=          <PontKeyword><Point>
```

```
<Point>::=                <OpenScope>
                          <XValue><PointSeparator>
                          <YValue><PointSeparator>
                          <ZValue>
                          <CloseScope>
<XValue>::=               <Integer>
<YValue>::=               <Integer>
<ZValue>::=               <Integer>
<PointListValue>::=       <PointListKeyword><OpenScope>
                          <Point> [<ListSeparator> <Point>] ...
                          <CloseScope>
<Point>::=                <OpenScope>
                          <XValue><PointSeparator>
                          <YValue><PointSeparator>
                          <ZValue>
                          <CloseScope>
<StringValue>::=          <String>
<TextValue>::=            <TextString> [<ListSeparator>
<TextString>]...
<TimeValue>::=            <TimeKeyword> <Time> <TimeClassValue>
<TimeClassValue>::=       <AbsoluteUTC> | <AbsoluteLocal>
                          | <RelativePositive>
                          | <RelativeNegative>
```

[extracted from EIA/IS-109, pages 19-25]

5.2.3.4.5.4 Example

All the CDIF examples are extracted from EIA/IS-109

BitmapValue

CDIF Source Example:

```
:BITMAP :HEIGHT 2 WIDTH 2
((120,50,35),(130,80,70),(100,28,231),(111,255,0))
```

[extracted from EIA/IS-109, page 20]

UOL Mapping Example:

```
class BitmapValue
       stereotyped with DataType
       feature {any}
            Height : integer;
            Width    : integer;
            PixelList [0..*] : Pixel
       end
       feature {none}
            Pixel[0..2] : integer
       end
end

a:BitmapValue
a.Height=2
a.Width=2
a.PixelList[0][0]=120
a.PixelList[0][1]=50
a.PixelList[0][2]=35
```

```
(...)
```

BooleanValue

CDIF

```
-TRUE-
```

[extracted from EIA/IS-109, page 21]

UOL

```
boolean is true
```

DateValue

CDIF

```
:DATE 1940/12/07 Absolute
```

[extracted from EIA/IS-109, page 21]

UOL (As a string)

```
string is '1940/12/07 Absolute'
```

EnumeratedValue

CDIF (As part of a enumeration)

```
(..,red,..)
```

[extracted from EIA/IS-109, page 21]

UOL (As a enumerated value)

```
unique {..,red,..}
```

FloatValue

CDIF

```
#f123.45E2
```

[extracted from EIA/IS-109, page 22]

UOL

```
float is 123.45E2
```

IdentifierValue

CDIF

```
*johnBrownsBody*
```

[extracted from EIA/IS-109, page 22]

UOL

```
Identifier is johnBrownsBody
```

IntegerValue

CDIF

```
#d12345
#d-12345
```

[extracted from EIA/IS-109, page 22]

UOL

```
integer is 12345
integer is -12345
```

IntegerListValue

CDIF

```
:INTEGERLIST (#d10,#d20,#d11)
```

[extracted from EIA/IS-109, page 23]

UOL

```
...
a[0..2]: integer;
...

a[0] = 10
a[1] = 20
a[2] = 11
```

PointValue

CDIF

```
:POINT (0 0 0)
```

[extracted from EIA/IS-109, page 23]

UOL

```
PointValue[0..2] : integer
p : PointValue
p[0]  = 0
p[1]  = 0
p[2]  = 0
```

PointListValue

CDIF

```
:POINTLIST ((0 0 0),(1 1 1),(3 3 3))
```

[extracted from EIA/IS-109, page 24]

UOL

```
...
a[0..*]: PointValue ;
...

a[0][0] = 0
a[0][1] = 0
a[0][2] = 0
...
```

StringValue

CDIF

```
“This is a string”
```

[extracted from EIA/IS-109, page 24]

UOL

```
string is ‘This is a string’
```

TextValue

CDIF

```
#[Program SumIntegers (Input, Output);
var
total, input_integer : Integer;
begin
while not EOF(Input) do
     begin
     ReadLn(input_integer);
     Total:= total + input_integer
     end
WriteLn(‘Total = ‘,total’);
end.]#
```

[extracted from EIA/IS-109, page 25]

UOL

```
a: string is 'Program SumIntegers (Input, Output);
     var
     total, input_integer : Integer;
     begin
     while not EOF(Input) do
          begin
          ReadLn(input_integer);
          Total:= total + input_integer
          end
```

```
      WriteLn(\'Total = \',total\');
      end.'
```

TimeValue

CDIF

```
:TIME 07:20:23 AbsoluteUTC
:TIME 00:00:00.250 RelativePositive
```

[extracted from EIA/IS-109, page 25]

UOL

```
string is '07:20:23 AbsoluteUTC'
string is '00:00:00.250 RelativePositive'
```

5.2.3.5 Comments

5.2.3.5.1 Introduction

Comments may appear anywhere in the syntax between any two terminal symbols.

5.2.3.5.2 UOL mapping

Comments are mapped in UOL as comments too, respecting the meaning that user gives them. If comment has more than one line, each line must be mapped in UOL has a comment (UOL has no multi-line comments, only permit them as a group of one-line comments).

5.2.3.5.3 Grammar

```
<Comment>
```

5.2.3.5.4 Example

CDIF Source Example:

```
#| this is
a multi-line
comment
|#
```

UOL Mapping Example:

```
-- this is
-- a multi-line
-- comment
```

5.2.4 Transfer Example and UOL Mapping

5.2.4.1 CDIF Code.

```
CDIF,SYNTAX "SYNTAX.1" "02.00.00",ENCODING "ENCODING.1" "02.00.00"
#| Sample CDIF Transfer using CDIF Integrated Meta-model |#
#| H e a d e r S e c t i o n |#
(:HEADER
(:SUMMARY
(ExporterName "CASE Genius")
(ExporterVersion "01.00.00")
(ExportDate "1991/04/01")
(ExportTime "06:00:00")
(PublisherName "Mary Lomas")
)
)
#| M e t a - m o d e l S e c t i o n |#
(:META-MODEL
 (:SUBJECTAREAREFERENCE DataModeling
(:VERSIONNUMBER "01.00")
)
(:SUBJECTAREAREFERENCE DataDefinition
(:VERSIONNUMBER "01.00")
)
)
#| M o d e l S e c t i o n |#
(:MODEL
(DataModel MOD01
(Name "Example 1")
(BriefDescription "The first example Data Model.")
(ModelType "Conceptual")
)
(Entity ENT02
(Name "Customer Account")
(BriefDescription "The bank account of a customer.")
)
(Entity ENT06
(Name "Transaction")
(BriefDescription "An action generated by an ATM session against a
customer's bank account.")
)
(Entity ENT07
(Name "Card Retention")
(BriefDescription "The action generated by retaining the Card
within the ATM.")
)
(Entity ENT08
(Name "Single Account Transaction")
(BriefDescription "A subtype transaction indicating action against
one account.")
)
(Entity ENT09
(Name "Transfer")
(BriefDescription "A subtype transaction indicating movement of
```

```
funds between multiple accounts.")
)
(LocalAttribute DMATT12
(Name "Transaction Identifier")
(BriefDescription "The unique identifier of a Transaction.")
(IsOptional -False-)
)
(LocalAttribute DMATT13
(Name "Transaction Date")
(BriefDescription "The date the transaction took place.")
(IsOptional -False-)
)
(LocalAttribute DMATT14
(Name "Transaction Time")
(BriefDescription "The time the transaction took place.")
 (IsOptional -False-)
)
(LocalAttribute DMATT15
(Name "Customer Account Identifier")
(BriefDescription "The unique identifier of a Customer Account.")
(IsOptional -False-)
)
(LocalAttribute DMATT16
(Name "Customer Account Type")
(BriefDescription "A code that indicates the type of account.")
(IsOptional -False-)
)
(LocalAttribute DMATT17
(Name "Customer Account Balance")
(BriefDescription "The amount of money in a customer's account.")
(IsOptional -False-)
)
(LocalAttribute DMATT20
(Name "Amount")
(BriefDescription "The amount of money transferred between
accounts.")
(IsOptional -False-)
)
(Relationship REL04
(Name "CarriedOut")
(BriefDescription "This relates a Customer Account to the Single
Account Transaction performed against it.")
)
(Relationship REL05
(Name "Movement")
(BriefDescription "This relates a Customer Account to a
Transfer.")
)
(OrthogonalSubtypeSet OSS01
(Name "TransactionType")
)
(Role ROLE09
(RoleName "On")
(MinOuterCardinality "1")
(MaxOuterCardinality "1")
(MinInnerCardinality "0")
```

```
(MaxInnerCardinality "N")
)
(Role ROLE10
(RoleName "Is")
(MinOuterCardinality "1")
(MaxOuterCardinality "1")
(MinInnerCardinality "1")
(MaxInnerCardinality "1")
)
(Role ROLE11
 (RoleName "From")
(MinOuterCardinality "1")
(MaxOuterCardinality "1")
(MinInnerCardinality "0")
(MaxInnerCardinality "N")
)
(Role ROLE12
(RoleName "To")
(MinOuterCardinality "1")
(MaxOuterCardinality "1")
(MinInnerCardinality "0")
(MaxInnerCardinality "N")
)
(Role ROLE13
(RoleName "Is")
(MinOuterCardinality "1")
(MaxOuterCardinality "1")
(MinInnerCardinality "1")
(MaxInnerCardinality "1")
)
(RolePlayer RP01
(Name "Actor")
)
(RolePlayer RP02
(Name "Object")
)
(RolePlayer RP03
(Name "Actor")
)
(RolePlayer RP04
(Name "Sender")
)
(RolePlayer RP05
(Name "Receiver")
)
(IntegerType ELEM01
(Name "Int")
(SignedFlag -True-)
)
(DataElementType TYPE01
(Name "Identifier")
)
(DataElementType TYPE03
(Name "AccountType")
)
(MoneyType TYPE04
```

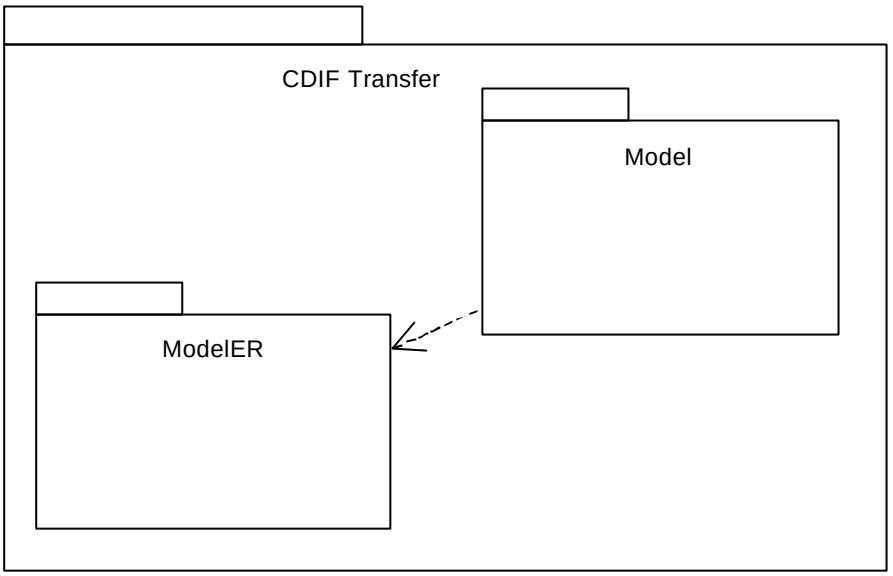
```
(Name "Amount")
)
(DateType TYPE08
(Name "Date")
)
(TimeType TYPE09
(Name "Time")
)
(DomainGroup DOM57)
(DomainValueEnumeration DOM58
(Name "Checking")
)
(DomainValueEnumeration DOM59
(Name "Savings")
)
(DataModel.IsCollectionOf.DataModelObject R001 MOD01 ENT02)
(DataModel.IsCollectionOf.DataModelObject R002 MOD01 ENT06)
(DataModel.IsCollectionOf.DataModelObject R003 MOD01 ENT07)
(DataModel.IsCollectionOf.DataModelObject R004 MOD01 ENT08)
(DataModel.IsCollectionOf.DataModelObject R005 MOD01 ENT09)
(DataModel.IsCollectionOf.DataModelObject R006 MOD01 REL04)
(DataModel.IsCollectionOf.DataModelObject R007 MOD01 REL05)
(InheritableDataModelObject.Has.OrthogonalSubtypeSet R044 ENT06 OSS01)
(OrthogonalSubtypeSet.IsContructedWith.InheritableDataModelObject R008
OSS01 ENT07)
(OrthogonalSubtypeSet.IsContructedWith.InheritableDataModelObject R009
OSS01 ENT08)
(OrthogonalSubtypeSet.IsContructedWith.InheritableDataModelObject R010
OSS01 ENT09)
(DataModelObject.Plays.Role R011 ENT02 RP02)
(DataModelObject.Plays.Role R012 ENT08 RP01)
(DataModelObject.Plays.Role R013 ENT02 RP04)
(DataModelObject.Plays.Role R014 ENT02 RP05)
(DataModelObject.Plays.Role R015 ENT09 RP03)
(RolePlayer.Plays.Role R039 RP01 ROLE10)
(RolePlayer.Plays.Role R040 RP02 ROLE09)
(RolePlayer.Plays.Role R041 RP03 ROLE13)
(RolePlayer.Plays.Role R042 RP04 ROLE11)
(RolePlayer.Plays.Role R043 RP05 ROLE12)
(Role.BelongsTo.Relationship R016 ROLE09 REL04)
(Role.BelongsTo.Relationship R017 ROLE10 REL04)
(Role.BelongsTo.Relationship R018 ROLE11 REL05)
(Role.BelongsTo.Relationship R019 ROLE12 REL05)
(Role.BelongsTo.Relationship R020 ROLE13 REL05)
(DataType.HasSubtype.DataType R021 ELEM01 TYPE01)
(DataType.TakesValuesFrom.DomainGroup R022 TYPE03 DOM57)
(DomainGroup.Contains.Domain R023 DOM57 DOM58)
(DomainGroup.Contains.Domain R024 DOM57 DOM59)
(DataObject.IsDescribedBy.Attribute R025 ENT06 DMATT12
(SequenceNumber #d1)
)
(DataObject.IsDescribedBy.Attribute R026 ENT06 DMATT13
(SequenceNumber #d2)
)
(DataObject.IsDescribedBy.Attribute R027 ENT06 DMATT14
(SequenceNumber #d3)
```

(UOL 1.2)

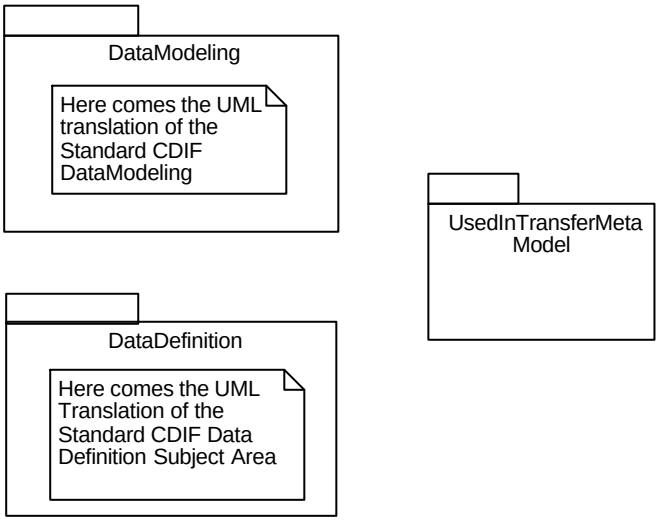
```
)
(DataObject.IsDescribedBy.Attribute R028 ENT02 DMATT15
(SequenceNumber #d1)
)
(DataObject.IsDescribedBy.Attribute R029 ENT02 DMATT16
(SequenceNumber #d2)
)
(DataObject.IsDescribedBy.Attribute R030 ENT02 DMATT17
(SequenceNumber #d3)
)
(DataObject.IsDescribedBy.Attribute R031 ENT09 DMATT20
(SequenceNumber #d1)
)
(Attribute.IsOccurrenceOf.DataType R032 DMATT12 TYPE01)
(Attribute.IsOccurrenceOf.DataType R033 DMATT13 TYPE08)
(Attribute.IsOccurrenceOf.DataType R034 DMATT14 TYPE09)
(Attribute.IsOccurrenceOf.DataType R035 DMATT15 TYPE01)
(Attribute.IsOccurrenceOf.DataType R036 DMATT16 TYPE03)
(Attribute.IsOccurrenceOf.DataType R037 DMATT17 TYPE04)
(Attribute.IsOccurrenceOf.DataType R038 DMATT20 TYPE04)
)
```

5.2.4.2 UML Translation.

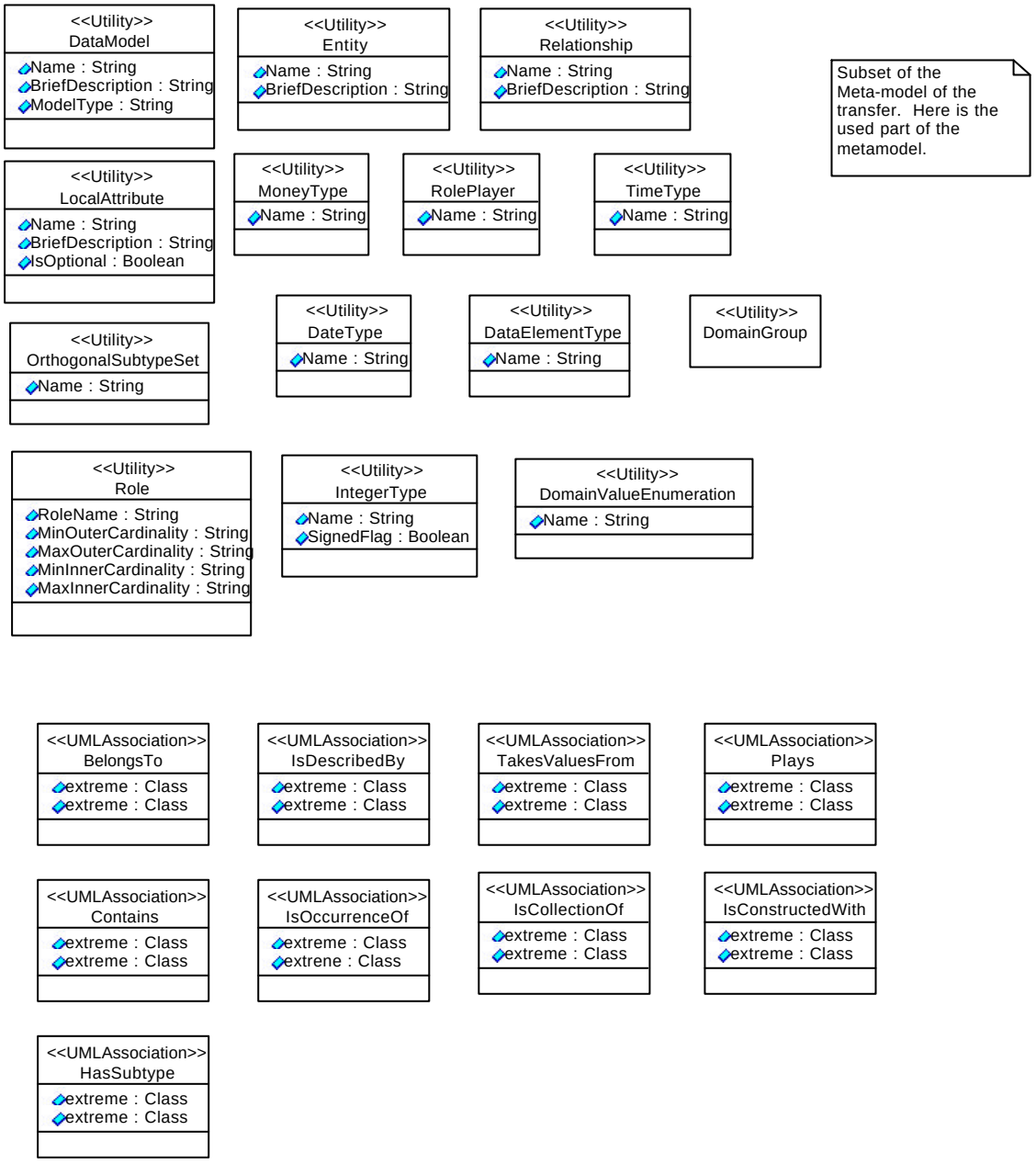
In this chapter we introduce the structure of the packages used for the translation. The definitions of the stereotypes used in the example are also included.



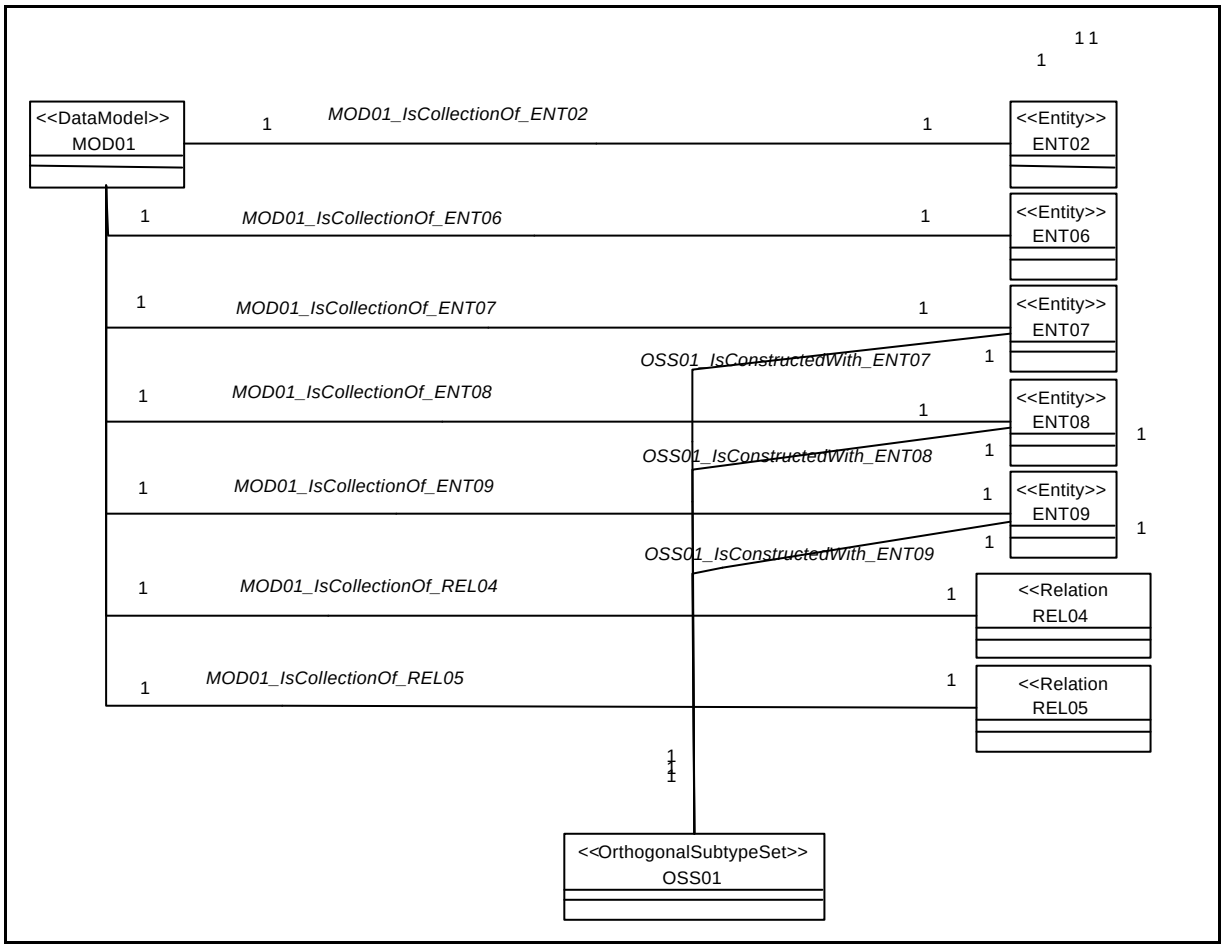
Main package

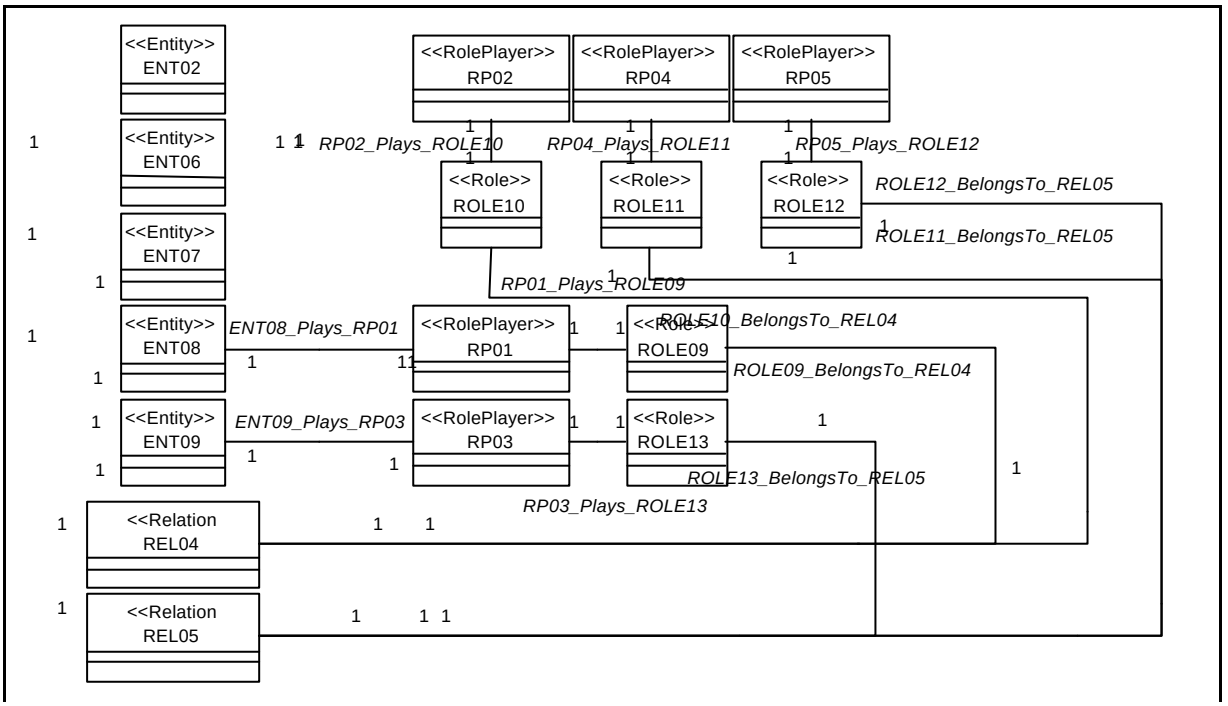


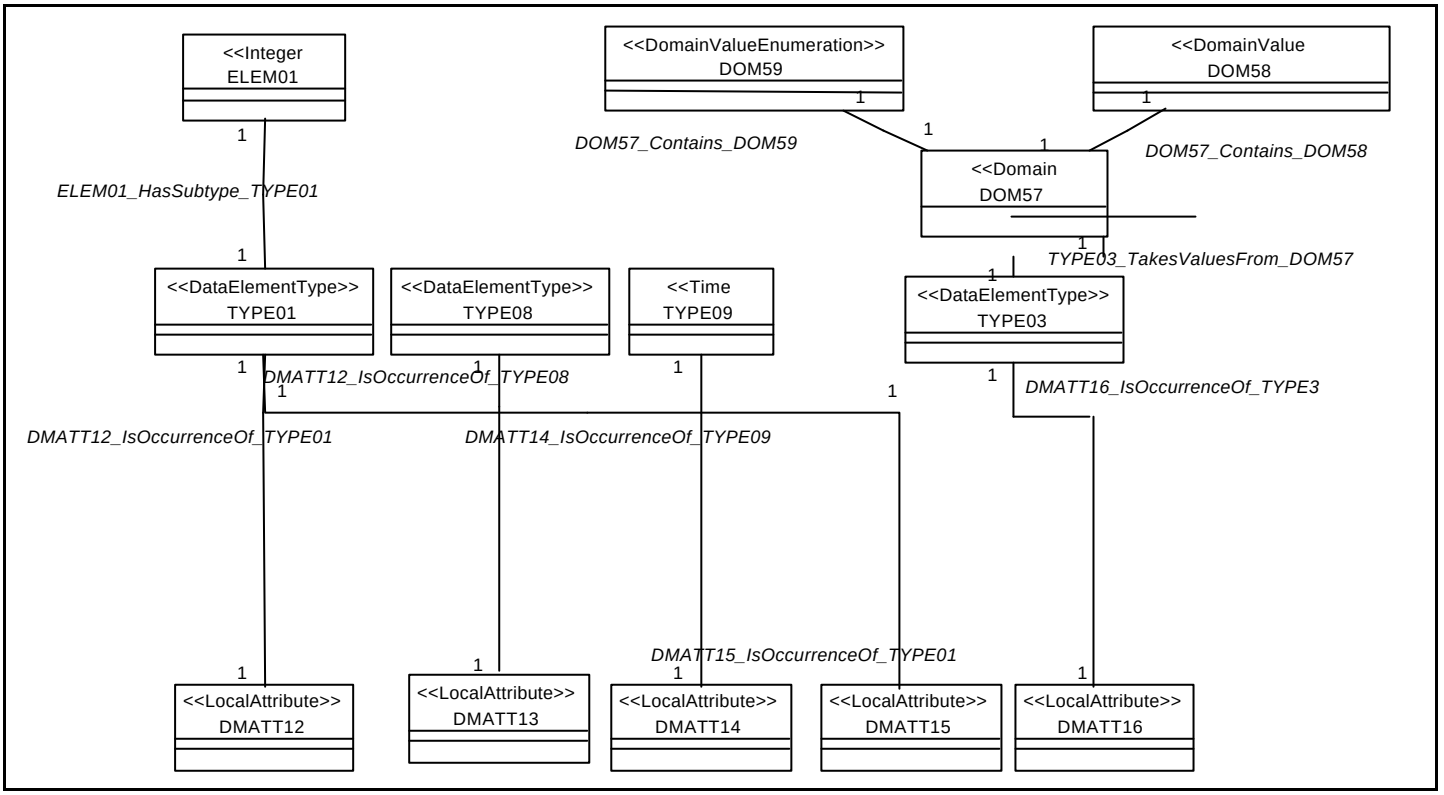
Model ER package

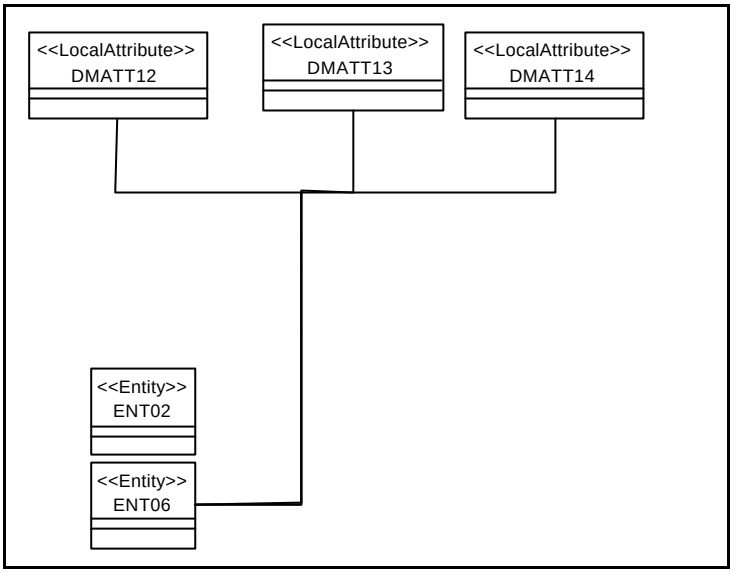


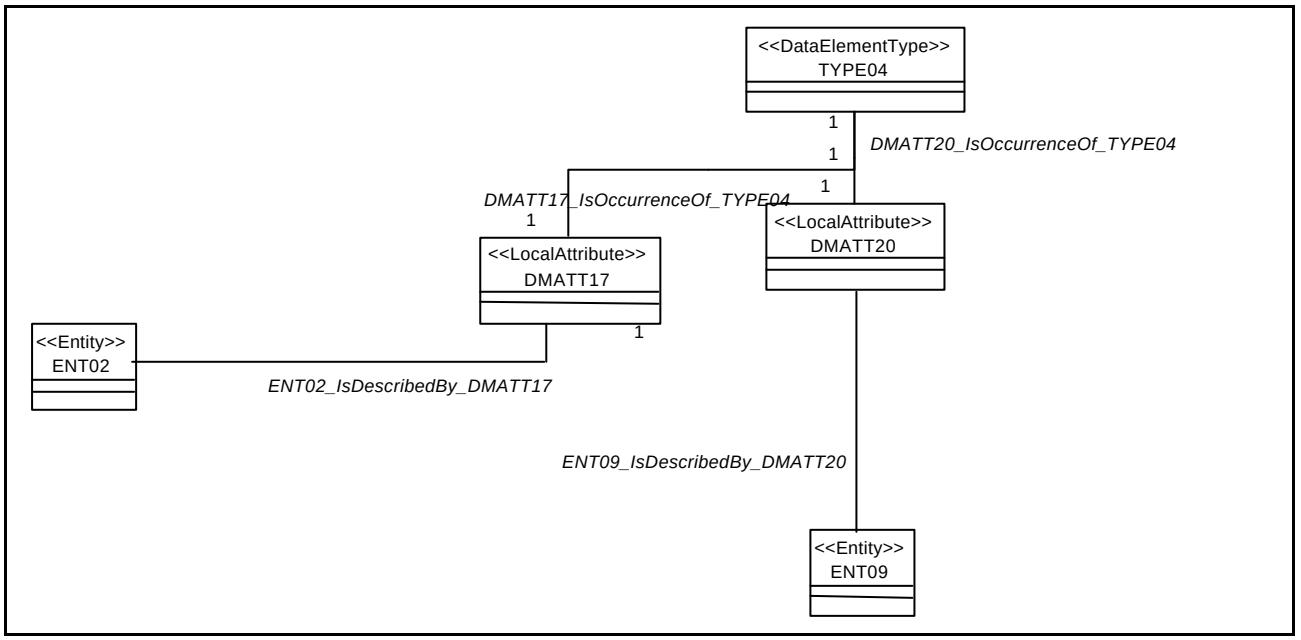
UsedInTransferMetamodel


<<DataModel>>
MOD01
<<Entity>>
ENT02
<<Entity>>
ENT06
<<Entity>>
ENT07
<<Entity>>
ENT08
<<Entity>>
ENT09
<<Relation
REL04
<<Relation
REL05
<<OrthogonalSubtypeSet>>
OSS01
MOD01_IsCollectionOf_ENT02
MOD01_IsCollectionOf_ENT06
MOD01_IsCollectionOf_ENT07
MOD01_IsCollectionOf_ENT08
MOD01_IsCollectionOf_ENT09
MOD01_IsCollectionOf_REL04
MOD01_IsCollectionOf_REL05
OSS01_IsConstructedWith_ENT07
OSS01_IsConstructedWith_ENT08
OSS01_IsConstructedWith_ENT09


<<Entity>>
ENT02
<<Entity>>
ENT06
<<Entity>>
ENT07
<<Entity>>
ENT08
<<Entity>>
ENT09
<<Relation
REL04
<<Relation
REL05
<<RolePlayer>>
RP02
<<RolePlayer>>
RP04
<<RolePlayer>>
RP05
<<RolePlayer>>
RP01
<<RolePlayer>>
RP03
<<Role>>
ROLE10
<<Role>>
ROLE11
<<Role>>
ROLE12
<<Role>>
ROLE09
<<Role>>
ROLE13
RP02_Plays_ROLE10
RP04_Plays_ROLE11
RP05_Plays_ROLE12
ROLE12_BelongsTo_REL05
ROLE11_BelongsTo_REL05
RP01_Plays_ROLE09
ROLE10_BelongsTo_REL04
ENT08_Plays_RP01
ROLE09_BelongsTo_REL04
ENT09_Plays_RP03
ROLE13_BelongsTo_REL05
RP03_Plays_ROLE13


<<Integer
ELEM01
<<DomainValueEnumeration>>
DOM59
<<DomainValue
DOM58
DOM57_Contains_DOM59
DOM57_Contains_DOM58
<<Domain
DOM57
ELEM01_HasSubtype_TYPE01
TYPE03_TakesValuesFrom_DOM57
<<DataElementType>>
TYPE01
<<DataElementType>>
TYPE08
<<Time
TYPE09
<<DataElementType>>
TYPE03
DMATT12_IsOccurrenceOf_TYPE08
DMATT16_IsOccurrenceOf_TYPE3
DMATT12_IsOccurrenceOf_TYPE01
DMATT14_IsOccurrenceOf_TYPE09
DMATT15_IsOccurrenceOf_TYPE01
<<LocalAttribute>>
DMATT12
<<LocalAttribute>>
DMATT13
<<LocalAttribute>>
DMATT14
<<LocalAttribute>>
DMATT15
<<LocalAttribute>>
DMATT16


<<LocalAttribute>>
DMATT12
<<LocalAttribute>>
DMATT13
<<LocalAttribute>>
DMATT14
<<Entity>>
ENT02
<<Entity>>
ENT06


<<DataElementType>>
TYPE04
1
1
DMATT20_IsOccurrenceOf_TYPE04
1
DMATT17_IsOccurrenceOf_TYPE04
1
<<LocalAttribute>>
DMATT17
1
<<LocalAttribute>>
DMATT20
<<Entity>>
ENT02
ENT02_IsDescribedBy_DMATT17
ENT09_IsDescribedBy_DMATT20
<<Entity>>
ENT09

5.2.4.3 UOL Mapping.

```
package MetaModelER
is {any}
   package ModelER
   import from UOL_UML
   is {any}
        class RelationShip
             stereotyped with Utility
             feature {any}
                  Name: String;
                  BriefDescription: String
             end
        end
        class LocalAttribute
             stereotyped with Utility
             feature {any}
                  Name: String;
                  BriefDescription: String;
                  IsOptional: Boolean
             end
        end
        class DataModel
             stereotyped with Utility
             feature {any}
                  Name: String;
                  BriefDescription: String;
                  ModelType: String
             end
        end
        class MoneyType
             stereotyped with Utility
             feature {any}
                  Name: String
             end
        end
        class TimeType
             stereotyped with Utility
             feature {any}
                  Name: String
             end
        end
        class Entity
             stereotyped with Utility
             feature {any}
                  Name: String;
                  BriefDescription: String
             end
        end
        class Role
             stereotyped with Utility
             feature {any}
                  RoleName: String;
                  MinOuterCardinality: Character;
                  MaxOuterCardinality: Character;
```

(UOL 1.2)

```
                MinInnerCardinality: Character;
                MaxInnerCardinality: Character
            end
        end
        class DataElementType
            stereotyped with Utility
            feature {any}
                Name: String
            end
        end
        class DomainGroup stereotyped with Utility end
        class RolePlayer
            stereotyped with Utility
            feature {any}
                Name: String
            end
        end
        class OrthogonalSubtypeSet
            stereotyped with Utility
            feature {any}
                Name: String
            end
        end
        class IntegerType
            stereotyped with Utility
            feature {any}
                Name: String;
                SignedFlag: Boolean
            end
        end
        class DateType
            stereotyped with Utility
            feature {any}
                Name: String
            end
        end
        class DomainValueEnumeration
            stereotyped with Utility
            feature {any}
                Name: String
            end
        end
        class IsCollectionOf stereotyped with UMLAssociation end
        class IsConstructedWith stereotyped with UMLAssociation end
        class HasSubtype stereotyped with UMLAssociation end
        class IsOccurrenceOf stereotyped with UMLAssociation end
        class IsDescribedBy stereotyped with UMLAssociation end
        class Contains stereotyped with UMLAssociation end
        class TakesValuesFrom stereotyped with UMLAssociation end
        class Plays stereotyped with UMLAssociation end
        class BelongsTo stereotyped with UMLAssociation end
    end
    package Model
    import from ModelER
    is {any}
        class MOD01   stereotyped with DataModel end
```

```
        class ENT02   stereotyped with Entity end
        class ENT06   stereotyped with Entity end
        class ENT07   stereotyped with Entity end
        class ENT08   stereotyped with Entity end
        class ENT09   stereotyped with Entity end
        class REL04   stereotyped with RelationShip end
        class REL05   stereotyped with RelationShip end
        class OSS01   stereotyped with OrthogonalSubtypeSet end
        class ELEM01  stereotyped with IntegerType end
        class TYPE01  stereotyped with DataElementType end
        class TYPE08  stereotyped with DataElementType end
        class TYPE09  stereotyped with TimeType end
        class DOM59   stereotyped with DomainValueEnumeration end
        class DOM58   stereotyped with DomainValueEnumeration end
        class DOM57   stereotyped with DomainGroup end
        class TYPE03  stereotyped with DataElementType end
        class TYPE04  stereotyped with DataElementType end
        class DMATT12 stereotyped with LocalAttribute end
        class DMATT13 stereotyped with LocalAttribute end
        class DMATT14 stereotyped with LocalAttribute end
        class DMATT15 stereotyped with LocalAttribute end
        class DMATT16 stereotyped with LocalAttribute end
        class DMATT17 stereotyped with LocalAttribute end
        class DMATT20 stereotyped with LocalAttribute end
        class RP01    stereotyped with RolePlayer end
        class RP02    stereotyped with RolePlayer end
        class RP03    stereotyped with RolePlayer end
        class RP04    stereotyped with RolePlayer end
        class RP05    stereotyped with RolePlayer end
        class ROLE09  stereotyped with Role end
        class ROLE10  stereotyped with Role end
        class ROLE11  stereotyped with Role end
        class ROLE12  stereotyped with Role end
        class ROLE13  stereotyped with Role end
        relation MOD01_IsCollectionOf_ENT02
                  stereotyped with IsCollectionOf
                  link MOD01[1], ENT02[1]
        end
        relation MOD01_IsCollectionOf_ENT06
                  stereotyped with IsCollectionOf
                  link MOD01[1], ENT06[1]
        end
        relation MOD01_IsCollectionOf_ENT07
                  stereotyped with IsCollectionOf
                  link MOD01[1], ENT07[1]
        end
        relation MOD01_IsCollectionOf_ENT08
                  stereotyped with IsCollectionOf
                  link MOD01[1], ENT08[1]
        end
        relation MOD01_IsCollectionOf_ENT09
                  stereotyped with IsCollectionOf
                  link MOD01[1], ENT09[1]
        end
        relation MOD01_IsCollectionOf_REL04
             stereotyped with IsCollectionOf
```

(UOL 1.2)

```
            link MOD01[1], REL04[1]
        end
        relation MOD01_IsCollectionOf_REL05
            stereotyped with IsCollectionOf
            link MOD01[1], REL05[1]
        end
        relation OSS01_IsConstructedWith_ENT07
            stereotyped with IsConstructedWith
            link OSS01[1], ENT07[1]
        end
        relation OSS01_IsConstructedWith_ENT08
            stereotyped with IsConstructedWith
            link OSS01[1], ENT08[1]
        end
        relation OSS01_IsConstructedWith_ENT09
            stereotyped with IsConstructedWith
            link OSS01[1], ENT09[1]
        end
        relation ELEM01_HasSubtype_TYPE01
            stereotyped with HasSubtype
            link ELEM01[1], TYPE01[1]
        end
        relation DOM57_Contains_DOM59
            stereotyped with Contains
            link DOM57[1], DOM59[1]
        end
        relation DOM57_Contains_DOM58
            stereotyped with Contains
            link DOM57[1], DOM58[1]
        end
        relation TYPE03_TakesValuesFrom_DOM57
            stereotyped with TakesValuesFrom
            link TYPE03[1], DOM57[1]
        end
        relation DMATT12_IsOccurrenceOf_TYPE01
            stereotyped with IsOccurrenceOf
            link DMATT12[1], TYPE01[1]
        end
        relation DMATT13_IsOccurrenceOf_TYPE08
            stereotyped with IsOccurrenceOf
            link DMATT13[1], TYPE08[1]
        end
        relation DMATT14_IsOccurrenceOf_TYPE09
            stereotyped with IsOccurrenceOf
            link DMATT14[1], TYPE09[1]
        end
        relation DMATT15_IsOccurrenceOf_TYPE01
            stereotyped with IsOccurrenceOf
            link DMATT15[1], TYPE01[1]
        end
        relation DMATT16_IsOccurrenceOf_TYPE03
            stereotyped with IsOccurrenceOf
            link DMATT16[1], TYPE03[1]
        end
        relation DMATT17_IsOccurrenceOf_TYPE04
            stereotyped with IsOccurrenceOf
```

```
        link DMATT17[1], TYPE04[1]
    end
    relation DMATT20_IsOccurrenceOf_TYPE04
        stereotyped with IsOccurrenceOf
        link DAMTT20[1], TYPE04[1]
    end
    relation ENT02_IsDescribedBy_DMATT15
        stereotyped with IsDescribedBy
        link ENT02[1], DMATT15[1]
    end
    relation ENT02_IsDescribedBy_DMATT16
        stereotyped with IsDescribedBy
        link ENT02[1], DMATT16[1]
    end
    relation ENT02_IsDescribedBy_DMATT17
        stereotyped with IsDescribedBy
        link ENT02[1], DMATT17[1]
    end
    relation ENT06_IsDescribedBy_DMATT12
        stereotyped with IsDescribedBy
        link ENT06[1], DMATT12[1]
    end
    relation ENT06_IsDescribedBy_DMATT13
        stereotyped with IsDescribedBy
        link ENT06[1], DMATT13[1]
    end
    relation ENT06_IsDescribedBy_DMATT14
        stereotyped with IsDescribedBy
        link ENT06[1], DMATT14[1]
    end
    relation ENT09_IsDescribedBy_DMATT20
        stereotyped with IsDescribedBy
        link ENT09[1], DMATT20[1]
    end
    relation ENT02_Plays_RP02
        stereotyped with Plays
        link ENT02[1], RP02[1]
    end
    relation ENT02_Plays_RP04
        stereotyped with Plays
        link ENT02[1], RP04[1]
    end
    relation ENT02_Plays_RP05
        stereotyped with Plays
        link ENT02[1], RP05[1]
    end
    relation ENT08_Plays_RP01
        stereotyped with Plays
        link ENT08[1], RP01[1]
    end
    relation ENT09_Plays_RP03
        stereotyped with Plays
        link ENT09[1], RP03[1]
    end
    relation RP02_Plays_ROLE10
        stereotyped with Plays
```

(UOL 1.2)

```
            link RP02[1], ROLE10[1]
        end
        relation RP04_Plays_ROLE11
            stereotyped with Plays
            link RP04[1], ROLE11[1]
        end
        relation RP05_Plays_ROLE12
            stereotyped with Plays
            link RP05[1], ROLE12[1]
        end
        relation RP01_Plays_ROLE09
            stereotyped with Plays
            link RP01[1], ROLE09[1]
        end
        relation RP03_Plays_ROLE13
            stereotyped with Plays
            link RP03[1], ROLE13[1]
        end
        relation ROLE10_BelongsTo_REL04
            stereotyped with BelongsTo
            link ROLE10[1], REL04[1]
        end
        relation ROLE11_BelongsTo_REL05
            stereotyped with BelongsTo
            link ROLE11[1], REL05[1]
        end
        relation ROLE12_BelongsTo_REL05
            stereotyped with BelongsTo
            link ROLE12[1], REL05[1]
        end
        relation ROLE09_BelongsTo_REL04
            stereotyped with BelongsTo
            link ROLE09[1], REL04[1]
        end
        relation ROLE13_BelongsTo_REL04
            stereotyped with BelongsTo
            link ROLE13[1], REL04[1]
        end
    end
end
```

5.3 The mapping between UOL and STEP/EXPRESS

It outlines a mapping between the proposed human readable Unified Object Language (UOL) and the international standard in data modeling STEP/EXPRESS.

The structure of this chapter is oriented on the normative STEP/EXPRESS document (ISO 10303-11:1994(E)).

The mapping described here is a way to show a STEP/EXPRESS file with UOL, it's not a mapping between STEP/EXPRESS and UML. Even if it's possible to extend it in that way.
Hence no UML specific constructs are used within the resulting UOL schema.

Every UOL statement within this document has been parsed and checked with the latest version of the UOL grammar available.

5.3.1 Data types

5.3.1.1 Simple data types

UOL doesn't make any assumptions constraining valid data types. All data types used within an EXPRESS schema are valid within UOL as well.

The basic EXPRESS data types are:

- NUMBER
  EXPRESS syntax:
  ```
  248 number_type = NUMBER
  ```
- REAL; optional enriched by the tag value "precision" (holding the length of the value as numeric expression)
  EXPRESS syntax:
  ```
  264 real_type = REAL [ '(' precision_spec ')' ].
  255 precision_spec = numeric_expression.
  ```
- INTEGER
  EXPRESS syntax:
  ```
  277 integer_type = INTEGER.
  ```
- LOGICAL
  EXPRESS syntax:
  ```
  243 logical_type =LOGICAL.
  ```
- BOOLEAN
  EXPRESS syntax:
  ```
  173 boolean_type = BOOLEAN.
  ```
- STRING; optional enriched by the tag values "width" (holding the length of the string as integer value) and "fixed" (true if the specified length is immutable)
  EXPRESS syntax:
  ```
  293 string_type = STRING [width_spec].
  318 width_spec = '(' width ')' [FIXED].
  317 width = numeric_expression.
  ```
- BINARY; optional enriched by the tag values "width" (holding the length of the length as integer value) and "fixed" (true if the specified length is immutable).
  EXPRESS syntax:
  ```
  172 binary_type = BINARY [width_spec].
  318 width_spec = '(' width ')' [FIXED].
  317 width = numeric_expression.
  ```

Example:

EXPRESS

```
ENTITY Test1;
   myNumber = NUMBER;
   myReal1 = REAL;
   myReal2 = REAL (3);
   myInteger = INTEGER;
   myLogical = LOGICAL;
   myBoolean = BOOLEAN;
   myString1 = STRING;
   myString2 = STRING (10);
   myString3 = STRING (5) FIXED;
   myBinary1 = BINARY;
   myBinary2 = BINARY (2);
   myBinary3 = BINARY (2) FIXED;
END_ENTITY;
```

UOL

```
class Test1
   feature {any}
      myNumber : NUMBER;
      myReal1 : REAL;
      myReal2 : REAL with tag values (<precision,3>);
      myInteger : INTEGER;
      myLogical : LOGICAL;
      myBoolean : BOOLEAN;
      myString1 : STRING;
      myString2 : STRING with tag values (<width,10>);
      myString3 : STRING with tag values (<width,5>,<FIXED>);
      myBinary1 : BINARY;
      myBinary2 : BINARY with tag values (<width,2>);
      myBinary3 : BINARY with tag values (<width,2>,<FIXED>)
   end
end
```

5.3.1.2 Aggregation data types

EXPRESS semantics:

Aggregation data types have as their domain collections of values of a given base data type. These base data types values are called elements of the aggregation collection. EXPRESS provides for the definition of four kinds of aggregation data types: ARRAY, LIST, BAG and SET. Each kind of aggregation data type attaches different properties to its values.

- An ARRAY is a fixed-size ordered collection. It is indexed by a sequence of integers
  EXPRESS syntax:
  ```
  165 array_type = ARRAY bound_spec OF [OPTIONAL] [UNIQUE]
  base_type.
  176 bound_spec = '[' bound_1 ':' bound_2 ']'.
  174 bound_1 = numeric_expression.
  175 bound_2 = numeric_expression.
  171 base_type = aggregation_types | simple_types | named_types.
  ```
- A LIST is a sequence of elements which can be accessed according to their position. The number of elements in a list may vary, and can be constrained by the definition of the data type.
  EXPRESS syntax:
  ```
  237 list_type = LIST [bound_spec] OF [UNIQUE] base_type.
  176 bound_spec = '[' bound_1 ':' bound_2 ']'.
  174 bound_1 = numeric_expression.
  175 bound_2 = numeric_expression.
  171 base_type = aggregation_types | simple_types | named_types.
  ```
- A BAG is an unordered collection in which duplication is allowed. The number of elements in a bag may vary, and can be constrained by the definition of the data type.
  EXPRESS syntax:
  ```
  170 bag_type = BAG [bound_spec] OF base_type.
  176 bound_spec = '[' bound_1 ':' bound_2 ']'.
  174 bound_1 = numeric_expression.
  175 bound_2 = numeric_expression.
  171 base_type = aggregation_types | simple_types | named_types.
  ```
- A SET is an unordered collection of elements in which no two elements are instance equal. The number of elements in a set may vary, and can be constrained by the definition of a data type.[5]
  EXPRESS syntax:
  ```
  285 set_type = SET [bound_spec] OF base_type.
  176 bound_spec = '[' bound_1 ':' bound_2 ']'.
  174 bound_1 = numeric_expression.
  175 bound_2 = numeric_expression.
  171 base_type = aggregation_types | simple_types | named_types.
  ```

[5] Cf. [ISO EXPRESS RM 94] 22

Rules and restrictions:

- The bound_1 expression shall evaluate to an integer value greater or equal to zero.
- The bound_2 expression shall evaluate to an integer value greater or equal to bound_1, or an indeterminate (?) value.
- If the bound_spec is omitted the limits are [0:?][6]

EXPRESS aggregations are one dimensional. An aggregation data type can represent objects usually considered to have multiple dimensions (such as mathematical matrices) whose base type is another aggregation data type. Aggregation data types can be thus nested to an arbitrary depth, allowing any number of dimensions to be represented.

Example:

```
A = LIST [1:3] OF ARRAY [5:10] OF INTEGER
```

Generic mapping:
Since UOL allows only one dimensional objects the transformation processor splits the one n dimensional defined object into n one dimensional defined ones. By doing so the processor has to create internal names for the build objects.

A special named UOL identifier supports this (in the example: ?001).

For the example above:

```
A : [1..3] ?001
   with tag values (<list>)
?001 : [5..10] INTEGER
   with tag values (<array>)
```

[6] [ISO EXPRESS RM 94] 26

Example:

EXPRESS

```
ENTITY Test_Entity;
   myArray1 = ARRAY [1:5] OF INTEGER;
   myArray2 = ARRRAY [1:?] OF BOOLEAN;
   myArray3 = ARRAY [1:5] OF OPTIONAL REAL (2);
   myArray4 = ARRAY [1:?] OF UNIQUE INTEGER;
   myList1 = LIST OF INTEGER;
   myList2 = LIST OF STRING (10) FIXED;
   myList3 = LIST [0:?] OF REAL;
   myBag1 = BAG OF NUMBER;
   myBag2 = BAG [1:?] OF NUMBER;
   mySet1 = SET OF NUMBER;
   mySet2 = SET [1:?] OF INTEGER;
   myMultiArray = ARRAY [1:10] OF
      ARRAY [11:14] OF UNIQUE something;
END_ENTITY;
```

UOL

```
class Test_Entity
   feature {any}
   myArray [1..5] : INTEGER with tag values (<array>);
   myArray2 [1..*] : BOOLEAN with tag values (<array>);
   myArray3 [0,1..5] : REAL
      with tag values (<array>,<precision,2>);
   myArray4 [1..*] : INTEGER
      with tag values (<array>,<UNIQUE>);
   myList1 [0..*] : INTEGER
      with tag values (<list>);
   myList2 [0..*] : STRING
      with tag values (<list>,<width,10>,<FIXED>);
   myList3 [0..*] : REAL;
   myBag1 [0..*] : NUMBER
      with tag values (<bag>);
   myBag2 [1..*] : NUMBER
      with tag values (<bag>);
   mySet1 [0..*] : NUMBER
      with tag values (<set>);
   mySet2 [1..*] : INTEGER
      with tag values (<set>);
   myMultiArray [1..10] : ?001
      with tag values (<array>);
   ?001 [11..14] : something
      with tag values (<array>,<UNIQUE>)
   end
end

relation ?002
   stereotyped with express_association
   link TestEntity, something [11..14]
   feature {TestEntity}
      with tag values (<IsNavigable,'FALSE'>)
   end
   feature {something}
      with tag values (<IsNavigable,'TRUE'>)
   end
end
```

Remarks:
The concrete semantics of a multi valued attribute is determined by a tag value (array, set, bag or list). Additional constraints on the collection are also expressed using tag values (e.g. UNIQUE) Constraints on the elements of the collection arising from the basic type (e.g. the width specification for a STRING) are also expressed as tag values of the new collection type.
The myMulitArray shows the splitting of a two dimensional attribute into two one dimensional ones by creating a new attribute.

5.3.1.3 Named data types

EXPRESS semantics:
The named data types are the data types that may be declared in a formal specification. There are two kinds of named data types: entity data types and defined data types.[7]

5.3.1.3.1 Entity data type

EXPRESS semantics:
Entity data types are established by ENTITY declarations (see 2.2).

Example:

EXPRESS

```
ENTITY point;
   x, y, z : REAL;
END_ENTITY;

ENTITY line;
   p0, p1 : point;
END_ENTITY;
```

UOL

```
class point
   feature {any}
      x : REAL;
      y : REAL;
      z : REAL
   end
end

class line
   feature {any}
      p0 : point;
      p1 : point
   end
end

relation ?001
   stereotyped with express_association
   link line, point[1..1]
   feature {line}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {point}
      with tag values (<IsNavigable,TRUE>,
         <AssociationEndName,p0>)
   end
end

relation ?002
   stereotyped with express_association
   link line, point[1..1]
      feature {line}
      with tag values (<IsNavigable,FALSE>)
   end
```

[7] Cf. [ISO EXPRESS RM 94] 28

(UOL 1.2)

```
          feature {point}
             with tag values (<IsNavigable, TRUE>,
                <AssociationEndName,p1>)
          end
       end
```

5.3.1.3.2 Defined data type

EXPRESS semantics:
Defined data types are declared by TYPE declarations (see 2.1).

Mapping:
The TYPE maps into a class stereotyped with *user_declared_type*. The underlying type is expressed using a tag value *underlying_type*.

Example:

EXPRESS

```
TYPE volume = REAL;
END_TYPE;

ENTITY PART;
   bulk : volume;
END_ENTITY;
```

UOL

```
class volume
   stereotyped with user_declared_type
   with tag values (<underlying_type,REAL>)
end

class PART
   feature {any}
      bulk : volume
   end
end

relation ?001
   stereotyped with express_association
   link PART, volume[1..1]
   feature {volume}
      with tag values (<IsNavigable,FALSE>,
         <AssociationEndName,bulk>)
   end
   feature {part}
      with tag values (<IsNavigable,TRUE>)
   end
end
```

5.3.1.4 Constructed data types

EXPRESS semantics:
There are two kinds of constructed data types in EXPRESS: ENUMERATION data types and SELECT data types.

5.3.1.4.1 Enumeration data types

EXPRESS semantics:
An ENUMERATION data type has as its domain an ordered set of names. The names represent values of the enumeration data type. These names are designated by enumeration_ids and are referred to as enumeration items.[8]

EXPRESS syntax:
```
201 enumeration_type = ENUMERATION OF ‘(‘ enumeration_id { ‘;’
enumeration_id} ‘)’.
```

Mapping:
The enumeration type is transferred into a class stereotyped with *express_enumeration*. Additionally the class gets a tag *express_enumeration* with a string value consisting of all the enumeration items.

When using a enumerated type the enumeration is expanded within the resulting UOL schema. Additionally the name of the enumeration is placed as tag value.

A dependency association is established between the resulting classes, directed from the enumeration using to the declaring class. This dependency is also stereotyped with *express_enumeration*.

Example:

EXPRESS
TYPE car_can_move = ENUMERATION OF (left, right, backward, forward); END_TYPE; ENTITY Use_enum; car_move : car_can_move; END_ENTITY;

[8] Cf. [ISO EXPRESS RM 94] 29

UOL

```
class car_can_move
   stereotyped with express_enumeration
   with tag value
      (<express_enumeration,'left,right,backward,forward'>)
end

class Use_enum
   stereotyped with express_entity
   feature {any}
      car_move : unique {left, right, backward, forward}
         with tag values (<express_enumeration, car_can_move>)
   end
end

relation ?001
   stereotyped with express_enumeration
   link Use_enum to car_can_move
end
```

5.3.2 Declaration

5.3.2.1 Type declaration

EXPRESS semantics:
A type declaration creates a defined data type (see 1.3.2) and declares an identifier to refer to it. Specially, the type_id is declared as the name of a defined type. The representation of this data type is the underlying_type.[9]

EXPRESS syntax:

```
304 type_decl = TYPE type_ID ‘=’ underlying_type ‘;’
[where_clause] END_TYPE ‘;’.
309 underlying_type = constructed-types | aggregate_types |
simple_types | type_ref.
```

Mapping:
Every TYPE maps into a class stereotyped with “user_declared_type”. A tag value of the class specifies the underlying type.

Example:

EXPRESS
`TYPE person_name = STRING` `END_TYPE;`
UOL

`class person_name`
`  stereotyped with user_declared_type`
`  with tag values (<underlying_type,STRING>)`
`end`

[9] Cf. [ISO EXPRESS RM 94] 33

Domain rules (WHERE clause):

EXPRESS semantics:
Domain rules specify constraints that restrict the domain of the defined data type. The domain of the defined data type is the domain of its underlying representation constrained by the domain rule(s).[10]

EXPRESS syntax:
```
315 where_clause = WHERE domain_rule ';' {domain_rule ';'}.
```

Mapping:
Each where clause corresponds to a tag *domain_rule* with the name of the rule as value. Additionally a second tag labeled with the name of the domain_rule (i.e., the value of the first tag) and the complete domain rule as uninterpreted string serving as value is used.

Example:

EXPRESS

```
TYPE positive = INTEGER
WHERE
   notnegative : SELF > 0;
END_TYPE;
```

UOL

```
class positive
   stereotyped with user_declared_type
   with tag value (<underlying_type,INTEGER>)
   with tag value (<domain_rule,notnegative>,
      <notnegative,'SELF > 0'>)
end
```

[10] Cf. [ISO EXPRESS RM 94] 34

5.3.2.2 Entity declaration

EXPRESS semantics:
An ENTITY declaration creates an entity data type and declares an identifier to refer to it.
Each attribute represents a property of an entity and may be associated with a value in each entity instance. The data type of the attribute establishes the domain of possible values.[11]

EXPRESS syntax:

```
196 entity_decl = entity_head entity_body END_ENTITY ';'.
197 entity_head = ENTITY entity_id [subsuper].
194 entity_body = {explcit_attr} [derive_clause] [inverse_clause]
[unique_clause] [where_clause].
```

Mapping:
An entity is mapped into a class with the same name as the entity. The class is stereotyped with "express_entity".

Example:

EXPRESS
`ENTITY Entity1;` `END_ENTITY;`
UOL

`class ENTITY1`
`  stereotyped with express_entity`
`end`

5.3.2.2.1 Attributes

EXPRESS semantics:
The attributes of an entity data type represent an entity's essential traits, qualities or properties. An attribute declaration establishes a relationship between the entity data type and the data type referenced by the attribute. The name of an attribute represents the role played by its associated value in the context of the entity in which it appears.
There are three kinds of attribute:

- Explicit: An attribute whose value shall be supplied by an implementation in order to create an entity instance.
- Derived: An attribute whose value is computed in some manner.
- Inverse: An attribute whose value consists of the entity instance, which uses the entity in a particular role.[12]

Since EXPRESS doesn't support any kind of visibility constraints all attributes of an entity are public, visible when mapping EXPRESS to UOL.
On the other hand, when mapping an existing UOL schema into STEP/EXPRESS all visibility constraints are ignored.

[11] Cf. [ISO EXPRESS RM 94] 35
[12] Cf. [ISO EXPESS RM 94] 25-36

5.3.2.2.1.1 Explicit attribute

EXPRESS semantics:
An explicit attribute represents a property whose value shall be supplied by an implementation in order to create an instance. Each explicit attribute identifies a distinct property. An explicit attribute declaration creates one or more explicit attributes having the indicated domain, and assigns an identifier to each.[13]

EXPRESS syntax:

```
203 explicit_attr = attribute_decl { ';' attribute_decl} ':'
[OPTIONAL] base_type ';'.
167 attribute_decl = attribute_id | qualified_attribute.
171 base_type = aggregation-type | simple_type | named_types.
```

Mapping:
Every attribute of an EXPRESS entity maps into an attribute of the resulting UOL class. Multivalued attributes are mapped into a UOL attribute with the corresponding cardinality (see 1.2). If the multivalued attribute consists of more than one dimension new attributes have to be introduced by splitting the one n-dimensional attribute into n one-dimensional ones.
If attribute is declared as optional the cardinality is changed to reflect this property.

Attributes which basic type is not one of the EXPRESS simple types BINARY, BOOLEAN, INTEGER, LOGICAL, NUMBER, REAL or STRING are additionally mapped into an unidirectional associations which are stereotyped with *express_association*. The association is directed form the using element to the supplying one. The name of the explicit attribute is transferred into the role of the embedding class against the supplying one.

Example:

EXPRESS

```
ENTITY Entity2;
   a, b : REAL;
   c : INTEGER;
   d : OPTIONAL STRING (10);
   e : OPTIONAL ARRAY [2:5] OF OPTIONAL REAL;
   f : LIST [1:5] OF LIST [0:10] OF STRING;
END_ENTITY;
```

UOL

```
class Entity2
   stereotyped with express_entity
   feature {any}
      a : REAL;
      b : REAL;
      d [0..1] : INTEGER with tag values (<width,10>);
      e [0,2..5] : REAL with tag values (<array>,<OPTIONAL>);
      f [1..5] : ?001 with tag values (<list>);
      ?001 [0..10] : STRING
   end
end
```

See also example of section 1.2

[13] Cf. [ISO EXPRESS RM 94] 36

5.3.2.2.1.2 Derived attribute

EXPRESS semantics:
A derived attribute represents a property whose value is composed by evaluating an expression. Derived attributes are declared following the DERIVE keyword. The declaration consists of the attribute identifier, its representation type and an expression to be used to compute the attribute value.[14]

EXPRESS syntax:

```
190 derived_attr = attribute_decl ':' base_type ':=' expression
';'.
167 attribute_decl = attribute_id | qualified_attribute.
171 base_type = aggregation_types | simple_types | named_types.
```

Mapping:
Every derived attribute of an EXPRESS entity maps into an attribute of the resulting UOL class. Additionally the attribute gets a tag value "derived" followed by the derivation formula as string.

Example:

EXPRESS

```
ENTITY circle;
   centre : point;
   radius : REAL;
   axis : vector;
DERIVE
   area : REAL := PI*radius**2;
   perimeter : REAL := REAL := 2.0*PI*radius;
END_ENTITY;
```

UOL

```
class circle
   stereotyped with express_entity
   feature {any}
      centre : point;
      radius : REAL;
      axis : vector;
      area : REAL
         with tag values (<derived,'PI*radius**2'>);
      perimeter : REAL
         with tag values (<derived,'2.0*PI*radius'>)
   end
end

relation ?001
   stereotyped with express_association
   link circle, point[1..1]
   feature {circle}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {point}
      with tag values (<IsNavigable,TRUE>,
         <AssociationEndName,centre>
   end
end
```

[14] Cf. [ISO EXPRESS RM 94] 37

```
relation ?002
   stereotyped with express_association
   link circle, vector[1..1]
   feature {circle}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {vector}
      with tag values (<IsNavigable,TRUE>,
         <AssociationEndName,axis>)
   end
end
```

5.3.2.2.1.3 Inverse attribute

EXPRESS semantics:
If another entity has established a relationship with the current entity by way of an explicit attribute, an inverse attribute may be used to describe that relationship in the context of the current entity. This inverse attribute may also be used to constrain the relationship further.
Inverse attributes are declared following the INVERSE keyword.[15]

EXPRESS syntax:

```
234 inverse_attr = attribute_decl ':' [ (SET|BAG) [bound_spec] OF
] entity_ref FOR attriubte_ref ';'.
167 attribute_decl = attribute_id | qualified_attribute.
176 bound_spec = '[' bound_1 ':' bound_2 ']'.
174 bound_1 = numeric_expression.
175 bound_2 = numeric_expression.
```

Mapping:
The inverse attribute is mapped similar to the explicit one. Additionally the tag value "inverse" and the name of the referenced attribute as its value.

Since the inverse attribute is the inverse direction of the implicit relationship stated by an explicit attribute which type is an entity it's not explicated just as the explicit attribute.

EXPRESS

```
ENTITY door;
   handle : knob;
   hinges : SET [1:?] OF hinge;
END_ENTITY;

ENTITY knob;
...
INVERSE
   opens : door FOR handle;
END_ENTITY;
```

UOL

```
class door
   stereotyped with express_entity
   feature {any}
      handle : knob;
      hinges [1..*] : hinge with tag values (<set>)
   end
end

class knob
   stereotyped with express_entity
   feature {any}
      opens : door with tag values (<inverse,handle>)
   end
end
```

[15] Cf. [ISO EXPRESS RM 94] 38

```
relation ?001
   stereotyped with express_association
   link door, knob[1..1]
   feature {door}
      with tag values (<IsNavigable, FALSE>)
   end
   feature {knob}
      with tag values (<IsNavigable, TRUE>)
   end
end

relation ?001
   stereotyped with express_association
   link door, knob[1..1]
   feature {door}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {knob}
      with tag values (<IsNavigable,TRUE>,
         <AssociationEndName,handle>)
   end
end

relation ?002
   stereotyped with express_association
   link door, hinge[1..*]
   feature {door}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {hinge}
      with tag values (<IsNavigable,TRUE>,
         <AssociationEndName,hinges>)
   end
end
```

5.3.2.2.2 Local rules

EXPRESS semantics:
Local rules are assertions on the domain of entity instances and thus apply to all instances of that entity data type. There are two kinds of local rules. Uniqueness rules control the uniqueness of attribute values among all instances of a given entity data type. Domain rules describe other constraints on or among the values of the attributes of each instance of a given entity data type.[16]

5.3.2.2.2.1 Uniqueness rule

EXPRESS semantics:
A uniqueness constraint for individual attributes or combinations of attributes may be specified in a uniqueness rule. The uniqueness rules follow the UNIQUE keyword, and specify either a single attribute name or a list or attribute names. A rule, which specifies a single attribute name, called a simple uniqueness rule, specifies that no two instances of the entity data type in the domain shall use the same instance for the named attribute. A rule, which specifies two or more attribute names, called a joint uniqueness rule, specifies that no two instances of the entity data type shall have the same combination of instances for the named attributes.[17]

EXPRESS syntax:
```
310 unique_clause = UNIQUE unique_rule ‘;’ {unique_rule ‘;’}.
311 unique_rule = [label ‘:’] refernced_attribute {‘;’
referenced_attribute}.
266 referenced_attribute = attribute_ref | qualified_attribute.
```

Mapping:
The uniqueness rule is transferred into the tag value “express_unique” of the affected attribute(s). The tag value is true if the attribute is unique or holds the name of the uniqueness rule (i.e. the label specified in EXPRESS). If the tag values of two or more attributes hold the same value for the “unique” tag the two attributes combined should be unique. If no label is specified in EXPRESS and a uniqueness constraint should apply to two or more attributes, the transformation processor creates an internal name.

[16] Cf. [ISO EXPRESS RM 94] 40
[17] Cf. [ISO EXPRESS RM 94] 40

Example:

EXPRESS

```
ENTITY e;
   a, b, c : INTEGER;
UNIQUE
   ur1 : a;
   ur2 : b;
   ur3 : c;
END_ENTITY;

ENTITY person_name;
   last : STRING;
   first : STRING;
   middle : STRING;
   nickname : STRING;
END_ENTITY;
ENTITY employee;
   badge : NUMBER;
   name : person_name;
UNIQUE
   ur1: badge, name;
END_ENTITY;
```

UOL

```
class e
   stereotyped with express_entity
   feature {any}
      a : INTEGER with tag values (<express_unique,ur1>);
      b : INTEGER with tag values (<express_unique,ur2>);
      c : INTEGER with tag values (<express_unique,ur3>)
   end
end

class person_name
   stereotyped with express_entity
   feature {any}
      last : STRING;
      first : STRING;
      middle : STRING;
      nickname : STRING
   end
end

class employee
   stereotyped with express_entity
   feature {any}
      badge : NUMBER with tag values (<express_unique,ur1>);
      name : person_name with tag values (<express_unique,ur1>)
   end
end
```

(UOL 1.2)

```
relation ?001
   stereotyped with express_association
   link employee, person_name[1..1]
   feature {employee}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {person_name}
      with tag values (<IsNavigable,TRUE>,
         <AssociationEndName,name>)
   end
end
```

5.3.2.2.2.2 Domain rules (WHERE clause)

EXPRESS semantics:
Domain rules constrain the values of individual attributes or combinations of attributes for every entity instance. All domain rules follow the WHERE keyword.[18]

EXPRESS syntax:
```
315 where_clause = WHERE domain_rule ';' {domain_rule ';'}.
```

Mapping:
Each domain rule maps into a tag value "domain rule" optionally followed by the domain rule label. If a rule label is specified the rule label introduces an additional tag value followed by the rule as string. If not rule label is specified the transformation processor generates an internal name.

Example:

EXPRESS

```
ENTITY unit_vector1;
   a, b, c : REAL;
WHERE
   length_1 : a**2 + b**2 + c**2 = 1.0;
END_ENTITY;

ENTITY unit_vector2;
   a, b : REAL;
   c : OPTIONAL REAL;
WHERE
   length_1 : a**2 + b**2 + c**2 = 1.0;
END_ENTITY

ENTITY unit_vector3;
   a, b : REAL;
   c : OPTIONAL REAL;
WHERE
   length_1 : a**2 + b**2 + NVL(c, 0.0) = 1.0;
END_ENTITY;
```

UOL

```
class unit_vector1
   stereotyped with express_entity
   with tag values (<domain_rule,length_1>,
      <length_1, 'a**2 + b**2 + c**2 = 1.0')
   feature {any}
   a : REAL;
   b : REAL;
   c : REAL
   end
end
```

[18] Cf. [ISO EXPRESS RM 94] 41

(UOL 1.2)

```
class unit_vector2
   stereotyped with express_entity
   with tag values (<domain_rule,length_1>,
      <lenght_1, 'a**2 + b**2 + c**2 = 1.0'>)
   feature {any}
   a : REAL;
   b : REAL;
   c [0..1] : REAL
   end
end

class unit_vector3
   stereotyped with express_entity
   with tag values (<domain_rule,length_1>,
      <lenght_1, 'a**2 + b**2 + NVL(c, 0.0) = 1.0'>)
   feature {any}
      a : REAL;
      b : REAL;
      c [0..1] : REAL
   end
end
```

5.3.2.2.3 Subtypes and supertypes

EXPRESS semantics:
EXPRESS allows for the specification of entities as subtypes of other entities, where a subtype entity is a specialization of its supertype. This establishes an inheritance (i.e., subtype/supertype) relationship between the entities in which the subtype inherits the properties (i.e., attributes and constraints) of its supertype. Successive subtype/supertype relationships establish an inheritance graph in which every instance of a subtype is an instance of its supertype(s).[19]

EXPRESS syntax:
```
294 supsuper = [subpertype_constraint] [subtype_declaration].
297 supertype_constraint = abstract_supertype_declaration |
supertype_rule.
156 abstract_supertype_declaration = ABSTRACT SUPERTYPE
[supertype_constraint].
295 subtype_constraint = OF ‘(‘ supertype_expression ‘)’.
298 supertype_expression = supertype_factor { ANDOR
supertype_factor}.
299 supertype_factor = supertype_term { AND supertype_term }.
301 supertype-term = entity_ref | one_of | ‘(‘
supertype_expression ‘)’.
250 one_of = ONEOF ‘(‘ supertype_expression { ‘;’
supertype_expression } ‘)’.
300 supertype_rule = SUPERTYPE subtype_constraint.
```

5.3.2.2.3.1 Specifying subtypes

EXPRESS semantics:
An entity is a subtype if it contains a SUBTYPE declaration. The subtype declaration shall identify all the entity’s immediate supertype(s).[20]

EXPRESS syntax:
```
296 subtype_declaration = SUBTYPE OF ‘(‘ entity_ref { ‘;’
entity_ref } ‘)’.
```

[19] Cf. [ISO EXPRESS RM 94] 43
[20] Cf. [ISO EXPRESS RM 94] 44

5.3.2.2.3.2 Specifying supertypes

EXPRESS semantics:
An entity is a supertype through either an explicit or implicit specification. An entity is explicitly specified to be a supertype if it contains an ABSTRACT SUPERTYPE declaration and is implicitly specified to be a supertype if it is named in a subtype declaration of at least one other entity.[21]

EXPRESS syntax:

```
297 supertype_constraint = abstract_supertype_declaration |
supertype_rule.
156 abstract_supertype_declaration = ABSTRACT SUPERTYPE
[subtype_constraint].
295 subtype_constraint = OF '(' supertype_expression ')'.
298 supertype_expression = supertype_factor { ANDOR
supertype_factor }.
299 supertype_factor = supertype_term { AND supertype_term }.
301 supertype_term = entity_ref | one_of | '('
supertype_expression ')'.
250 one_of = ONEOF '(' supertype_expression { ';'
supertype_expression } ')'.
300 supertype_rule = SUPERTYPE subtype_constraint.
```

Mapping:
Inheritance between entities map into inheritance between the resulting classes.

Example:

EXPRESS

```
ENTITY integer_number;
   val : INTEGER;
END_ENTITY;

ENTITY odd_number
   SUBTYPE OF (integer_number);
WHERE
   not_even : ODD (val);
END_ENTITY;
```

UOL

```
class integer_number
   stereotyped with express_entity
   feature {any}
      val : INTEGER
   end
end

class odd_number
   stereotyped with express_entity
   with tag values (<domain_rule,not_even>,
      <not_even,'ODD(val)'>)
   inherit integer_number
end
```

[21] Cf. [ISO EXPRESS RM 94] 44

5.3.2.2.3.3 Attribute inheritance

EXPRESS semantics:
The attribute identifiers in a supertype are visible within the scope of the subtype. Thus, a subtype inherits all of the attributes of its supertype. This allows the subtypes to specify either constraints or their own attributes using the inherited attribute. If a subtype has more than one supertype, subtype inherits all of the attributes from all of its supertypes.[22]

Mapping:
Like the single inheritance of the prior example, the EXPRESS inheritance maps into an inheritance hierarchy in UOL.

Example:

EXPRESS

```
ENTITY e1;
   attr : REAL;
END_ENTITY;

ENTITY e2;
   attr : BINARY;
END_ENTITY;

ENTITY e12
   SUBTYPE OF (e1, e2);
WHERE
   positive : SELF\e1.attr > 0.0;
END_ENTITY;
```

UOL

```
class e1
   stereotyped with express_entity
   feature {any}
      attr : REAL
   end
end

class e2
   stereotyped with express_entity
   feature {any}
      attr : BINARY
   end
end

class e12
   stereotyped with express_entity
   inherit e1,e2
   with tag values (<domain_rule,positive>,
      <positive,' SELF\\e1.attr > 0.0')
end
```

Remark: UOL recognizes the backslash '\' character as escape character and hence the backslash of the EXPRESS source is converted into two subsequent ones.

[22] Cf. [ISO EXPRESS RM 94] 45

5.3.2.2.3.4 Attribute declaration

EXPRESS semantics:
An attribute declaration in a supertype can be redeclared in a subtype. The attribute remains in the supertype but the allowed domain of values for that attribute is governed by the redeclaration given in the subtype.[23]

EXPRESS syntax:

```
262 qualified_attribute = SELF group_qualifier
attribute_qualifier.
219 group_qualifier = '\' entity_ref.
169 attribute_qualifier = '.' attribute_ref.
```

Mapping:
Every redeclared attribute at subtype level maps into an explicit attribute declaration in the subclass. A tag "redeclaration" with the name of the source class as value clarifies the redeclaration within the redeclaring subclass.

Example I:

EXPRESS

```
ENTITY point;
   x : NUMBER;
   y : NUMBER;
END_ENTITY;

ENTITY integer_point
   SUBTYPE OF (point);
   SELF\point.x : INTEGER;
   SELF\point.y : INTEGER;
END_ENTITY;
```

UOL

```
class point
   stereotyped with express_entity
   feature {any}
      x : NUMBER;
      y : NUMBER
   end
end

class integer_point
   stereotyped with express_entity
   inherit point
   feature {any}
      x : INTEGER with tag values (<redeclaration,point>);
      y : INTEGER with tag values (<redeclaration,point>)
   end
end
```

[23] Cf. [ISO EXPRESS RM 94] 46

Example II:

EXPRESS

```
ENTITY super;
   things : LIST [3:?] OF thing;
   items : BAG [0:?] of widget;
   may_be : OPTIONAL stuff;
END_ENTITY;

ENTITY sub
   SUBTYPE OF (super);
   SELF\super.things : LIST [3:?] OF UNIQUE thing;
   SELF\super.items : SET [1:10] OF widget;
   SELF\super.may_be : stuff;
END_ENTITY;
```

UOL

```
class super
   stereotyped with express_entity
   feature {any}
      things [3..*] : thing with tag values (<list>);
      items [0..*] : widget with tag values (<bag>);
      may_be [0..1] : stuff
   end
end

relation ?001
   stereotyped with express_association
   link super, thing[3..*]
   feature {super}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {thing}
      with tag values (<IsNavigable, TRUE>,
         <AssociationEndName,things>)
   end
end

relation ?002
   stereotyped with express_association
   link super, widget[0..*]
   feature {super}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {widget}
      with tag values (<IsNavigable, TRUE>,
         <AssociationEndName,items>)
   end
end
```

```
relation ?003
   stereotyped with express_association
   link super, stuff[0..1]
   feature {super}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {stuff}
      with tag values (<IsNavigable, TRUE>,
         <AssociationEndName,may_be>)
   end
end

class sub
   stereotyped with express_entity
   inherit super
   feature {any}
      things [3..*] : thing
         with tag values (<list>,<UNIQUE>,
            <redeclaration,super>);
      items [1..10] : widget
         with tag values (<set>,<redeclaration,super>);
      may_be [0..1] : stuff
         with tag values (<redeclaration,super>)
   end
end

relation ?004
   stereotyped with express_association
   link sub, thing[3..*]
   feature {super}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {thing}
      with tag values (<IsNavigable, TRUE>
         <AssociationEndName,things>)
   end
end

relation ?005
   stereotyped with express_association
   link sub, widget[1..10]
   feature {super}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {widget}
      with tag values (<IsNavigable, TRUE>,
         <AssociationEndName,items>)
   end
end
```

```
relation ?006
   stereotyped with express_association
   link sub, stuff[0..1]
   feature {super}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {stuff}
      with tag values (<IsNavigable, TRUE>,
         <AssociationEndName,may_be>)
   end
end
```

Example III:

EXPRESS

```
FUNCTION distance(p1, p2 : point) : REAL;
   (* Compute the shortest distance between two points *)
END_FUNCTION;

FUNCTION NORMAL (p1, p2, p3 : point) : vector
(* Compute normal of a plane given three points on the plane *)
END_FUNCTION;

ENTITY circle;
   centre : point;
   radius : REAL;
   axis : vector;
DERIVE
   area : REAL := PI*radius**2;
END_ENTITY;

ENTITY circle_by_points
   SUBTYPE OF (circle)
   p2 : point;
   p3 : point;
DERIVE
   SELF\circle.radius : REAL := distance(cantre,p2);
   SELF\circle.axis : vector := normal(centre,p2,p3);
WHERE
   not_coincident : (centre <> p2) AND
                    (p2 <> p3) AND
                    (p3 <> centre);
   is_circle : distance (centre,p3) = distance(centre,p2);
END_ENTITY;
```

UOL

```
class schema
   stereotyped with express_schema
   feature {any}
      deferred distance (p1,p2 : point) : REAL is text "distance";
      deferred normal (p1, p2 ,p3 :point) : vector is text "normal"
   end
end

class circle
   stereotyped with express_entity
   feature {any}
      centre : point;
      radius : REAL;
      axis : vector;
      area : REAL with tag values (<derived,'PI*radius**2'>)
   end
end
```

```
relation ?001
   stereotyped with express_association
   link circle, point[1..1]
   feature {circle}
      with tag values (<IsNavigable,FALSE>)
   end
   feature {point}
      with tag values (<IsNavigable,TRUE>,
         <AssociationEndName,centre>)
   end
end

relation ?002
   stereotyped with express_association
   link circle, vector[1..1]
   feature {circle}
      with tag values (<IsNavigable,TRUE>,
         <AssociationEndName,axis>)
   end
end

class circle_by_points
   stereotyped with express_entity
   with tag values (<domain_rule,not_coincident>,
   <not_coincident,'(centre <> p2) AND (p2 <> p3) AND
      (p3 <> centre)'>,
   <domain_rule,is_circle>,
   <is_circle,'distance(centre,p3) = distance(centre,p2)'>)
   inherit circle
   feature {any}
      p2 : point;
      p3 : point;
      radius : REAL
         with tag values (<redeclaration,circle>,
         <derived,'dirstance(centre,p2)'>);
      axis : vector
         with tag values (<redeclaration,circle>,
         <derived,'normal(centre,p2,p3)'>)
   end
end
```

5.3.2.2.3.5 Rule inheritance

EXPRESS semantics:
Every local or global rule that applies to a supertype applies to its subtype(s). Thus, a subtype inherits all the rules of its supertype(s). If a subtype has more than one supertype, the subtype shall inherit all the rules constraining the supertypes.
It is not possible to change or delete any of the rules that are associated with a subtype via rule inheritance but it is possible to add new rules, which further constrain the subtype.[24]

Mapping:
Since rules are not redeclared within the inheriting subtype in the EXPRESS schema they're not redeclared in the resulting UOL class, as well. Additional rules may be declared on the level of the subtype.

Example:

EXPRESS

```
SCHEMA s;
ENTITY person;
   ss_no : INTEGER;
   born : date;
DERIVE
   age : INTEGER := years_since(born);
UNIQUE
   un1 : ss_no;
END_ENTITY;

ENTITY teacher
   SUBTYPE OF (person);
   teaches : SET [1:?] OF course;
WHERE
   old : age >= 21;
END_ENTITY;

ENTITY student
   SUBTYPE OF (person);
   takes : SET [1:?] OF course;
WHERE
   young : age >= 5;
END_ENTITY;

ENTITY graduate
   SUBTYPE OF (student, teacher);
WHERE
   limited : NOT (GRAD_LEVEL IN teaches);
END_ENTITY;

TYPE course = ENUMERATION OF (....., GRAD_LEVEL, ...);
END_TYPE;
END_SCHEMA;
```

[24] Cf. [ISO EXPRESS RM 94] 48

UOL

```
package s is
   class person
      stereotyped with express_entity
      feature {any}
         ss_no : INTEGER with tag values
            (<express_unique,un1>);
         born : date;
         age : INTEGER with tag values
            (<derived,'years_since(born)'>)
      end
   end

   relation ?001
      stereotyped with express_association
      link person, date[1..1]
      feature {person}
         with tag values (<IsNavigable,FALSE>)
      end
      feature {date}
         with tag values (<IsNavigable, TRUE>,
            <AssociationEndName,born>)
      end
   end

   class teacher
      stereotyped with express_entity
      with tag values (<domain_rule,old>, <old,'age >= 21'>)
      inherit person
      feature {any}
         teaches [1..*] : course_ with tag values (<set>)
      end
   end

   relation ?002
      stereotyped with express_association
      link teacher, course_[1..*]
      feature {teacher}
         with tag values (<IsNavigable,FALSE>)
      end
      feature {thing}
         with tag values (<IsNavigable, TRUE>,
            <AssociationEndName,teaches>)
      end
   end

   class student
      stereotyped with express_entity
      with tag values (<domain_rule, young>,
         <young, 'age >= 5'>)
      inherit person
      feature {any}
         takes : unique {Level1,GRAD_LEVEL,Level2}
          with tag values (<set>,<express_enumeration,course_>)
      end
```

(UOL 1.2)

```
        end


        class graduate
           stereotyped with express_entity
           with tag values (<domain_rule, limited>,
              <limited,'NOT (GRADE_LEVEL IN teaches)'>)
           inherit student; teacher
        end

        class course_
           stereotyped with express_enumeration
           with tag values (<Level1>,<GRAD_LEVEL>,<Level2>)
        end
     end
```

Remark: Since "course" is a UOL keyword a tailing underscore "_" is appended to the string during the transformation process. The tool has to remove it when re-transferring the UOL code.

5.3.2.2.4 Subtype/supertype constraints

EXPRESS semantics:
An instance of an entity data type, which is a subtype, is an instance of each of its supertypes. An instance of an entity data type which is either explicitly or implicitly declared to be a supertype may also be an instance of one or more of its subtypes.[25]

EXPRESS syntax:

```
294 subsuper = [supertype_constraint] [subtype_declaration].
297 supertype_constraint = abstract_supertype_declaration |
supertype_rule.
156 abstract_supertype_declaration = ABSTRACT SUPERTYPE
[subtype_constraint].
295 subtype_constraint = OF ‘(‘ supertype_expression ‘)’.
298 supertype_expression = supertype_factor {ANDOR
supertype_factor}.
299 supertype_factor = supertype_term {AND supertype_term}.
301 supertype_term = entity_ref | one_of |
‘(‘ supertype_expression ‘)’.
250 one_of = ONEOF ‘(‘ supertype_expression { ‘,’
supertype_expression} ‘)’.
300 supertype_rule = SUPERTYPE subtype_constraint.
```

[25] Cf. [ISO EXPRESS RM 94] 49

5.3.2.2.4.1 Abstract supertypes

EXPRESS semantics:
EXPRESS allows for the declaration of supertypes that are not intended to be directly instantiated. An entity data type shall include the phrase ABSTRACT SUPERTYPE in a supertype constraint for this purpose. An abstract supertype shall not be instantiated except in conjunction with at least one of its subtypes.[26]

Mapping:
An abstract supertype of EXPRESS corresponds to an abstract UOL class. Additionally, the UOL class has a tag value *abstract*.

Example:

EXPRESS

```
ENTITY vehicle
   ABSTRACT SUPERTYPE
END_ENTITY;

ENTITY land_based
   SUBTYPE OF (vehicle);
END_ENTITY;

ENTITY water_based
   SUBTYPE OF (vehicle);
END_ENTITY;
```

UOL

```
class vehicle
   stereotyped with express_entity
   with tag values (<abstract>)
end

class land_based
   stereotyped with express_entity
   inherit vehicle
end

class water_based
   stereotyped with express_entity
   inherit vehicle
end
```

[26] Cf. [ISO EXPRESS RM 94] 50

5.3.2.2.4.2 ONEOF

EXPRESS semantics:
The ONEOF constraint states that the elements of the ONEOF list are mutually exclusive. None of the elements may be instantiated with any other element in the list. Each element shall be a supertype expression, which may resolve to a single subtype of the entity data type.[27]

EXPRESS syntax:
```
250 one_of = ONEOF ‘(‘ supertype_expression { ‘,’
supertype_expression } ‘)’.
299 supertype_factor = supertype_term { AND supertype_term}.
301 supertype_term = entity_ref | one_of | ‘(‘
supertype_expression ‘)’.
```

Mapping:
The EXPRESS *ONEOF* constraint is transferred into a constraint applied on all of the resulting inheritance relationships. This constrained is named *EXPRESS_ONEOF*.

Example:

EXPRESS

```
ENTITY pet
   ABSTRACT SUPERTYPE OF (ONEOF(cat, rabbit, dog));
   name : pet_name;
END_ENTITY;

ENTITY cat
   SUBTYPE OF (pet);
END_ENTITY;

ENTITY rabbit
   SUBTYPE OF (pet);
END_ENTITY;

ENTITY dog
   SUBTYPE OF (pet);
END_ENTITY;
```

UOL

```
class pet
   stereotyped with express_entity
   with tag values (<abstract>)
end

class cat
   stereotyped with express_entity
   inherit pet constrained by {EXPRESS_ONEOF}
end

class rabbit
   stereotyped with express_entity
   inherit pet constrained by {EXPRESS_ONEOF}
end
class dog
   stereotyped with express_entity
   inherit pet constrained by {EXPRESS_ONEOF}
end
```

[27] Cf. [ISO EXPRESS RM 94] 50

5.3.2.2.4.3 ANDOR

EXPRESS semantics:
If the subtypes are not mutually exclusive, that is, an instance of the supertype may be an instance of more than one of its subtypes, the relationship between the subtypes shall be specified using the ANDOR constraint.[28]

Mapping:
The EXPRESS *ANDOR* constraint is transferred into a constraint applied on all of the resulting inheritance relationships. This constrained is named *EXPRESS_ANDOR.*

Example:

EXPRESS

```
ENTITY person
   SUPERTYPE OF (employee ANDOR student);
END_ENTITY;

ENTITY employee
   SUBTYPE OF (person);
END_ENTITY;

ENTITY student
   SUBTYPE OF (person);
END_ENTITY;
```

UOL

```
class person
   stereotyped with express_entity
end

class employee
   stereotyped with express_entity
   inherit person constrained with {EXPRESS_ANDOR}
end

class student
   stereotyped with express_entity
   inherit person constrained by {EXPRESS_ANDOR}
end
```

[28] Cf. [ISO EXPRESS RM 94] 51

5.3.2.2.4.4 AND

EXPRESS semantics:
If the supertype instances are categorized into multiple groups of mutually exclusive subtypes (i.e., multiple ONEOF groupings) indicating that there is more than one way to completely categorize the supertype, the relationship between those groups shall be specified sing the AND constraint. The AND constraint is only used to relate groupings established by other subtype/supertype constraints.[29]

Mapping:
The EXPRESS *AND* constraint is transferred into a constraint applied on all of the resulting inheritance relationships. This constrained is named *EXPRESS_AND.*

Every inheritance path gets a unique name, which serves as anchor point of a constrained nondirected dependency relation connecting them. The relation describing the intra-inheritance constrained is stereotyped with *EXPRESS_INHERITANCE_CONSTRAINT*.

Example:

EXPRESS

```
ENTITY person
   SUPERTYPE OF (ONEOF(male,female) AND ONEOF(citizen,alien));
END_ENTITY;

ENTITY male
   SUBTYPE OF (person)
END_ENTITY;

ENTITY female
   SUBTYPE OF (person)
END_ENTITY;

ENTITY citizen
   SUBTYPE OF (person)
END_ENTITY;

ENTITY alien
   SUBTYPE OF (person)
END_ENTITY;
```

UOL

```
class person
   stereotyped with express_entity
end

class male
   stereotyped with express_entity
   with tag values (<inheritance_id,?001>)
   inherit person
end

class female
   stereotyped with express_entity
   with tag values (<inheritance_id,?002>)
   inherit person
```

[29] Cf. [ISO EXPRESS RM 94] 52

```
        end


        class citizen
           stereotyped with express_entity
           with tag values (<inheritance_id,?003>)
           inherit person constrained by {EXPRESS_ONEOF}
        end

        class alien
           stereotyped with express_entity
           with tag values (<inheritance_id,?004>)
           inherit person constrained by {EXPRESS_ONEOF}
        end

        relation ?005
           stereotyped with express_inheritance_constraint
           link ?001, ?002
           constrained by {text “express_oneof”}
        end

        relation ?006
           stereotyped with express_inheritance_constraint
           link ?003, ?004
           constrained by {text “express_oneof”}
        end

        relation ?007
           stereotyped with express_inheritance_constraint
           link ?005, ?006
           constrained by {text “express_and”}
        end
```

5.3.2.2.4.5 Precedence of supertype operators

EXPRESS semantics:
The evaluation of supertype expressions proceeds from left to right, with the highest precedence operators being evaluated first. The table summarizes the precedence rules for the supertype expression operators. Operators in the same row have the same precedence, and the rows are ordered by decreasing precedence.

Precedence	Operators
1	() ONEOF
2	AND
3	ANDOR

Mapping:
If the precedence is stated explicitly by using additional brackets the brackets are transferred into UOL.

Since UOL serves as transfer format with the same expressive power as EXPRESS there's no need to state the implicit precedence rules of EXPRESS explicitly in UOL. The retransformation has to ensure the preservation of the precedence order.

Example:

EXPRESS

`ENTITY x`
`  SUPERTYPE OF (a ANDOR b AND c);`
`END_ENTITY;`
 	
`ENTITY x`
`  SUPERTYPE OF ((a ANDOR b) AND c);`
`END_ENTITY;`

UOL

5.3.2.2.5 Implicit declarations

EXPRESS semantics:
When an entity is declared, a constructor is also implicitly declared. The constructor identifier is the same as the entity identifier and the visibility of the constructor declaration is the same as that of the entity declaration.

The constructor, when invoked, shall return a partial complex entity value for that entity data type of the point of invocation. Each attribute in this partial complex entity value is given by the actual parameter passed in the constructor call, if an actual parameter is an entity instance, that entity instance plays the role described by an attribute in the partial complex entity value. The constructor shall only provide the attribute, which are explicit in a particular entity declaration.[30]

EXPRESS syntax:
```
195 entity_constructor = entity_ref ‘(‘ [expression { ‘;’
expression } ] ‘)’.
```

Mapping:
Since the constructor is only implicit declared it is not explicited with in the UOL.

5.3.2.3 Schema

EXPRESS semantics:
A SCHEMA declaration defines a common scope for a collection of related entity and other data type declarations.[31]

EXPRESS syntax:
```
281 schema_decl = SCHEMA schema_id ‘;’ schema_body END_SCHEMA ‘;’.
280 schema_body = {interface_specification} [constant_decl]
{declaration | rule_decl}.
228 interface_specification = reference_clause | use_cluase.
189 declaration = entity_decl | function_decl | procedure_decl |
type_decl.
```

Mapping:
The schema definition matches to the UOL package. The package gets the same name as the EXPRESS schema. Additionally a “schema class” is created within the package covering all the schema global accessible information. This class is stereotyped with “express_schema”.

Example:

EXPRESS
SCHEMA test; END_SCHEMA;
UOL

package test is
  class test
  stereotyped with express_schema
  end
end

[30] Cf. [ISO EXPRESS RM 94] 53-54
[31] Cf. [ISO EXPRESS RM 94] 55

5.3.2.4 Constant

EXPRESS semantics:
A constant declaration is used to declare named constants. The scope of a constant identifier shall be the function, procedure, rule or schema in which the constant declaration occurs. A named constant appearing in a CONSTANT declaration shall have an explicit initialization the value of which is computed by evaluating the expression. A named constant may appear in the declaration of another named constant.[32]

EXPRESS syntax:
```
185 constant_decl = CONSTANT constant_body {constant_body}
END_CONSTANT ';'.
184 constant_body = constant_id ':' base_type ':=' expression ';'.
171 base_type = aggregation_types | simple_types | named_types.
```

Mapping:
The EXPRESS constants match immutable attributes of the UOL. The value of the constant remains unchanged as string within UOL.
Depending on the occurrence of the constant declaration it's mapped into:

- an attribute of the schema class – if it's declared on schema level
- a part of the whole function mapping – if it's declared within a function
- a part of the whole procedure mapping – it's declared within a procedure
- a part of the whole rule mapping – if it's declared within a rule

Example:

EXPRESS

```
CONSTANT
   thousand : NUMBER := 1000;
   million : NUMBER := thousand**2;
   origin : point := point(0.0, 0.0, 0.0);
END_CONSTANT;
```

UOL

```
frozen thousand : NUMBER is '1000';
frozen million : NUMBER is 'thousand**2';
frozen origin : point is 'point(0.0, 0.0, 0.0)
```

[32] Cf. [ISO EXPRESS RM 94] 56

5.3.2.5 Algorithms

EXPRESS semantics:
An algorithm is a sequence of statements that produces some desired and state. The two kinds of algorithms that can be specified are functions and procedures.[33]

5.3.2.5.1 Function

EXPRESS semantics:
A function is an algorithm, which operates on parameters and that, produces a single resultant value of a specific data type.[34]

EXPRESS syntax:

```
208 function_decl = function_head [algorithm_head] stmt {stmt}
END_FUNCTION';'.
209 function_head = FUNCTION function_id [ '(' formal_parameter {
';' formal_parameter } ')' ] ':' parameter_type ';'.
206 formal_parameter = parameter_id { ',' parameter_id } ':'
parameter_type.
253 parameter_type = generalized_type | named types |
simple_types.
163 algorithm_head = {declaration} [constant_decl] [local_decl].
189 declaration = entity_decl | function_decl | procedure_decl |
type_decl.
```

Mapping:
Since functions are declared on schema level (i.e., the same level as the entity declaration) they correspond to operations declared within the UOL schema class resulting from the EXPRESS schema. Additionally the UOL operation is stereotyped with “express_function”.

Furthermore EXPRESS doesn’t support any visibility constraints; hence all corresponding UOL functions have public visibility.

[33] Cf. [ISO EXPRESS RM 94] 56
[34] Cf. [ISO EXPRESS RM 94] 57

5.3.2.5.2 Procedure

EXPRESS semantics:
A procedure is an algorithm that receives parameters from the point of invocation and operates on them in some manner to produce the desired end state. Changes to the parameters within a procedure are only reflected to the point of invocation when the formal parameter is preceded by the VAR keyword.[35]

EXPRESS syntax:

```
258 procedure_decl = procedure_head [algorithm_head] {stmt}
END_PROCEDURE ‘;’.
259 proedure_head = PROCEDURE procedure_id [ ‘(‘ [VAR]
formal_parameter { ‘;’ [VAR] formal_parameter} ‘)’ ] ‘;’.
206 formal_parameter = parameter_id { ‘,’ parameter_id } ‘:’
parameter_type.
253 parameter_type = generalized_types | named_types |
simple_types.
163 algorithm_head = {declaration} [constant_decl] [local_decl].
189 declaration= entity_decl | function_decl | type_decl.
```

Mapping:
An EXPRESS procedure corresponds to an operation of schema class. This operation is stereotyped with *express_procedure*.

[35] Cf. [ISO EXPRESS RM 94] 58

5.3.2.5.3 Parameters

EXPRESS semantics:
A function or procedure can have formal parameters. Each formal parameter specifies a name and parameter type. The name is an identifier, which shall be unique within the scope of the function or procedure. A formal parameter to a procedure may also be declared as VAR (variable), which means that, if the parameter is changed within the procedure, the change shall be propagated to the point of invocation. Parameters not declared as VAR can be changed also, but the change will not be apparent when control is returned to the caller.[36]

EXPRESS syntax:

```
206 formal_parameter = parameter_id { ‘,’ parameter_id} ‘:’
parameter_ype.
253 parameter_type = generalized_types | named_types |
simple_types.
```

Mapping:
The formal parameter list of EXPRESS corresponds to the formal parameter list of UOL. If a parameter is declared as VAR within the EXPRESS schema the tag value *express_VAR_parameter* reflects the name of the parameter declared so.

Example:

EXPRESS

```
FUNCTION dist (p1, p2 : point) : REAL;

PROCEDURE midpt (p1, p2 : point; VAR result : point);
```

UOL

```
class schema_class
   stereotyped with express_schema
   feature {any}
      dist (p1,p2 : point):REAL
         stereotyped with express_function
         text "imlementation" is text "...";
      midpt (p1, p2, result : point)
         stereotyped with express_procedure
         with tag values (<VAR_parameter,result>)
         text "implementation" is text "..."
   end
end
```

[36] Cf. [ISO EXPRESS RM 94] 58

5.3.2.5.3.1 Aggregate data type

EXPRESS semantics:
An AGGREGATE data type is a generalization of all aggregation data types.
When a procedure or function which has a formal parameter defined to be an aggregate data type is invoked, the actual parameter passed shall be an ARRAY, BAG, LIST or SET. The operations that can be performed shall then depend on the data type of the actual parameter.

Type labels may be used to ensure that two or more parameters passed are of the same data type, or that the return data type is the same as one of the passed parameters, irrespective of the actual data type passed.[37]

EXPRESS syntax:

```
161 aggregate_type = AGGREGATE [ ':' type_lavel ] OF
parameter_type.
306 type_label = type_label_id | type_label_ref.
253 parameter_type = generalized_types | named_types |
simple_types.
```

Mapping:
The EXPRESS aggregate types map UOL's multi valued parameters. The multiplicity is stated as 1..*.

Example:

EXPRESS

```
FUNCTION scale (input:AGGREGATE:intype OF NUMBER;
                scalar:NUMBER):AGGREGATE:intype OF NUMBER;
   LOCAL
      result : AGGREGATE:intype OF NUMBER;
   END_LOCAL;

   REPEAT i := LOINDEX(input) TO HIINDEX(input);
      result[i] := scalar * input[i];
   END_REPEAT;

   RETURN(result);
END_FUNCTION;
```

UOL

```
scale (input [1..*] : NUMBER, scalar : NUMBER) : [1..*] NUMBER
   stereotyped with express_function
   with tag values (<input,intype>)
   text "implementation" is text
      "LOCAL
           result : AGGREGATE:intype OF NUMBER;
        END_LOCAL;

        REPEAT i := LOINDEX(input) TO HIINDEX(input);
           result[i] := scalar * input[i];
        END_REPEAT;

        RETURN(result);"
```

[37] Cf. [ISO EXPRESS RM 94] 59

5.3.2.5.3.2 Generic data type

EXPRESS semantics:
A GENERIC data type is a generalization of all other data types.
When a procedure or function is invoked with a generic parameter, the actual parameter passed may not be of GENERIC data type. The operations that can be performed depend on the data type of the actual parameter.

The labels may be used to ensure that two or more parameters passed are of the same data type, or that the return data type is the same as one of the passed parameters, irrespective of the actual data types passed.[38]

EXPRESS syntax:
```
218 generic_type = GENERIC [ ':' type_label ].
306 type_label = type_label_id | type_label_ref.
```

Mapping:
The GENERIC data type labeled with type type_label corresponds to a UOL operation parameter of type "GENERIC" additionally enriched by a tag value stating the type_label of the given formal parameter.
If the return value is GENERIC, the tag value "express_return_type" is used.

Example:

EXPRESS

```
FUNCTION add(a,b : GENERIC : intype) : GENERIC:intype;
   LOCAL
      nr : NUMBER;
      vr : vector;
   END_LOCAL;

   IF ('NUMBER' IN TYPEOF(a)) AND ('NUMBER' IN TYPEOF(b)) THEN
      nr := a+b;
      RETURN(nr);
   ELSE
      IF ('THIS_SCHEMA.VECTOR' IN TYPEOF(a)) AND
         ('THIS_SCHEMA.VECTOR' IN TYPEOF(b)) THEN
         vr := vector (a.i + b.i, a.j + b.j, a.k +b.k);
         RETURN (vr);
      END_IF;
   END_IF;
   RETURN(?);
END_FUNCTION;
```

[38] Cf. [ISO EXPRESS RM 94] 60

UOL

```
add (a,b : GENERIC) : GENERIC
   stereotyped with express_function
   with tag values (<a,intype>,
      <b,intype>,<express_return_type,intype>)
   text “implementation” is text
      “ LOCAL
         nr : NUMBER;
         vr : vector;
      END_LOCAL;

      IF (‘NUMBER’ IN TYPEOF(a)) AND
         (‘NUMBER’ IN TYPEOF(b)) THEN
            nr := a+b;
         RETURN(nr);
      ELSE
         IF (‘THIS_SCHEMA.VECTOR’ IN TYPEOF(a)) AND
            (‘THIS_SCHEMA.VECTOR’ IN TYPEOF(b)) THEN
            vr := vector (a.i + b.i, a.j + b.j, a.k +b.k);
            RETURN (vr);
         END_IF;
      END_IF;
      RETURN(?);
   END_FUNCTION;”
```

5.3.2.5.3.3 Type labels

EXPRESS semantics:
Type labels shall be used to relate data type of an actual parameter at invocation to the data types of another actual parameters, local variables, or the return type of a function. Type labels are declared for AGGREGATE and GENERIC data types within the formal parameter declaration of a function or procedure and may be referenced by AGGREGATE or GENERIC data types in the formal parameter declaration, local variable declaration or the declaration of the returned data type of a FUNCTION.[39]

EXPRESS syntax:
```
306 type_label = type_label_id | type_label_ref.
```

Mapping:
The transfer format doesn't check the underlying EXPRESS semantics. Hence there's no difference in the mapping between declaration and reference.

Example:

EXPRESS

```
ENTITY a;
...
END_ENTITY;

ENTITY b SUBTYPE OF (a);
...
END_ENTITY;

ENTITY c SUBTYPE OF (b);
...
END_ENTITY;

FUNCTION test (p1 :GENERIC:x; p2:GENERIC:x):GENERIC:x;
...
END_FUNCTION;
```

UOL

```
class a
   stereotyped with express_entity
end

class b
   stereotyped with express_entity
   inherit a
end

class c
   stereotyped with express_entity
   inherit b
end
```

[39] Cf. [ISO EXPRESS RM 94] 60

```
class schema
   stereotyped with express_schema
   feature {any}
      test (p1:GENERIC; p2:GENERIC) : GENERIC
         stereotyped with express_function
         with tag values (<p1,x>, <p2,x>,
            <express_return_value,x>
      text “implementation” is text
         “...”
end
```

5.3.2.5.3.4 General aggregation data types

EXPRESS semantics:
General aggregation data types from part of the class of types called generalized data types. They represent a specific aggregation data type (ARRAY, BAG, LIST and SET) with a relaxing of the constraints which would normally be applied when specifying the aggregation data type (i.e., a `general_list_type` is a generalization of a `list_type`).[40]

EXPRESS syntax:

```
212 general_aggregation_types = general_array_type |
general_bag_type | general_list_type | general_set_type.
213 general_array_tpe = ARRAY [bound_spec] OF [OPTIONAL] [UNIQUE]
parameter_type.
176 bound_spec = ‘[‘ bound_1 ‘:’ bound_2 ‘]’.
174 bound_1 = numeric_expression.
175 bound_2 = numeric_expression.
253 parameter_type = generalized_types | named_types |
simple_types.
214 general_bag_type = BAG [bound_spec] OF parameter_type.
215 general_list_type = LIST [bound_spec] OF [UNIQUE]
parameter_type.
217 general_set_type = SET [bound_spec] OF parameter_type.
```

Mapping:
The aggregation typed formal parameter corresponds to a multi valued formal parameter within the UOL. The concrete EXPRESS aggregation type is stated as tag value, with the name of the formal parameter as tag and the type of the aggregation as value.

Example:

EXPRESS

```
FUNCTION dimension (input:SET [2:3] OF GENERIC):INTEGER;
```

UOL

```
dimension (input [2..3] GENERIC) : INTEGER
   stereotyped with express_function
   with tag values (<input,set>
```

[40] Cf. [ISO EXPRESS RM 94] 62

5.3.2.5.4 Local variables

EXPRESS semantics:
Variables local to an algorithm are declared after the LOCAL keyword. A local variable is only visible within the scope of the algorithm in which it is declared. Local variables may be assigned values and may participate in expressions.[41]

EXPRESS syntax:

```
239 local_decl = LOCAL local_variable {local_variable} END_LOCAL
';'.
240 local_variable = variable_id { ',' variable_id } ':'
parameter_type [ ':=' expression ] ';'.
253 parameter_type = generalized_types | named_types |
simple_types.
```

Mapping:
Since the local variable declaration occurs only within an algorithm declaration the whole local section corresponds to the implementation text string describing the implementation unchanged.

Example:

EXPRESS

```
FUNCTION f1:INTEGER;
   LOCAL
      r_result : REAL := 0.0;
      i_result : INTEGER;
   END_LOCAL;
...
EXISTS(r_result)
EXISTS(i_result)
END_FUNCTION;
```

UOL

```
f1():INTEGER
   stereotyped with express_function
   text "implementation" is text
      "LOCAL
         r_result : REAL := 0.0;
         i_result : INTEGER;
      END_LOCAL;
      ...
      EXISTS(r_result)
      EXISTS(i_result)
      END_FUNCTION;"
```

[41] Cf. [ISO EXPRESS RM 94] 69

5.3.2.6 Rule

EXPRESS semantics:
Rules permit the definition of constraints that apply to one or more entity data types within the scope of a schema. Local rules (i.e., the uniqueness constraint and domain rules in an entity declaration) declare constraints that apply individually to every instance of an entity data type. A RULE declaration permits the definition of constraints that apply collectively to the entire domain of an entity data type, or to instances of more than one entity data type. One application of a RULE is to constrain the values of attributes that exist in different entities in a coordinated manner.[42]

EXPRESS syntax:

```
277 rule_decl = rule_head [algorithm_head] {stmt} where_clause
END_RULE ';'.
278 rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref }
')' ';'.
163 algorithm_head = {declaration} [constant_decl] [local_decl].
189 declaration = entity_decl | function_decl | procedure_decl |
type_decl.
```

Mapping:
Every rule constraining an EXPRESS entity corresponds to two tag values. The first tag is "rule" followed by the name of the rule as value. Secondly, the name of the rule as tag followed by the whole rule as string (uninterpreted in the original EXPRESS format).

Example I:

EXPRESS

```
RULE point_match FOR (point);
LOCAL
   first_oct,
   seventh_oct : SET OF POINT := [];
END_LOCAL
   first_oct := QUERY(temp <* point | (temp.x > 0) AND
                                      (temp.y > 0) AND
                                      (temp.z > 0));
   seventh_oct := QUERY(temp <* point | (temp.x < 0) AND
                                        (temp.y < 0) AND
                                        (temp.z < 0));
WHERE
   SIZEOF(first_oct) = SIZEOF(seventh_oct);
END_RULE;
```

[42] Cf. [ISO EXPRESS RM 94] 63

UOL

```
class point
   stereotyped with express_entity
   with tag values (<rule,point_match>,
      <point_match,' LOCAL
   first_oct,
   seventh_oct : SET OF POINT := [];
END_LOCAL
   first_oct := QUERY(temp <* point | (temp.x > 0) AND
                                      (temp.y > 0) AND
                                      (temp.z > 0));
   seventh_oct := QUERY(temp <* point | (temp.x < 0) AND
                                        (temp.y < 0) AND
                                        (temp.z < 0));
WHERE
   SIZEOF(first_oct) = SIZEOF(seventh_oct);'>)
end
```

Example II:

EXPRESS

```
ENTITY b;
   a1 : c;
   a2 : d;
   a3 : f;
UNIQUE
   ur1: a1, a2;
END_ENTITY;

RULE vu FOR (b);
   ENTITY temp;
      a1 : c;
      a2 : d;
   END_ENTITY;
LOCAL
   s : SET OF temp := [];
END_LOCAL;
REPEAT i := 1 TO SIZEOF(b);
   s := s + temp(b[i].a1, b[i].a2);
END_REPEAT;
WHERE
   wr : VALUE_UNIQUE(s);
END_RULE;
```

UOL

```
class b
   stereotyped with express_entity
   with tag values (<rule,vu>,
      <vu,' ENTITY temp;
         a1 : c;
         a2 : d;
      END_ENTITY;
      LOCAL
         s : SET OF temp := [];
      END_LOCAL;
      REPEAT i := 1 TO SIZEOF(b);
         s := s + temp(b[i].a1, b[i].a2);
      END_REPEAT;
      WHERE
         wr : VALUE_UNIQUE(s);'>)
   feature {any}
      a1 : c with tag values (<express_unique,ur1>);
      a2 : d with tag values (<express_unique,ur1>);
      a3 : f
   end
end
```

Implicit declaration

EXPRESS semantics:
Within a rule each population is implicitly declared to be a local variable which contains the set of all instances of the named entity in the domain; i.e., the set of entity instances governed by the rule.[43]

EXPRESS syntax:
```
254 population = entity_ref.
```

Mapping:
Since the declaration is implicit, it remains even implicitly within the UOL source.

[43] Cf. [ISO EXPRESS RM 94] 65

5.3.3 Interface specification

EXPRESS semantics:
This clause specifies the constructs, which enable items declared in one schema to be visible in another. There are two interface specifications (USE and REFERNCE), both of which enable item visibility. The USE specification allows items declared in one schema to be independently instantiated in the schema specifying the USE construct.

An entity instance is independent if it does not play the rule described by an attribute of any other entity instance, i.e., ROLESOF for an independent entity instance will return an empty set. An entity data type, which was either declared locally within or USE'd by the schema may be instantiated independently or play the role described by an attribute of an entity within the schema.[44]

EXPRESS syntax:
```
228 interface_specification = refernce_clause | use_clause.
```

5.3.3.1 Use interface specification

EXPRESS semantics:
An entity data type or defined data type declared in a foreign schema is made usable by way of a USE specification. The USE specification gives the name of the foreign schema and optionally the names of entity data types declared therein. If there are no `named_types` specified, all of the named types declared within or USE'd by the foreign schema are treated as if declared locally.[45]

EXPRESS syntax:
```
313 use_clause = USE FROM schema_ref [ '(' named_type_or_rename {
',' named_type_or_rename } ')' ] ';'.
246 named_type_or_rename = named_types [ AS ( entity_id | type_id
) ].
```

Mapping:
The use-clause corresponds to a dependency between the packages built from the EXPRESS schemas. The dependency link has the using package (the dependant element) as source and the used one (the independent element) as destination. The relation is stereotyped with *uses*.

The renamed elements correspond to tag values. Using *expresss_rename* as tag and the name in the source schema followed by the name in the destination schema comma separated as string (e.g., express_rename,'src_name,dst_name').

Since each UOL relation must have a unique name the transformation implementing tool automatically generates one. This name is invisible to the user, and lost if the UOL format is re-transferred into other formats.

[44] Cf. [ISO EXPRESS RM 94] 76
[45] Cf. [ISO EXPRESS RM 94] 77

Example:

EXPRESS

```
SCHEMA src;
   ENTITY a1;
   END_ENTITY;
END_SCHEMA;

SCHEMA dst;
   USE FROM src (a1 as a2);
END_SCHEMA;
```

UOL

```
package src is
   class src
      stereotyped with express_schema
   end
   class a1
      stereotyped with express_entity
   end
end

package dst
   class dst
      stereotyped with express_schema
   end
end

relation ?001
   stereotyped with uses
   with tag values (<a1,a2>)
   link dst to src
end
```

5.3.3.2 Reference interface specification

EXPRESS semantics:
A REFERENCE specification enables the following EXPRESS items, declared in a foreign schema, to be visible in the current schema:

- Constant;
- Entity;
- Function;
- Procedure;
- Type.

The REFERENCE specification gives the name of the foreign schema, and optionally the names of EXPRESS items declared therein. If there are no names specified, all the EXPRESS items declared in or USE'd by the foreign schema are visible within the current schema.[46]

EXPRESS syntax:

```
267 reference_clause = REFERENCE FROM schema_ref [ '(' 
resource_or_rename { ',' resource_or_rename } ')' ] ';' .
274 reource_or_rename = resource_ref [ AS rename_id ].
275 resource_ref = constant_ref | entity_ref | function_ref |
procedure_ref | type_ref.
270 rename_id = constant_id | entity_id | function_id |
procedure_id | type_id.
```

Mapping:
The reference-clause corresponds to a dependency between the packages built from the EXPRESS schemas. The dependency link has the referencing package (the dependant element) as source and the referenced one (the independent element) as destination. The relation is stereotyped with *references*.

The renamed elements correspond to tag values. Using *expresss_rename* as tag and the name in the source schema followed by the name in the destination schema comma separated as string (e.g., express_rename,'src_name,dst_name').

Since each UOL relation must have a unique name the transformation implementing tool automatically generates one. This name is invisible to the user, and lost if the UOL format is re-transferred into other formats.

[46] Cf. [ISO EXPRESS RM 94] 88

5.3.3.3 The interaction of use and reference

Note: The distinction between USE and REFERENCE
The USE and REFERENCE statements both enable the import of definitions from another schema. The definitions that are mentioned in the USE statements become *first-class* definitions within the importing schema. That is, in an instantiation of the model, these items may have an independent existence – instances can occur which are not utilized as attribute values of their items. Definitions that are imported via a REFERENCE statement are *second-class*. That is, instances can only occur when required as attribute values.[47]

EXPRESS semantics:
If an entity data type or defined data type is both USE'd and REFERENCE'd into the current schema, the USE specification takes precedence.

When a named data type is USE'd into the current schema, that named data type may be USE'd or REFERENCE'd from the current schema by another schema (i.e., USE specifications may be chained between schemas).[48]

Mapping:
This transitivity semantics is preserved within UOL for the dependencies connecting packages and stereotyped either with *uses* or *references*.

Example:

EXPRESS

```
SCHEMA s1;
   ENTITY e1;
   END_ENTITY;
END_SCHEMA;

SCHEMA s2;
USE FROM s1 (e1 AS e2);
END_SCHEMA;

SCHEMA s3;                      SCHEMA s3;
USE FROM s1 (e1 AS e2);         USE FROM s2 (e2);
END_SCHEMA;                     END_SCHEMA;
```

[47] Cf. [Schenck, Wilson 94] 63
[48] Cf. [ISO EXPRESS RM 94] 78

UOL

```
package s1 is
   class s1
      stereotyped with express_schema
   end
   class e1
   end
end

package s2 is
   class s2
      stereotyped with express_schema
   end
end

package s3 is
   class s3
       stereotyped with express_schema
   end
end

relation ?001
   stereotyped with uses
   with tag values (<express_rename,'e1,e2'>)
   link s2 to s1
end

relation ?002
   stereotyped with uses
   with tag values (<express_rename,'e1,e2'>)
   link s3 to s1
end

relation ?002
   stereotyped with uses
   with tag values (<express_uses,e2>)
   link s3 to s2
end
```

5.3.4 Expression

EXPRESS semantics:
Expressions are combinations of operators, operands and function calls, which are evaluated to produce a value.[49]

EXPRESS syntax:
```
204 expression = simple_expression [ rel_op_extended
simple_expression ].
269 rel_op_extended = rel_op | IN | LIKE.
268 rel_op = ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘<>’ | ‘=’ | ‘:<>:’ |
‘:=:’.
287 simple_expression = term { add_like_op term }.
303 term = factor { multiplication_like_op factor }.
205 factor = simple_factor [ ‘**’ simple_factor ].
288 simple_factor = aggregate_initializer | entity_constructor |
enumeration_reference | inverval | query_expression | ( [ unary_op
] ( ‘(‘ expression ‘)’ | primary ) ) .
308 unary_op = ‘+’ | ‘-‘ | NOT.
256 primary = literal | (qualifiable_factor { qualifier } ).
244 multiplication_like_op = ‘*’ | ‘/’ | DIV | MOD | AND | ‘||’.
158 add_like_op = ‘+’ | ‘-’ | OR | XOR.
```

Mapping:
All expression within the EXPRESS source remain uninterpreted and correspond to strings within the individual use.

5.3.5 Executable statements

EXPRESS semantics:
Executable statements define the actions of functions, procedures and rules. These statements act only on variables local to a FUNCTION, PROCEDURE or RULE. They are used to define the logic and actions required to support the definition of constraints, i.e., WHERE clauses and RULEs. These statements do not affect the entity instances within the domain.[50]

EXPRESS syntax:
```
291 stmt = alias_stmt | assignment_stmt | case_stmt |
compound_stmt | escape_stmt | if_stmt | null_stmt|
procedure_call_stmt | repeat_stmt | return_stmt | skip_stmt.
```

Mapping:
Since the executable statements occur only within functions, procedures and rules they are mapped within the sections concerning functions, procedures and rules.

In general they remain uninterpreted and correspond to strings within UOL.

[49] Cf. [ISO EXPRESS RM 94] 81
[50] Cf. [ISO EXPRESS RM 94] 112

5.3.6 Built-in constants

Mapping:
Since constants appear either within algorithms or as initialization of attributes they correspond to uninterpreted strings (in the case of use within algorithms) or initialization statements.

5.3.7 Built-in functions

Mapping:
Since functions appear either within algorithms or as initialization of attributes they correspond to uninterpreted strings (in the case of use within algorithms) or initialization statements.

6 Additional Specification

6.1 Full UML Support

6.1.1 Justification

The SMIF RFP requires the proposals to submit a format not dependent on meta-model constructs. This is very reasonable considering that the SMIF is for a meta-meta-model and, therefore, it should not rely on a lower level meta-model.

However, the main users of MOF will be CASE tool builders and these will also support the UML standard. In fact some will support UML and not MOF, as is the case of Microsoft. If UOL is extended to support full UML (all its constructs) a much more efficient interchange format because having more semantics:

- more information can be represented with less volume with great gains in speed,
- information being transferred is more comprehensible and
- easier to process for CASE tools

We, therefore, propose extending UOL to support UML constructs as an optional non-mandatory second level of the SMIF standard.

The support for UML constructs allows for an easier and more compact representation of UML models. Without the extensions all the UML concepts that are not MOF concepts must be added to the UOL code cluttering the representation with the description of the meta-model.

```
usecase CloseObject
       actor
            Writer
       is
            '(a)The system (G) will load the current object that is
referenced
            (b) update the current closing date (Date) of the document
            (c) ask to the actor Writer) for his username update the
username in the document finally,
            (d) save the document'

       extension in ”(a)”,”(b)”,”(c)”,”(d)”
end – CloseObject
```

In the following example we describe the former use case without specific constructs. It is incomplete for clarity reasons and shows a first part of description of the meta-model using MOF and a second part of description of the use case.

```
-- some classes omitted
class Classifier
       with tag values (<Metamodel>)
       inherit GeneralizableElement
       -- rest of body omitted
end

class Actor
       with tag values (<Metamodel>)
       inherit Classifier
       -- rest of body omitted
end

class Usecase
       with tag values (<Metamodel>)
       inherit Classifier
       -- rest of body omitted
end

Writer instance of Actor
       is
       name : Writer
end

CloseObject instance of Usecase
       is
       annotation :
            '(a) The system will load the current object that is
referenced
            (b) ask to the actor for its username update the username
in the document
            (c), finally save the document
            (d)';
       name : CloseObject;
       extension_point: <<'a','b','c','d'>>
end
```

Of course, even if the specific constructs are not used it is not necessary to include the meta-model as part of the transmission each time. The meta-model can be appended through an 'import' sentence, in which case the package must be available for the receiver, or declared with a tag known by the receiver.

With this extension there are also additional benefits, allowing UOL:

- to become a Universal Round-Trip Engineering Language
- to be a textual representation of UML
- to be an alternative representation to graphics to describe analysis and design models for visually impaired individuals

The UOL has been designed in such a way that all UML constructs are optional and only the MOF support is mandatory.

6.1.2 Mapping between UOL and UML with UML constructs

In order to describe a UML diagram with UOL, several new elements, stereotypes, tag values and constraints must be defined. These new elements and their mapping are the following:

UML 1.1	UOL 1.2
Action	Action
Action	Action
Action sequence	Action sequence
ActionSequence	ActionList
ActionState	State (with an Action associated in context of an Activity diagram)
Activity model	Activity model
ActivityState	State (in context of an Activity diagram)
Actor	Actor
Argument	EntityDeclaration
Association	Relationship
Association class	Association class
AssociationEnd	Feature clause of the relationship
Attribute	Attribute
AttributeLink	N/A
Binding	Actual generics
CallAction	Action (used with call clause)
CallEvent	Event (with a call clause)
ChangeEvent	Event
Class	Class
Collaboration	Collaboration
Comment	Comment
Component	Component
Composite state	Composite State
Constraint	Constraint (splitted in declaration and use)
CreateAction	Action (used with create clause)
Data type	Class (with stereotype)
DataValue	Expression
Dependency	Dependency (relationship with a direction link)
DestroyAction	Action (used with TextMultiline)
Element Ownership	Element Ownership
Element reference	Import clause
Event	Event
Event	Event
Exception	Exception
Generalization	Generalization (inherit clause)
Guard	Guard
Instance	Instance
Interface	Class (with stereotype)
Link	AssociationEnd (value in an instance of a Relation)
LinkEnd	AssociationEnd (with values)
LocalInvocation	Expression

MessageInstance	Instance
Method	Method
Model	Model
Node	Node
Object	Instance
ObjectFlowState	State (that flows an object associated in context of an Activity diagram)
Operation	Operation
Package	Package
Parameter	Parameter
Partition	Partition
Pseudostate	Pseudostate
Refinement	Dependency (with stereotype)
Request	Operation/Signal
ReturnAction	Action (used with TextMultiline)
SendAction	Action (used with TextMultiline)
Signal	Signal
SignalEvent	Event
Simple state	State
State machine	State machine
Stereotype	Stereotype
Submachine state	Submachine
Subsystem	Subsystem
Tagged value	Tagged value
Template	Formal generics
TerminateAction	Action (used with TextMultiline)
TimeEvent	Event (with a TimeExpression)
Trace	Dependency (with stereotype)
Transition	Transition
UninterpretedAction	Action (used with TextMultiline)
Usage	Dependency (with stereotype)
Usecase	Usecase
Usecase instance	Usecase instance
View Element	Diagram and viewed with clause

The abstract UML concepts do not have a specific UOL construct. Instead the information provided by such elements is included in the non abstract heir's specific construct.

The package UOL_UML package will be automatically loaded and it will contain all the standard stereotypes, tag values, and constraints defined in the document ad/970805.

6.1.3 Benefits of UOL with UML constructs

6.1.3.1 UOL as a Round-Trip Engineering Language

6.1.3.1.1 Justification

Summarizing the need of a round-trip justification language and the advantages of having UOL as the required round-trip engineering language we have seen in chapters 2.4 and 2.5:

- CASE tool builders can use it as a substitute for their proprietary incomplete (it is not well adapted to all OO languages) “mark-up language” that they are presently using. They usually have different versions of the mark-up code for different languages.

- Using only one mark-up language for all programming languages reduces over 80% of the cost of developing round-trip tools as we show in chapter 6, if round-trip engineering tools are split in two parsers: a front-end (common to all source languages) and a back-end (specific for each language) using, what we call, collaborative compilation.

- It may, also, be used by companies developing GUI builders, component libraries, etc. to allow the source code, and it's corresponding OOAD model embedded in the code, generated by their products to be easily imported into any model by CASE tools supporting UML.

- The programmer can easily change the code and the model during testing and debugging, due to the simplicity of the language and the easiness of learning it, avoiding on necessary guessing by the round-trip tool.

- It allows reverse engineering of non-CASE-tool-generated OO code to be imported correctly if it is enriched with UOL code. This can be done manually or with a software product that acts as a Wizard, that can learn from previous experience asking to the programmer questions of the sort of "Is this declaration an attribute or does it represent an association?" "Is this pointer a shared aggregation or simply an association?" etc.

- Makes interoperability, both for the model and the code, between CASE tools immediate and with more semantics than interchange formats allowing for more control during transfer between tools.

6.1.3.1.2 Modular Structure and Flow of a Round-Trip Engineering Tool

As we can see in the Figure 6.1 we describe the structure of the reverse engineering task modules.

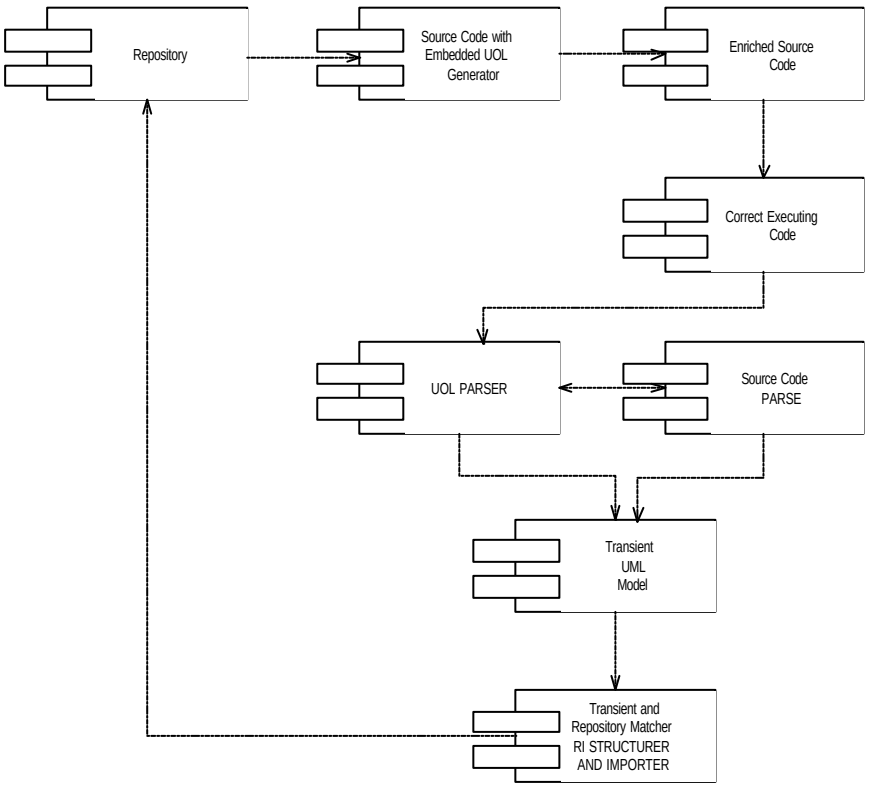


Figure 6.1

As can be seen in the diagram, once the code has been tested and debugged, re-importing requires three modules.

The first two are parsers of UOL and the target source language. They are responsible for parsing the source code and building, in transient memory, an OO model. We call this process collaborative compilation, because each parser collaborates with the other in building the OO model from the input. The UOL parser (if we have chosen to generate full UOL in the source language) starts this process. This module starts processing UOL sentences and building in memory an equivalent to the repository OO model. Once it detects input in the target source language it passes control to the second parser which does the same with the language sentences it is capable of parsing until it detects UOL code at which moment it returns control to the UOL parser. This process is repeated until the complete program has been processed.

The third is the repository and transient model matcher. This module will compare the transient model built by the parsers and update the repository. During this process, if the CASE tool supports version control of the OOAD model, the differences between the repository version and the current transient version will be stored for future use.

Since one of the most arduous and difficult tasks is checking the consistency of the program, it is of the most interest to use a meta-model with the maximum integrity constraints embedded within. This permits delegating to the meta-model the task of checking the consistency of the program instead of doing it the parsers (especially the target source language parser).

If we design the round-trip tool with this structure, we can easily see that both, the UOL parser and the matcher/importer, are target language independent and, therefore, reused for all languages. One last requirement to simplify the target language parser and make most of the tool language independent consists in embedding, in our meta-model, the integrity and/or consistency constraints of UML. If this is done we will relieve, the target source parser of any semantic/consistency analysis checking and we will be in a position of being able to build round-trip engineering tools for any language with only a very simple source target language parser.
Once we have built the UOL parser, the constraint-checking meta-model and the repository-transient model matcher and importer we will be able to produce round-trip engineering tools for any language with a reduction of over 80% of the efforts that have been necessary until now. This will allow CASE tool builders to offer round-trip engineering for all the languages they choose to support, instead of the few they are now offering and accessing larger markets than they presently can.

Reverse engineering with UOL is not a single pass process or scanning, although this does not imply a loss of efficiency. OO programming languages may have different structures physically, C++ has .h and .cpp (where we can define more than one class), Eiffel has only .e entries each representing one class, etc. This implies that a UML model when translated to code can be distributed in different ways depending on the target language. To cope with this, we rely on a MDL (Model Description Language à la PDL of Eiffel) and also on building the transient model iteratively.

This iterative process will consist in incrementally instantiate model elements and/or enrich them from smaller or larger pieces described in each target language file. When processing is complete we will have a complete model in transient memory that can be matched with the repository model. This iterative and incremental process is by no means inefficient because we process the source input to build our model only once except for the same syntactic and semantic processing that may be multi- pass as in all compilers.

One final comment on the two alternatives of generating UOL code. As we have mentioned previously, when the CASE tool generates code it has two alternatives. The first consists in generating the full UOL code for the model. This implies a redundancy with the target language, because all the OOAD constructs that can be expressed in the target language are written twice. In this case, the UOL parser will have the initiative and it will be its responsibility to call the target language parser. If we choose to generate only the minimum UOL code necessary, then, control will depend on both parsers depending on what type of sentences are being read at each moment.

Since one of the important objectives is to allow the programmer to maintain both the target language code and the UOL code, we prefer generating always full-UOL because it will make the program more readable.

In the Figure 6.2 we show the process that is involved in round-trip engineering with UOL.

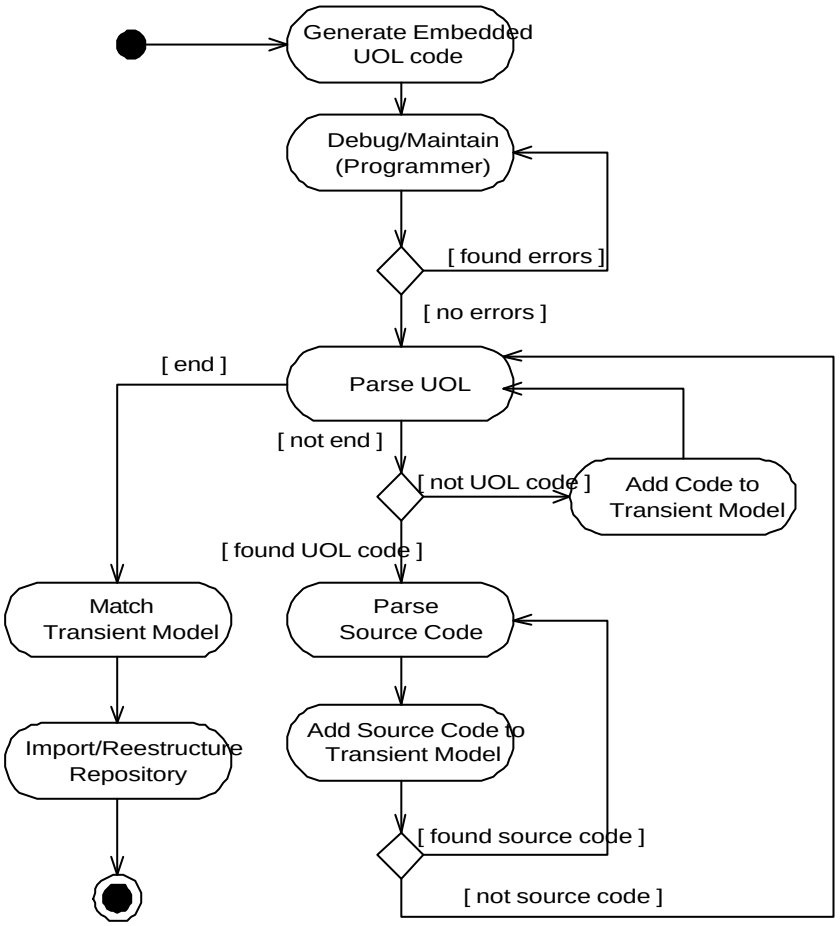


Figure 6.2

Besides syntactic and semantic analysis, there are important integrity and consistency checking that the parsing process can do. UML is a very complete and large language and it is easy, that in the process of maintaining a program, a programmer can make errors. Some of the constraints of the meta-model (ex. services used in sequence diagrams and not developed in the objects because of changes, etc.) may not be respected as we have mentioned previously and it is necessary to detect them before importing. Before we reimport the models reengineered and update the repository, it is important to check the full consistency of the transient model. In the process of compiling the program, the tool can inform the programmers of errors they have made, allowing them to correct the program before updating the repository. In the same way, the programmers may not have been aware of the implications of some of their changes with respect to the previous version or the other parts of the system developed by others. The matching process will allow detection of these errors or inconsistencies and warn the programmers so that they might take action.

6.1.3.2 UOL as a Textual UML

The OOA&D Task Force put in the road map the need of defining a textual UML. There are many reasons for this need. We have already given several reasons for this previously. There are reasons both from the point of view of the need of humans reading and processing textually the models as well as for compilers, code generators, metric evaluators, etc. that normally require working with text.

Many universities researching on formal methods and several companies working on critical systems have started adding formal specifications to UML. These users need a textual language to process for specification validators, proof generators, theorem demonstrators, code generation, etc.

UOL with full UML support is a complete textual representation of UML and completely adequate for these requirements.

6.1.3.3 UOL as an alternative to graphics for visually impaired individuals

Another milestone that the OOA&D road map has is developing an alternative to UML's graphics for visually impaired software engineers.

Object orientation is possibly the technology that requires more intensively the use of CASE tools. A software engineer using OO will have to handle hundreds, if not thousands, of classes and components. The only feasible way to control this volume of information is using CASE tools that allows us to organize our designs, find components to reuse, etc.

Visually impaired software engineers have been severely limited in using CASE tools because of their graphical interface. This limitation will therefore be worse in object orientation and possibly limiting many their professional activity to programming tasks instead of analysis and design.

The only valid alternative to graphics for visually impaired individuals is a textual full life-cycle language with all analysis and design constructs. UOL, being a full textual representation of UML, completely satisfies this requirement.

7 References

[RC93] Tom Atwood, Douglas Barry, Joshua Duhl, Jeff Eastman, Guy Ferran, David Jordan, Mary Loomis, Drew Wade, " The object database standard ODMG-93 release 1.2", edited by R.G.G. Cattell,Morgan Kaufmann, 1996

[Bock/Odell97a] Conrad Bock, James Odell, "A more complete model of relations and their implementation", Journal of Object-Oriented Programming, June 1997, Vol. 10, No. 3

[Bock/Odell97b] Conrad Bock, James Odell, "A more complete model of relations and their implementation: mappings", Journal of Object-Oriented Programming, October 1997, Vol. 10, No. 6

[DECexpress 92] Digital Equipment Corp.:
DECexpress – EXPRESS Language Reference Manual, Order Number: AA-NKWA-TE, Digital Equipment Corp., Maynard (USA), 1992

[EIA/CDIF97] EIA/CDIF Technical Committee, "The UML meta-model and the CDIF Transfer Format", June 19, 1997

[Gray et al92] Peter M.D. Gray, Krishnarao G. Kulkarni, Norman W. Paton, "Object-Oriented databases, A semantic approach", C.A.R Hoares series, Prentice-Hall, 1992

[ISO EXPRESS RM 94] International Organization for Standardization (eds.):
Industrial automation systems and integration – Product data representation and exchange – Part 11: Description methods: The EXPRESS language reference manual Reference number: ISO 10303-11:1994(E), International Organization for Standardization, Geneva (CH), 1994

[Meyer92] Bertrand Meyer, "Eiffel: The Language", Prentice-Hall, 1992

[MOF97] Unisys et al., "Meta Object Facility (MOF) Specification", 1 September 1997

[OCLv1.197] Rational et al., "Object Constraint Language Specification" version 1.1 ,1 September 1997

[Odella] Odell, James; "Six different kinds of composition", Journal of Object-Oriented Programming, Vol. X, No. 5

[Odellb] Odell, James; "A foundation for composition", Journal of Object-Oriented Programming, Vol. X, No. 7

[OML96] Donald Firesmith, Brian Henderson-Sellers, Ian graham, Meilir Page-jones; "OPEN Modeling Language (OML) Reference Manual" Version 1.0 8 December 1996

[Peralta97] Alonso J. Peralta, “UOL: A Full Life-Cycle Object-Oriented Software Development Language”, 1995 draft (complete specification); Ph.D. Thesis, 1998

[Schenck, Wilson 94] Schenck, D., Wilson, P.: Information Modeling the EXPRESS Way, Oxford University Press, New York (USA), 1994

[SMIF RFP 97] Object Management Group (eds.): Stream-based Model Interchange Format – Request for proposal, OMG-Document: ad/97-12-03, Object Management Group, Farmingham, MA (USA), 1997

[Tanzer95] Christian Tanzer, ”Remarks on object-oriented modeling of associations”, Journal of Object-Oriented Programming, February 1995, Vol. 7, No 9

[UMLv0.996] Rational “The Unified Modeling Language for Object-Oriented Development (Version 0.9)”,1996

[UMLv1.097] Rational “The Unified Modeling Language for Object-Oriented Development (Version 1.0)”, January 1997

[UMLv1.197] Rational et al.“UML Semantics (Version 1.1)”, 1 September 1997

[UML Notation Guide 1.1 97] Rational Software Corp., et al.: Unified Modeling Language 1.1 - Notation guide, Document: ad/97-08-05, Rational Software Corp., Santa Clara (USA), 1997