OMG Unified Modeling Language Specification (revised submission)

3C

Clear Clean Concise

Submitters:

Financial Systems Architects
MEGA International
TogetherSoft

Supporters:

Hitachi
Mercury ComputerSystems
OpenIT

Cap Gemini Ernst & Young

Version 2.0.14 short
Revised Infrastructure Submission, 9 September 2002

.

"I gave desperate warnings against
the obscurity, the complexity, and
the over-ambition of the new design
but my warnings went unheeded.

I conclude that there are two ways of
constructing a design:

One way is to

make it so simple that
there are obviously no deficiencies

and the other way is to

make it so complicated that
there are no obvious deficiencies."

Tony Hoare
Turing Award Lecture, 1980

The most general concepts of engineering might be

system, behavior, and interconnection,

formalized in such a way as to include
hierarchical whole-part relationships.

Joseph Goguen

"Buffering the shock of invention and
giving the 'clearest' presentation are not
necessarily the same thing:

the type of 'clarity' we seem
to be homing in on

does not exclude
surprises for the reader."

Edsger Dijkstra
Manuscript EWD-788

This document is formatted for two sided printing. Readers of the soft copy may find the occasional blank page. We have left them in for those who wish to print the document. We apologize to readers of the soft copy for the layout, which may distract, as the body of the text jumps from right to left on alternate pages.

The most recent revisions of the submissions are always available at:
www.community-ML.org/InfraWorkingDraft.pdf
www.community-ML.org/SuperWorkingDraft.pdf

Section 9.4 of Part II is a slightly revised version of the ODP approach to conformance text.
community-ml.org/RM-ODP/Part2/15.html

Part 0

The authors of this document are William Frank, Guy Genilloud, Haim Kilov, Hernan Astudillo, Alexey Solofnenko and Joaquin Miller, of Financial Systems Architects, Antoine Lonjon, of Mega International, Karl Frank, of TogetherSoft and Makoto Oya and Akira Tanaka, of Hitachi.
The process and data flow concepts are based on the work of Jeff Smith, of Mercury Computer Systems, Alan Moore, of Artisan Software, Kenneth Baclawski and Mieczyslaw Kokar, of Northeastern University, and Boris Gaber, of Thomson Financial Research.

We express our appreciation to Tony Clark, Steve Cook, Andy Evans, Jose Luiz Fiadeiro, Robert France, Daniel Jackson, Stuart Kent, Steve Mellor, Bernhard Rumpe and Perdita Stevens for their kind assistance. They may not agree with all we say here, but we benefited greatly from their help. We thank Jim Rumbaugh for reminding us of the great writing of Frank Herbert.

We acknowledge our debt to the work of the original UML partners, the UML RTFs and the many authors, too many to list, who have written on modeling, modeling languages and UML. We have benefited greatly from the workers of the Precise UML group.

Our submission would not be possible without the work of the mathematical linguists, philosophers of science and of language, mathematicians and computer scientists of the last century.

These include the giants Gottlob Frege, Bertrand Russel, David Hilbert, Rudolf Carnap, and Alfred Tarski, as well as the many distinguished participants in their tradition, such as C.A.R Hoare, E.W. Dijkstra, William Kent, John Sowa, Mario Bunge, Paul Grice, Hillary Putnam, Hao Wang, Saul Kripke, John Corcoran, Ed Keenan, George Lakoff, and Barbara Hall Partee.

Indeed, the need for our engineering discipline to build from the scientific foundation laid in the last 100 years is a major motivation of this submission.

Our own appeal to the clear thinking of scientists and mathematicians was forced on us by the hard questions that some of our business clients ask:

What is not crystal clear to them, they doubt.

Finally, and most of all we thank all those who are participating in the communityUML process.

The UML 2.0 Infrastructure RFP solicits proposals to:

- Restructure the architecture of the language so that it is easier to understand, implement and extend while preserving the [meaning] of the language.
- Provide first class extension mechanisms and profiles that are consistent with the metamodel architecture as we understand it.
- Improve the architectural alignment with other OMG modeling standards, such as the MOF and XMI.

In order to deliver what the RFP asks for, we have used, in the specification of the infrastructure, a single language for expressing all modelling concepts. The UML language of this submission can be expressed exactly in any natural language and can be expressed in any sufficiently rich formal language. Parts of this language can then be mapped as desired to various types of diagrams.

We have done our best to do all three.

This language is based on the foundations of mathematical logic as systematized by Tarski. We have used Tarski's theory of definition as the extension mechanism.

We have also tried to:

- Improve the architectural alignment with the CORBA Object Model.
- Improve the architectural alignment with other modeling standards, such as EXPRESS and ODP.

The specification has been compared with MOF, XMI, the General Relationship Model and ODP.

- Enable consistency checking and model transformation by modeling tools

We have been guided by the needs of Model Driven Architecture

- Increase the ease and precision of language extension.
- Use to maximum benefit the work of the last century on models and languages.

We build on this foundation in order to increase the practicality and usefulness of the UML.

Part I

Part 1 of the submission contains material required by the RFP. This short version of the submission

The full submission document consists of the three parts requested in the RFP.

Part I contains material required by the RFP. This shorter version of the submission omits most of Part I.

Part II includes twelve sections:

1 Overview
2 General remarks
3 Meaning and semantics
4 Structure of the language
5 The UML kernel
6 Model management
7 MOF extensions
8 Pattern application extensions
9 Conformance
10 Notation
11 Differences between UML 1 and 2

After Part III there are two appendices:

12 UML 1 in UML 2.0
13 Alternative Vocabularies

A list of references completes the submission:

14 References

The full submission document is available at:

www.cuml.org

Overall design rationale

The models and terminology of the CORBA Core specification and the Object Management Architecture Guide have been used in this submission.

Our design goals are utility, practicality, simplicity, clarity, solidity and exactification.

"Restructure the architecture of the language so that it is easier to understand, implement and extend while preserving the [meaning] of the language." [RFP]

Utility

Do not remove anything useful from UML. Add nothing that will not be useful to most modelers. Where feasible, add things that modelers will find very useful.

Practicality

The activity of modelling systems is a human endeavor directed to practical purposes. Make no room in the modeling language for anything that is not practical.

Simplicity

Make the language as simple as possible,

but no simpler.

Clarity

UML wants a single, standard approach to establishing the meaning of its terminology. Focus on the need of software designers and developers to have a clear, intuitive understanding of the language.

Make it clear which concepts are primitive. Provide a brief explication of each. Define all the other concepts, exactly.

At the same time, if the approach chosen also enables a formal semantics, experts can use that in order to establish the consistency and precision of the language.

Solidity

Place the UML squarely on the solid foundations built by mathematicians and other scholars during the last century.

Exactification

Replace any concept of UML 1 that is imprecise with another concept that has a precise meaning, which has a sizeable overlap with the meaning of the original concept.

Summary of our design rationale

We did not take the road of looking for the foundations for UML in logic and mathematics, in order to introduce new areas for specialist and expert argument.

Instead, we did this because there is already in place a solid foundation for our work, and using this foundation brings clarity, simplicity and practical utility.

"*It is wise to have decisions of great moment monitored by generalists. Experts and specialists lead you quickly into chaos.*"

— *Frank Herbert*

We offer this submission particularly to the generalists and users of UML among the voters on the UML 2 adoption.

The voters will make a decision of great moment.

Uses of UML 2.0

The UML 2 modeling language of this submission is designed specifically to be used in three different ways. To:

- specify or describe a software system (including its hardware and network)
- describe or specify the environment in which a system operates (including the people and social systems in that environment)
- specify a modeling language (including the MOF language and UML 2)

Since UML 2 is designed to describe the environment of a system, it is also well designed to describe that same environment without reference to any automated information system.

Part II

The text of this submission is in the other column. This column has commentary.

1 Overview of UML 2.0

This Infrastructure submission prescribes a kernel language for UML 2.

There is also a detailed discussion of conformance, since the meaning of a model is in its interpretation in terms of some actual system or systems.

1.1 Kernel

Some readers may be surprised at how small the kernel is, as we were a bit surprised when we carefully studied UML 1.4 to determine the actual size of the kernel.

As specified in the Infrastructure RFP, 'kernel' refers to those language elements that are used as the foundation for defining other language elements. This submission therefore limits itself to those UML 2 concepts that are required to specify UML 2, using UML 2 as the specification language.

Our intent is to keep the kernel as small as will suit the definition of 'kernel' in the RFP. And, at the same time, to keep it as general as is practical. This allows maximum freedom in designing the superstructure and other extensions.

To include in the kernel anything not required to specify UML 2 is to restrict the family of languages that can be based on the kernel. There is no authority in the Infrastructure RFP for any such restriction.

As a confirmation of the suitability of this kernel, we will provide in a separate document a specification, using the UML 2 kernel, of all the UML 1 concepts used in the UML 1 metamodel. This submission includes examples.

1.2 Extensibility

1.2.1 Extensibility within UML 2

Extensibility is by extension specifications, which precede a model and specify extensions to the language.
See [SCook]

The extensibility mechanism of UML 2 is the same as the language itself. Because of this, the distinction between adding to a particular model and extending the modeling language is this:

o An extension to the modeling language is identified to a modeling tool as such. It is given a name so that the extension may be included by the tool as an extension specification for other models.

o An addition to a model is identified to a modeling tool as just that, an addition to that model

o An addition to a model is made by:

- — adding a model element and statements using that model element (invariants or a definition) that specifies the way in which the model element can be used, or
- — by adding other statements.

o The language is extended by:

- adding a new sort of model element and statements about that sort of model element (invariants and definitions) that specify the way in which that sort of element can be used, or
- adding a statement that restricts the way in which an existing model element (or elements) may be used.

A modeling tool that complies with the Full compliance point will enable extension within the UML 2 language.

1.2.2 Extensibility using the MOF

A language designer or user extending UML 2 may store the specification of the extension in a MetaObject Facility. A modeling tool may use that specification to enable a user to model with the extension.

A modeling tool that complies with the MOF compliance point will use a MOF compliant metadata repository to store the specification of the modeling language or to obtain an extension specification, and will enable a user to model with the extension.

1.3 Compatibility

1.3.1 Backward compatibility

The UML 2 kernel is designed to ensure backward compatibility with UML 1.4. That is, UML 1.4 models will be well-formed models of an extension of the UML 2 kernel.

Compatibility is achieved by producing a kernel that is minimal and general, so that both UML 1.4 and an improved UML 2.0 can be specified as an extension of the same kernel. In this way, all UML 2.0 compliant tools will automatically support UML 1.4.

In addition, any changes to the meaning of UML 1.4 model concepts are specified in this submission, exactly. However, where we have not been able to determine the meaning of a UML 1.4 model element, we are not able to say if there has been a change in that meaning, nor, if so, what the change is. In any case, we specify the meaning of each UML 2 model element, exactly.

As mentioned, we will provide, in a separate document, a specification, using the UML 2 kernel, of all of the UML 1.4 concepts used in the UML 1.4 metamodel, specified as a standard extension of the UML 2 kernel.

This will provide full backward compatibility.

1.3.2 MOF compatibility

Our design rationale is that there can be no practical reason for any difference between the UML kernel and the core of the MOF model.

We propose that the UML be a superset of the MOF language; or, if the reader prefers, the MOF language be a subset of the UML. We are working with MOF submitters to specify a MOF 2 Core submission as a standard extension of the UML 2 kernel. The required concepts are provided here as MOF Extensions.

Whatever the resolution of the differences between the UML 1.4 and the MOF 1.3 or MOF 1.4 languages, we expect that the MOF 2.0 Core language will be a subset of the submitted UML 2.

After review of the revised MOF submissions, we will prepare a document that will prescribe which of the statements specifying the UML 2 language are to be interpreted by a MOF compliant meta-data server as immediate structural consistency checks.

1.4 Metamodel

Having used UML 2 to specify UML 2, we have built a model of the language, or UML metamodel.

Drawings of portions of the metamodel are included as illustrations.

This submission does not include a complete drawingof this model, using UML 1 or UML 2 notation; that diagram would correspond to what the UML 1 specification calls the “abstract syntax.” This diagram will be submitted as a part of UML 2 tutorial material being prepared. This drawing will not be normative. The text is the normative specification of the metamodel

One concrete syntax for a part of UML 2 is specified in the Notation section of this document. If this syntax is used, the rest of a model will be either:

- declared using OCL or a tool specific constraint language

or else

- entered into a tool via property sheets; these are not standardized.

A concrete syntax for all of UML is specified in the OCL submission.

The statements that specify UML 2 correspond to the combination of the what are called the “abstract syntax” and well-formedness rules in the UML 1 specification.

The definitions and modeling invariants specify the meaning of UML 2 models.

This metamodel is, of course and by design, expressed in terms of the MOF 2.0 model. By “the” MOF 2.0 model, we mean that subset of the UML 2 language used as the MOF Core model for MOF 2.0.

This is based on our feeling that the time for complete conformity of UML to the MOF specification is at the time of the adoption of the 2.0 specifications for MOF and UML.

1.5 Notation

This submission prescribes the correspondence between the language and the graphical notation.

We propose an addition to the notation to enable drawing a interaction without decomposing it.

We also propose an addition to the notation for associations, to enable drawing an association of associations.

1.6 Toolmaker's model

This submission does not prescribe the data structure to be used by a UML 2 compliant modeling tool. However, the specification might be used in that way by a toolmaker.

1.7 Model interchange

UML 2 models are interchanged as XMI files. The structure of these files corresponds to the structure of the UML 2 specification.

We will provide in a separate document an XMI DTD for UML 2 model interchange, based on the MOF Core submission we support.

2 General remarks

2.1 The UML 2 kernel language

This is the specification of the kernel language of UML 2.

We strive to be precise in our use of natural language. To achieve this, we have also (in another document) expressed parts of this specification in a formal language, as a check on the precision of the natural language.

Every time we use a word that is the name of some element in the UML 2 language we mean that element.

2.2 System

UML is used to describe something that exists or to specify something to be built. That thing will be of interest as a whole and also as composed of parts. So it is a system. UML is for description or specification of systems.

2.3 Environment

UML will be used to specify or describe open systems: those that interact with their environment.

Of course, the system and its environment form another system. When the distinction does not matter, it is this larger system we mean in this document.

So, in order to understand an existing system, parts of the environment of that system must be described. And parts of the environment of a system to be built must be specified, in order to understand what the environment must be like for the system to work.

In a UML model, the environment of a system is everything in the model, other than that system.

2.4 Model

A model is a description or specification of a system and its environment for some certain purpose. A UML model is model elements, statements about those model elements, and drawings representing some of those elements and statements.

See the MDA documents, ormsc/01-07-01 and ormsc/02-08-01.

A model may be a viewpoint model, describing or specifying a system from a particular viewpoint.

The system may be an automated information processing system, or any other system, such as the human genome, a manufacturing plant, a communication network, a securities market, an e-commerce business or a modeling language.

2.5 ModelElement and Statement

Examples:
omgDocumentServer is an object, a kind of model element. WebServer is an object type, another kind of model element.
'omgDocumentServer is of type WebServer.' is a statement.

UML 2 models are made of basic model elements and statements. Statements are about model elements.

Some statements are definitions, defining new kinds of model elements or new individual model elements.

Many statements are statements about a particular state of a system (e.g. a snapshot) or are invariants about the system (e.g. the specification of a class).

The statements made in this specification are about all UML 2 models. The system specified by this submission is the UML 2 kernel language.

Every statement may be written as a text. Many statements may be drawn or displayed as a figure. The UML notation specifies such figures.

In the kernel language of UML 2, statements are statements in OCL 2.0.

In this submission, every OCL statement is translated into a statement in a narrow subset of English, which statement captures the same meaning as the OCL statement.

We do not expect that all modelers will rush to use only statements to specify a model, most will still prefer to use drawings and property sheets for most work,
but we do expect that basing the language on statements will have three advantages. This approach will:

- *— provide a precise specification of all of UML*
- *— enable precise interpretion of the meaning of a UML model*
- *— simplify consistency checking and language extension for tools.*

2.6 Viewpoint

A viewpoint on a system is a form of abstraction achieved using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within a system.

community-ML.org/RM-ODP/Part2/3.html#3.2.7

Viewpoints can be applied to a complete system, in which case the environment defines the context in which the system operates. Viewpoints can also be applied to individual components of an system, in which case the component's environment will include some abstraction of both the system's environment and other system components.

community-ML.org/RM-ODP/Part3/4.html#4.1.2

Applying a viewpoint results is a viewpoint specific model, called a viewpoint specification or a view of the system.

IEEE 1471—Recommended Practice for Architectural Description for Software-Intensive Systems

A view is a partial specification of a system focusing on particular concerns.

A complete specification of a system includes statements of correspondences relating one viewpoint specification to any other viewpoint specifications, showing that these views are consistent.

community-ML.org/RM-ODP/Part3/10.html#10.1

2.7 Extension specification

A extension specification to a model specifies extensions to the language. These extensions consist of statements. The statements may include definitions of new sorts of model elements.

The UML kernel plus an extension specification yields an extended UML language. In particular, the UML kernel, plus the extensions in our Superstructure submission yields the UML 2 language. Any extended language may again be extended by another extension specification.

Tools distinguish an extension specification, which extends the language, from a model, which is in that extended language. A model in an extended language is preceded by an extension specification or by a reference to an extension specification available to a tool in a MOF or in some tool specific format. That extension specification specifies the extended language in which that model is expressed. More than one extension specification may precede a model.

2.8 Model Driven Architecture

The Model Driven Architecture adopted by OMG sets expectations for models.

It requires that an MDA model be based on a language that has:

— a well defined form

— a well defined meaning

— and possibly rules of analysis, inference or proof.

So, for MDA, the UML must have a well defined meaning.

"In the MDA, a specification that [does not meet these expectations] is not a model."

MDA contemplates automatic assistance in creating a platform specific model (PSM) from a platform independent model and a platform specification.

Automatic assistance will be severely limited to the extent that the meaning of a model is not well defined.

To the extent that a language enables analysis, inference or proof, both automatic assistance and automatic checking of manual work will be greatly enhanced.

The foundations chosen for UML 2 are the strongest available for a modeling language to be used in a model driven architecture.

3 Meaning and semantics

3.1 Meaning

The meaning of the UML 2 language is given by the statements of this UML 2 language specification.

The basic elements of UML 2 are not defined. Their meaning is given by their use in the statements of the language specification, that is, the ***modeling invariants*** and the ***definitions*** of other elements. The remaining language elements are defined in terms of the basic elements.

The meaning of a model comes from practical rules for its use in describing or specifying a system.

Elements of a model are used to represent what is in the system.

When the model is a description of a system, it means: the system is like this.

When the model is a specification for a standard it means: a conforming system must be like this or a compliant standard must be like this.

When the model is a specification for a system to be built, it means: build one (or many) like this.

Or it means build something that can make things like this.

3.2 Formal semantics

A formal semantics for UML 2 will contribute to understanding and making consistent the intended meaning of the language.
It will enable a triangulation between:
- the language,
- our intuitions about the intended meaning of the language, and
- a formal specification of semantic rules for interpreting the language in a mathematical domain.

This will provide checks and balances:
when results in any one of these aspects of the language do not match the results expected in another, some adjustment must be made.

Because the UML 2 language is an order n language, there is guaranteed to be a set theoretic formal semantics. This guaranteed formal semantics will consist of a standard set theory and a semantic mapping from UML 2 to that set theory, providing an interpretation in set theory of any UML statement.

Other methods of providing a formal semantics include those of several of the OCL 2.0 submissions.

In fact, we believe that UML is best served by a third order language with a Henkin semantics; but we leave formal analysis to others.

We do not provide a formal semantics, but focus instead on providing a meaning:
a clear, clean, concise

and exact

meaning.

4 Structure of the language

4.1 Names

A ***model element name*** designates a model element. For example, an object, an action, a time, and a role all have names.

Constant Name

A ***constant name*** is the name of a single specific model element.

Variable Name

A ***variable name*** designates some unidentified model element. During the course of reasoning about the system, the model element designated by a variable name can change.

For example, the modeler might write: "Suppose o is some object bound to the role, client." Here 'o' is a variable name.

4.2 Statements

Statements mention model elements. They are built using:

the ***model vocabulary***:

— model element names — *Example: aFoo*
— predicate names — *Example: __ is even.*
— function names — *Example: color(__)*

the ***fixed vocabulary***:

— quantifiers — *Examples:* ***for every for some for no***
— connectives — *Examples:* ***not and or xor if...then means***
— the identity and equivalence symbols — *Namely:* ***is is equivalent to***
— the binding symbol — *Namely:* ***binds***
— punctuation — *Examples:* ***) (; : .***

The model vocabulary will change from one model to the next. The fixed vocabulary is common to all models.

The syntax of statements is specified and the formal semantics of statements is mentioned in the FSA OCL submission.

www.community-ML/docs/FSA_OCL.pdf

The meaning of a statement is as given by the modeling invariants and definitions in this submission and the intended meaning of the model vocabulary of the model in which that statement appears.

The meaning of statements is explained by the text and examples of this submission.

4.3 Definitions

One kind of statement is a definition.

Example: A binary association *means an association with exactly two roles for the elements associated.*

A ***definition*** consists of a statement, the ***first part***, the connective, ***means***, and another statement, the ***second part***.

The first part places what is being defined in a context; the second part defines it in that context.

Definitions

x is ***used in the definition of*** *y* means *x* appears in the second part of the definition of *y*

y is ***dependent on the definition of*** *x* means *x* is in the transitive closure of the relationship, *used in the definition of*, with respect to the constant names, function names, and predicate names in the second part of the definition of *y*

Note that recursive definitions are not circular, because they are shorthand for complex assertions in which the second part contains only a variable name for the term being defined; they depend on the uniqueness of the element that this name can represent.

The definition of *x* is ***circular*** means a constant name, function name, or predicate name in the second part of that definition is dependent on the definition of x.

Modeling Invariants

There are no circular definitions.

Every constant name, function name, or predicate name has at most one definition.

4.4 Function names

Examples: size(_) motherOf(_) sumOf(_ , _)

A function name is used to indicate a mapping between model elements.

A function represents some regular connection between things in the system (or its environment).

Every function has a certain number of places. A function name together with a model element name in each place is a model element or collection of model elements (the value of the function).

A function name, with a constant or variable name in each place, may be used in a statement, function, or predicate, where the name of a model element is expected.

Example: sumOf(_ ,1)

When a function name is written with fewer model element names than the number of places, so that some of the places are not filled, the result is another function, with fewer places.

In OCL functions are called operations.

The functions in a model are those defined or presented as primitive in that model. These may include standard functions provided in an extension specification.

4.5 Predicate names

A predicate name is used to form a statement. The meaning of a predicate name is a predicate, in the ordinary language sense of 'predicate.'

A predicate name is used by the modeler to make assertions about the system and its environment; which the modeler believes can be tested.

Every predicate has a certain number of places. A predicate name written with a model element name in each place is a statement.

Predicates can be combined with other predicates using logical connectives, such as ***and*** and ***not***. The combination is a predicate.

When a predicate is written with fewer model element names than the number of places, so that some of the places are not filled, the result is another predicate, with fewer places.

Examples of predicates:

__ is a CORBA trader
__ is greater than __
__ is between __ and __

Example:
__ is between __ and Needham

4.6 Quantifiers

Quantifiers are used to form statements from other statements. They include: ***for every***, ***for some***, and ***for no***.

4.7 Connectives

Connectives are used to build statements from other statements. They include: ***not***, ***and***, ***or, xor***, and ***if...then***. The connective, ***means***, is used to build a definition.

4.8 Identity

The identity symbol is used to specify that two names designate the same thing. The identity symbol is: ***is***.

is *may be written* **=**

4.9 Identity criterion

Each model element can be distinguished unambiguously from every other model element. The criteria for distinguishing one from another may vary according to the kind of model element. Unless otherwise specified by UML or by the model, the criterion is that each element added to a model is distinct.

This will be the case if an identical element is already in the model.
Example: If there is one dataValue representing the number 42 already in a model, then, after adding a dataValue representing 42, there is still one dataValue in that model representing 42.

The identity criterion for a model element may be such that adding that element to a model does not change the number of elements in the model.

The identity criterion for a model element is used in determining whether two names designate the same element.

Identity is distinct from equivalence.

4.10 Equivalence

is equivalent to *may be written* ≅

The equivalence symbol, is equivalent to, is used to specify that two things are equivalent for some purpose. The symbol has a meaning only with respect to some specific criterion of equivalence. The equivalence symbol is: ***is equivalent to***.

4.11 Binding

binds *may be written* **:=**

A binding is used to assign a constant name to a variable name. The binding symbol is: ***binds***.

When a constant name is assigned to a variable name, that variable name designates the same model element as the constant name.

Definition

An element ***is bound to*** a variable means that variable binds that element.

4.12 Statement construction and interpretation

Statements may be true or false. But, as a statement stands in a model, it is neither true nor false.

The truth or falsity of a simple statement is determined only by testing. That requires two steps:
— first interpreting that statement as a statement about the system
—and then observing the correspondence between that statement and the system.

So, if a model is a description of an existing system, the criterion for truth of a simple statement is whether that statement is true, when interpreted as a description of that system. If a simple statement in the model is not true as interpreted, then the model is in error.

The case is different if a model is a specification of a system to be built. The criterion for truth of a simple statement is still whether that statement is true, when interpreted as a description of that system. Now, however, if a simple statement in the model is not true as interpreted, then the system is in error.

Complex statements are composed by combining statements, using connectives, quantifiers, bindings and the identity and equivalence symbols. The truth or falsity of these complex statements depends on the truth or falsity of the simple statements of which they are composed. A complex statement is neither true nor false except when the simple statements are interpreted and the correspondence between those statements and the system is observed.

Example:
predicates:
__ is a CORBA trader
__ conforms to International Standard 13235
name:
corbaloc:iiop:1.1@fsarch.com/TradingService
statement:
corbaloc:iiop:1.1@fsarch.com/TradingService is a CORBA trader and conforms to IS 13235.

5 The UML 2 kernel

"Kernel refers to ... constructs that are used as the foundation for defining other... constructs." [RFP]

The kernel of the ***UML 2*** language includes the fundamental elements of the language and any additional elements needed to build a model of the language, that is, to define the remainder of the language using the language itself.

This submission specifies the kernel langauge

WARNING

Dear Reader: This section of our submission is a the language definition of the UML 2 kernel.

It is not a tutorial on the use of UML 2.
Nor is there a tutorial in another section of this document. A tutorial is provided at:
www.community-ML.org/2torial

The examples are given in terms of the language definition.
Thus they are not examples of the appearance of UML 2 models you might create.

You will create UML 2 models using drawings, property sheets, and the constraint language supplied by the tool vendor.

And, if your modeling tool provides full compliance, you may add constraints and other statements in this language.

The examples do illustrate the variety of exact UML 2 specifications that can be created.

5.1 Model Elements

Many of the modeling invariants and some of the definitions use terms that are introduced later in this document: there are forward references.

The name of an element may be hidden by abstraction; then that element appears as anonymous.

The collections used in this submission are specified in the OCL 2.0 and MOF 2.0 submissions.

Modeling invariants

Every element of a model is a basic model element or a defined model element

Every element of a model has a name.

Every element of a model is declared in a statement in that model.

Elements may be grouped into collections.

The basic elements are the representing elements, the system structuring elements, the model structuring elements, and the dynamic elements.

5.2 Representing model elements

Modeling invariants

The representing elements are objects, actions and associations.

Formally, this modeling invariant may be written:
For every e, *representingElement(*e*)* ***iff*** *Object(*e*)* ***or*** *Action(*e*)* ***or*** *Association(*e*)****.***
Similarly, it is straightforward to translate any of the definitions and model invariants in this specification into such a formalism, when that might be useful. For example, they all may be translated into OCL 2.0.

5.2.1 Object

An ***object*** is used when the modeler wishes to represent something that she or he regards as existing in the system modeled or in its environment.

Objects may be used to represent both things and concepts. The concepts represented may include numbers and other mathematical items.

An object may be a combination of other objects, representing something the modeler wishes to represent as having parts. Likewise, a single object may be used to represent something that has parts, without representing the parts in the model.

A model may include invariants for an object. A model may also classify an object as being of one or more types; in that case, the invariants of each of those types are invariants of that object.

Examples of objects:
UML 2 objects include objects representing:
— concepts in the problem area
— things in the environment of the system
— the system
— parts of the system, including
— subsystems of the system
— components in the system
— nodes of the system
— mobile software agents
— programming language objects
— data values
— encoded messages in transit
— data storage systems
— hardware
Objects may also be used to represent:
— parts of the delivered software
— documentation of the system
— records of the project

Definitions

A ***part*** of an object means a model element encapsulated by that object or a part of an object, which object is a combination.

Modeling invariants

At each time, each object has a state.

The state of an abstract object may be empty. The state of any object may be hidden by abstraction.

5.2.2 Action

An ***action*** is used when the modeler wishes to represent something that happens in the system or its environment.

The granularity of actions is a design choice. An action need not be instantaneous. Actions may overlap in time.

A model will also have action types and action templates.

Every action of interest results in a change of state of at least one object.

Depending on the granularity of actions, the change of state may not be observable.

For each action there are roles for objects. Each of the objects participating in an action is bound to one or more of these roles. A single object or a collection of objects may be bound to a role.

The objects bound to roles may be hidden by abstraction

Each action also has a role for the time of the action and a role for the place of the action.

This time and place may be hidden by abstraction.

A model may specify that an action has happened, is happening, or may happen.

Examples:
An action specified as a combination of the primitive actions specified in the UML Action Semantics.
UML 2 provides the partial order relation, before, for ordering actions. An ordered collection of actions is a combination of actions.

An action may be specified as a combination of other actions. Ways of combining actions are provided by UML 2.

Definitions

To ***participate in*** an action means to be an object bound to a role for objects of that action.

A ***participant*** in an action means an object that participates in that action.

An ***action*** of an object means an action in which that object is a participant.

Modeling invariants

Every action has at least one role for an object.

In every action, at least one object is bound to at least one of the roles for objects of that action.

Every action has a role for a time.

Every action has a role for a place.

Definitions

An ***action of an object*** means an action to which that object is bound.

The ***time*** of an action means the time bound to that action.

When an action ***happens*** means the time of that action.

The ***place*** of an action means the place bound to that action.

The ***initial action*** of a system in a model means the action of that system that is before every other action of that system in that model .

5.2.3 Association

We are using 'association' to mean a particular association of particular objects (or other elements). In UML 1 this is called 'link.' This is another example of the fact that we are object-oriented, rather than class-oriented. (See the Alternate Vocabularies for our discussion of names for UML concepts.)

A UML 2 model will usually also have association types and association templates, and these might exist without any instances (for example, there might be no associations of a particular association type).

An association is used when the modeler wishes to represent some structure of things in the system or its environment. It represents a structure that is present in the system or its environment and not only in the model.

For each association there are roles for the associated model elements. Each of these elements is bound to one or more of these roles. A single model element or the elements in a collection of elements may be bound to a role.

The invariants of the association are a collection of statements that, among them, mention each of the roles of that association at least once. The meaning of the association is provided by those statements.

An association also has roles for actions. These actions include the action that establishes the association and the action that terminates the association.

These actions may be hidden by abstraction.

Definitions

A ***participant in an association*** means an element bound to a role for associated model elements of that association.

This is simply a term of convenience.

The ***association invariants*** means the statements of an association.

Modeling invariants

An association is bound to the action that establishes that association and to the action that terminates that association.

These actions may be hidden by abstraction.

Each of the roles for participants of the association, and thereby each of the participants of that association, is mentioned in one or more of the association invariants.

This invariant provides the meaning of the association.
For example: 'The whole owns the parts and the lifetime of each part is coincident with the lifetime of the whole' is part of one version of an association invariant for UML 1 composition association; it mentions the objects in the roles, whole and part, of the links of that UML 1 association.

5.3 System structuring elements

5.3.1 Time

A time is used to represent some interval of time in the system or, if the modeler wishes, some moment in time.

In any model that specifies a behavior, there is (at least) an implicit finite collection of times and a partial order on those times.

In any model which specifies no theory of time, there is one time, any actions are bound to that time and there are no actions that terminate associations.

A time may be specified by relation to another time or by relation to some clock.

The particular modeling invariants of time will depend on the needs of the modeler. Specifications of distributed real time systems will use an explicit theory of time. Many application specifications will make do with naive universal clock time or will not be concerned with time at all, other than the simple relative time provided by this UML 2 kernel for use in behavior specification.

UML 2 uses the following tiny theory of time in order to have a technique to specify changes. This is a theory in which there is a partial order of times, called '***before***.' In this order, for any given time, there may be some times that are before that time.

Definition

The relation, ***before,*** is a binary relation of times.

Before may be written <

Modeling invariants

Another commonly used partial order of times is the relation often written as '$\leq$'. This is an antisymmetric, transitive and reflexive relation. It may be defined using the before relation: $t_1 \leq t_2$ means t_1 is before t_2 or t_1 is t_2.

The relation, before, of times is:
asymmetric (if t_1 is before t_2, then t_2 is not before t_1),
transitive (if t_1 is before t_2, and t_2 is before t_3, then t_1 is before t_3) and
irreflexive (t_1 is not before t_1).

Definitions

t_1 ***is before*** t_2 means t_1 before t_2.

t_1 ***is not before*** t_2 means not (t_1 before t_2).

t_1 ***is after*** t_2 means t_2 is before t_1.

t_1 ***is concurrent with*** t_2 means t_1 is not before t_2 and t_2 is not before t_1.

Action a_1 ***happens before*** action a_2 means the time of a_1 is before the time of a_2.

Before an action means in a time before the time of that action.

After an action means in a time after the time of that action.

Concurrent with an action means in a time concurrent with the time of that action.

Two actions are ***concurrent*** means one of the actions is concurrent with the other.

Immediately before an action means a time before that action and after every other action before that action.

Immediately after an action means after that action and before every other action after that action.

Example: Client opens file before client reads file.

Action a_1 ***before*** action a_2 means action a_1 ***happens before*** action a_2

Modeling invariants

There is at least one time.

5.3.2 Place

Specifications of geographic information systems will use an explicit and exact model of place. A model of the architecture of a distributed system might simply use place names or might use internet names as surrogates for place names.

A place is used to represent some region of space in the system.

UML 2 mandates no theory of place.

(Some definitions or modeling invariants mention place, for example, those of object, action, and interface.)

5.3.3 Purpose

A ***purpose*** is a model element used to express the practical advantage or intended effect of something represented by another model element.

Example:
The purpose of the checker is to ensure compliance.

5.4 Model structuring elements

5.4.1 Type

A type is used for three purposes:
— to classify elements,
— as a shorthand used when specifying an element
— for type checking.

Examples:
x is a NaturalNumber means x is an Integer and x is greater than zero.
Three alternate specifications for a type named Integer:
x is an Integer means x satisfies the Peano postulates.
x is an Integer means x is of an IntegralType or x is an instance of class Integer.
x is an Integer means x is of CORBA integer type as specified in formal/01-02-01, at 3.10.1.1.

A model specifies a type by providing a name for the type and statement about elements of that type.

The definition of a type takes the form of a definition containing exactly one variable name in the first part of the definition, with that variable name also appearing in the second part of the definition. The definition determines the conditions that must hold for an element to be of that type.

To determine if some element is of a certain type, we take the statement in the second part of the definition and substitute the name of that element for the variable name that also appears in the first part. The result is a statement about that element. If the statement holds, that element is of that type.

Example: Is 3 a natural number? Here is the type definition: x is a NaturalNumber means x is an Integer and x is greater than zero.
We get: 3 is an Integer and 3 is greater that zero.
So: 3 is of type, NaturalNumber.

An element is said to be ‘of’ the type, to ‘satisfy’ the type, or to be an ‘instance of’ the type.

An element of type, *T*, may be called ‘a *T* ’.

Example: Two is an Integer.

The specification of a type may include a criterion for determining when names of elements of that type designate the same element. Unless otherwise specified by UML, each element added to a model is distinct.

An element does not satisfy both of two types if the identity criteria specified for those types are such that an element can be distinct from another according to the identity criterion of one type, but identical according to the identity criterion of the other type.

This restriction is not necessary; it is included for compatibility with the MOF Core.

A model specifies which of the elements it uses have types. In UML, types are needed for, at least, objects, actions and associations.

An element may be of several types.

A model contains those only types declared.

This is a technical requirement, to ease implementation of tools, in particular, model checkers.

Type: NaturalNumber
Test: two is an Integer and is greater than zero
Result: two satisfies NaturalNumber.

Type: JavaInteger
Test: 9223372036854775808 is of an IntegralType or is an instance of class Integer
Result: 9223372036854775808 is not a JavaInteger.

Definitions

An element ***satisfies*** type, *T*, means the second part of the definition of that type holds after substituting the name of that element for each occurrence, in the second part of that definition, of the variable name found in the first part of that definition.

An element is ***of*** type, *T*, an element is ***an instance of*** type, *T*, and an element ***is*** a *T* all mean that element satisfies the type, *T*.

The types, T_i , ***used to define*** a type *T*, are all the types used in the definition of *T*, and all the types used to define any of those T_i .

A is a ***subtype*** of *B* means any element that satisfies the type *A* will satisfy the type *B*.

B is a ***supertype*** of *A* means *A* is a subtype of *B*.

Modeling invariants

Who does shave that barber?

Another way to write this: To every collection, A, and to every statement form, S(x), there corresponds a collection, B, whose elements are exactly those elements, x, of A for which S(x) holds. (This collection may, of course, be empty.)

A type is not used to define itself.

For every collection of elements and every type there is a second collection whose elements are exactly those elements of the first collection that are instances of that type.

The types in a model are those declared in that model.

5.4.2 Role

Examples of role in UML 1 are an AssociationEnd, a ClassifierRole, and a role in an Actor.

A role is used to specify an action, association or relationship without identifying a particular participant in that action, association or relationship. That is, a role serves as a placeholder, used when the modeler does not wish to be more specific at that place in the model.

The uses of role in UML 2 are specified by the modeling invariants of other model elements, such as action, association, and relationship.

For a given role, there may or may not be an element bound to that role.

For each role, there is a type that specifies the model elements that may be bound to that role.

Definitions

An element ***in*** a role means an element bound to that role.

An element ***performs*** a role, an element ***plays*** a role, and an element ***fulfills*** a role all mean that element is bound to that role.

Modeling invariants

Every element bound to a role satisfies the type for that role.

A role binding is a statement, mentioning a model element and a role, which uses the binding symbol to bind a model element to a role.

Role binding is used to specify that a particular model element plays a particular role.

A role identification is a statement, mentioning two or more roles, which uses the identity symbol to prescribe that the roles mentioned are the same role.

Role identification is used to build or combine collaborations, as in Reenskaug role model composition. A role identification declares that two different role names are names of the same role.

5.4.3 Relationship

An relationship is used when the modeler wishes to specify something about a collection of model elements (and not about the system or its environment).

Relationship are distinguished from associations, which specify something about the system or its environment.

The elements in the collection are each bound to a role or roles of the relationship. The meaning of a relationship is provided by the statements of the declaration of that relationship.

Examples: UML 1 generalization, dependency, abstraction, import.

Definitions

A ***participant*** in a relationship is an element bound to a role of that relationship.

A ***relationship invariant*** means a statement of a relationship.

Modeling invariants

There is at least one participant in each role of a relationship.

5.5 Dynamic model elements

5.5.1 Possibility

Possibility is used to indicate that some part of a model represents what is permitted, but not required by that model. A statement that an element is possible is used to specify that what is represented by that element may exist or happen in the system, given a certain situation.

A UML 1 state machine represents possible sequences of actions. In the UML action semantics, possibility was used in the definition of ConditionalAction. A UML 1 multiplicity prescribes the possible number of elements.

A model uses possibility to declare that something specified by the model is optional.

Examples: The number of interface types implemented by a class may be zero or more.
A file close action with a file may happen after a file open action with that file.
Every object is either local or remote.

In UML 2, possibility is expressed using the connective, ***or***. The specification of a choice between possibilities is expressed using the connective, ***xor***.

5.5.2 Change

Change is used to indicate that some part of a model represents a difference in the system or its environment from one time to another.

In UML 1, change is represented by Flow relationships. In the semantics included in the UML action semantics submission, change was represented by Change. (The UML action semantics was adopted without a semantics.)

In UML 2, change is expressed by a difference in state at different times.

Definitions

An action ***changes*** an object means the state of that object immediately before that action is not the same as the state of that object immediately after that action.

Modeling invariants

In a model that specifies change, there are at least two times.

5.6 Defined model elements

Using the techniques hammered out in the previous century, we start with a few primitive concepts, defined implicitly and then provide explicit definitions for the rest.

The previous model elements are not defined explicitly. Instead they are defined implicitly by the modeling invariants and definitions that mention them. The rest of the UML model elements have explicit definitions.

5.6.1 Attribute

Attribute is defined using the same approach as that of UML 1. There is no change in the use or meaning of UML attribute link, except that what UML 1 calls 'attribute link,' UML 2 calls 'attribute.'

UML 2 also provides another way to specify the value of an attribute. If there is a model of time, then any attribute may be represented as a function from time to the collection of possible values of that attribute ; the value of that function is another object, the value of the attribute at that time.

Definitions

An ***attribute*** of an object is an association between that object and an other, in which the first object is in the role, describedObject and the other object is in the role attributeValue.

The ***name*** of an attribute means the name of that association.

That is, the object in the role attributeValue

The ***value*** of an attribute means the attributeValue.

That is, the type that comes with the role attributeValue, which an object must satisfy to be in that role.

The ***type*** of an attribute means the type specified for the attributeValue.

A ***reference*** means an attribute with an objectIdentifier as the attributeValue

Modeling invariants

The value of every attribute satisfies the type of that attribute.

Every attribute of an object is contained by that object.

Every attribute of an object is encapsulated by that object, unless it is specified to be not encapsulated.

5.6.2 Data value

UML 2 provides the concept, data value, for those who find it useful.

If desired, data values may be used to represent references in the system, such as CORBA object references.

Definition

A ***dataValue*** means an object.

An ***objectIdentifier*** means a dataValue that, in a given naming context, identifies exactly one object.

Modeling invariant

No dataValue changes.

No dataValue is bound to an association other than as value of an attribute.

This invariant is included only to align UML 2 data value with UML 1 data value.

A type is specified for every dataValue; the specification of that type includes a specification of the identity criterion for dataValues of that type.

5.6.3 State

The ***state*** of an object determines the behavior of that object.

The state of an object at a given time is determined by
—the attributes of the object
—the associations in which the object is bound
—the invariants of the object.

The invariants of an object include those that prescribe the types of actions in which that object will participate, for example, operations.

Taken together, all the states of an object that are permitted by the model form a state space

The total number of states of an object may be very large; the modeler will often wish to use a smaller set of states. A model may partition the states of an object to obtain a small set of states.

Or the model may simply enumerate a small list of states for the object.

This is common practice when declaring a state machine in UML 1.

Definitions

A ***complete state*** of an object at a time means, at that time, the collection of model elements and statements containing the attributes of that object, the associations to which that object is bound and the invariants that apply to that object.

The ***complete state space*** of an object means the collection of all possible complete states of that object.

An ***abstract state*** of an object with respect to an equivalence relation means an equivalence class of complete states of that object.

For example, an object may have an attribute called state *and specify that all complete states with the same value of that attribute are the same abstract state.*

Using an equivalence relation and using a partition are mathematically equivalent ways of specifying a set of abstract states.

An ***abstract state*** of an object with respect to a partition means one of the collections of states from a partition of the complete states of that object.

In UML 1 state machines, the states are (usually) enumerated states. (The UML 1.4 specification does say that a state represents "some (usually implicit) invariant condition." If those invariants are explicit in the model, the states are abstract states, partitioned by the invariants.)

An ***enumerated state space*** for an object is a collection of abstract states specified by the model, for which the partition or equivalence relation is not specified by that model.

An ***enumerated state*** of an object with respect to an enumerated state space for that object is a state in that state space.

A ***state*** of an object means a complete state or an abstract state of that object.

Thus an object may have more than one state space.

A ***state space*** for an object means
— the complete state space of that object or
— a collection of abstract states of that object with respect to a certain partition or set of equivalence classes of states of that object or
— an enumerated state space for that object.

Modeling invariants

At every time, an object is in one state of the complete state space of that object.

A model need not specify any state spaces. (In any case, every object has a complete state space.)

At every time, an object is in one state of each of the state spaces a model specifies for that object.

For example, a state machine may be used to specify the state of an object with respect to an enumerated state space.

A model that specifies an enumerated state space for an object provides a means of determining the state of that object with respect to that state space at every time.

Definition

The ***state*** of a system at a time means, at that time, the collection of the states of each of the parts of that system.

The ***initial state*** of a system means the state of that system an the time before every other time in the model of that system.

5.7 Template

A template is specification for a model element, that can be used to create a model element of a certain type.

The element created using a template may be a combination of elements.

A template may have parameters. The parameters are roles; when a model element is created using a template, elements are bound to all parameters. The elements to be bound to parameters may be provided when the element is created. A template may specify

that some elements to be bound to roles of the template are themselves to be created, according to specifications in the template, when the model element is created. A template may specify that an element need not be bound to every role.

A template may be used to , the result is another template, with fewer roles.

Definition

Template means a combination with two roles, templateType, for an element type, and specification, for a collection of specifications for an element.

For example, in Java "class declarations define new reference types and describe how they are implemented." [Gosling : 8]

The ***template type*** of a template means the type in the role, templateType, of that template.

A ***parameter*** of a template means a role, specified in that template, that is to be bound to a model element provided when a model element is created using that template.

An element is ***created*** means a model is changed by addition of that element as the result of an action in the model.

Modeling Invariant

A model element created using a template, with model elements bound to all parameters, satisfies the template type of that template.

5.8 Class

A class is used to declare an object type and at the same time specify an implementation for objects of that type.

The specification of a class may also include specifications for a programming language class, for a home interface, for an object implementing class operations and attributes, for an object factory, for interfaces to be implemented or for any other aspects the implementation of objects of that type.

The particular kinds of specifications that may be included in a class may be specified in a language extension.

For example, a UML 2 language extension specific to Java or to CORBA components.

A language extension may include specification of model elements to be used in preparing a platform independent model for use in an MDA transformation to produce a platform specific model.

A language extension may include specification of marks for model elements, to be used in an MDA transformation to produce a platform specific model.

A UML 1 template class is an example of a class that is not completely specified.

A class may specify the common features of a collection of objects in sufficient detail that an object can be constructed using it.

For example, if a class includes an operation, but no method for that operation, that class has a parameter for the method.

Or a class may not completely specify an implementation. In that case, the class has parameters to be used to complete the specification.

A model may include instances of an abstract class.
An abstract class may have class methods and attributes in addition to the methods and attributes for the objects instantiated using subclasses of that class.

A model may specify, for a particular class, that, in the system, no objects be created using that class. Such a class is called an abstract class.

Definitions

Class means a combination of a declaration of an object type and a template for objects of that type, which may also include specifications for other model elements to be created along with those objects

Specifications of UML1Operation, MOFOperation, CorbaOperation and JavaOperation will be provided in the form of language extensions as examples of subtypes of the operation action type.

An ***operation*** of a class means a statement that mentions an operation type and states that every object of that class will participate in actions of that type.

A ***method*** of a class means the specification of the implementation of an operation in sufficient detail that an object can be constructed using it.

Base class means a base element that is a class.

Derived class means a derived element that is a class.

Superclass of a class means a base class of that class or any superclass of that base class.

Subclass of a class means a derived class of that class or any subclass of that derived class.

Modeling Invariant

The template type of a class is the object type of that class.

5.9 Type definitions

5.9.1 Subtype and supertype

Definitions

A is a ***subtype*** of *B* means any element that satisfies the type, *A*, will satisfy the type, *B*.

B is a ***supertype*** of *A* means *A* is a subtype of *B*.

5.9.2 Object type

Definition

> ***Object type*** means a type that is satisfied only by an object.

Thus, the definition of an object type will include: x *is an object.*

One familiar technique for specifying an object type is to specify the attributes, behavior and invariants common to all objects of that type.

This corresponds to the UML 1 «type» class.

This may be done in UML 2 by using the same statements that would be used to specify a particular object, but using a variable name in place of a constant name for the object. Using that variable name in the first part of a definition of a type and those statements as the second part results in a definition in the form required for the declaration of a type.

UML 2 does not limit the declaration of object types to this technique. Any statement may be use to specify an object type that the modeler finds useful.

In particular, an object type may be declared by declaring a class.

An object type is used, as any other type, for three purposes:

— to classify objects,
— as a shorthand used when specifying an object
— for type checking.

5.9.3 Action type

Definition

> An ***action type*** means a type that is satisfied only by an action.

This general definition allows the modeler to classify actions in many different ways. For example, thread safe action, insecure action, long duration interaction, time-critical action, interaction with trading partner, CORBA best-effort request, asynchronous message sending action, ODP announcement.

UML 2 enables the inclusion in a model of a range of more or less specific action types. Generic action type declarations are not required to be related to a specific object type or class. At the same time, very specific action types, mentioning particular objects, may be declared.

One technique to declare an action type is to use the same statements as used to specify an action, but not bind the action to a time (that is, to use a variable name for the time).

Example: The system [an object bound to the role, managed] notifies the central management service [an object bound to the role, manager].

A more general action type is declared by not binding objects to one or more of the roles.

Example: The system [an object bound to the role, managed] notifies manager [a role].

One very useful form of action type is declared using a constant name for the type, a variable name for the action, and an action specification with no roles bound.

Example, The action type, Notify:
x is notify means x *is an action and* x *is managed notifies manager [both are roles].*

This provides a generic action type that can be reused in different parts of the model.

Interface types are declared using such generic object types.

Operation types are used by ORBs and compilers for type checking. They are used by designers to specify interfaces.

Definitions

An ***operation type*** means an action type that is satisfied only by an operation.

5.9.4 Association type

This corresponds to a UML 1 Association. But see Alternative Vocabularies.

Definition

Association type means a type that is satisfied only by an association.

The UML 1 extension specification will include the association types, UML 1 aggregation and UML 1 composition

An association type may be used to specify a particular association, or, along with some object types, to specify that objects of those types are associated according to that association type.

A UML 2 association type is not restricted to use with a particular set of classifiers.

Definition

x is a navigable binary association *means x is an association with two roles, navigator and destination.*

Modeling invariant

Every object bound to navigator in a navigable binary association may make a request of the object bound to destination

An example of association types are those concerned with navigability. The example opposite is only an example and not a part of the specification of the UML 2 kernel (since navigability is not required to specify UML 2).

5.9.5 Specialization of association types

An association type is specialized by:

— adding statements to the invariants of that association

or

— adding roles to the association and adding to the invariants statements mentioning the added roles.

5.9.6 Specialization of behavior types

A behavior type is specialized by:

— adding to the statements prescribing when actions may happen

or

— specializing action types of the behavior type

or

— adding action types.

5.10 Generic association definitions

5.10.1 Combination

Choosing the name, combination, we avoid every part of the whole white-diamond-black-composition-aggregation situation.

Combination is an association that relates a whole to its parts.

Definition

Combination means a collection of associations each with two roles, whole and part, with an element in whole and a collection of elements in part

A ***part*** means an element in the role, part.

The parts means the collection of elements in the role, part.

A part of the whole means an element is in the parts.

The combination and ***the whole*** both mean the element in the role, whole.

x is ***a combination of*** *y*s means x is the whole and the *y*s are the parts.

w is ***a combination of*** *x*s, ... and *z*s means *w* is the whole, the *x*s are the parts in one association of the collection of associations, ... and *z*s are the parts in one association of the collection of associations.

There are many subtypes of combination important for various modeling purposes. The ways in which they are different are manifold. UML 2 enables exact specification of whichever distinctions the modeler feels are important for a particular model.

We provide these "obvious" definitions to point out that is easy to define what concepts in a modeling language mean, exactly.

Model Invariants

The element in whole is the same in each association of a combination.

The invariant of a combination specifies the way a property of the whole is determined, collectively, by properties of the parts and specifies a property of the whole that is independent of any properties of the parts.

5.10.2 Containment

Containment is an association between a container and its contents.

Example: A UML 1 package is a container.

Definition

Containment means an association with two roles, container and content, with a collection of elements in content and one element in container.

Container means an element in the role container of a containment.

Contents means the elements in the role content of a containment.

An element is ***contained in*** the container of a containment means that element is in the content of that containment.

The elements, e_n, that ***contain*** the contents of a containment means the container and all the elements that contain any of those e_n.

Model Invariants

No element contains itself.

5.10.3 Enclosure

Example: A UML 1 package is an enclosure.

Enclosure is a containment which restricts an element to being in one container.

Definition

Enclosure is a containment association.

An element is ***enclosed in*** an enclosure means that element is in the content of that enclosure.

The elements, e_n, that ***enclose*** an element ***in*** the contents of an enclosure means the container and all other elements that enclose any of the e_n.

Model Invariants

No element encloses itself.

An element is in the contents of at most one enclosure.

5.10.4 Encapsulation

Example: The attributes of an object are encapsulated.

Encapsulation is an enclosure association between an object and the elements it encapsulates. When an element is encapsulated in an object that element can participate in an action in which that object also participates.

Definitions

The action in which an encapsulated element participates might be an internal action of the object or an interaction with other objects: the object that encapsulates that element will always participate in that action.

Encapsulation is an enclosure association with two roles, encapsulator and encapsulated, with an object in encapsulator and a model element in encapsulated.

An object ***encapsulates*** an element means that object is in encapsulator of an encapsulation and that element is in encapsulator, of that encapsulation.

An element is ***encapsulated by*** an object if that object encapsulates that element.

An element is ***encapsulated*** if it is encapsulated by some object.

Modeling invariants

In an encapsulation, encapsulated participates in only those actions in which encapsulator participates.

This invariant is redundant, as it is true of all enclosure associations.

Every element is encapsulated by no more than one object.

5.11 Generic relationship types

5.11.1 Classification

Classification is used to systematically arrange some model elements according to some criteria.

A classification is a collection of relationships; each relationship relates a collection of model elements, in the role, classified, to a type, in the role, er, classifying (well, we can't call this role classifier, can we?).

Using classification, a modeler may assign model elements to types. The criteria need not be in the model.

This declares, for each of those types, which of those model elements are satisfactory when an element of that type is wanted.

Any elements of model, which have types in that model, may be classified.

There may be model elements without types and types that no model elements satisfy.

Definition

Classification means a collection of relationships, each with two roles, classified and classifying, with a collection of elements in the classified role and a type in the classifying role.

An object type (or a class) may be declared in advance, as is often done, and then objects declared to be of that type, or a type may be declared at the same time that an object or objects are declared to be of that type.

e is classified as *T* means *e* is in classified in a classification with type *T* in classifying.

T is a ***type*** of *e* means *e* is classified as *T*.

T is a *type of e, not* the *type of e, because e may be classified as several types*

Modeling invariants

An element that is classified as a type satisfies that type.

No type is both classified and classifying in the same classification.

5.11.2 Generalization

Generalization is a relationship, which is used to systematically arrange some types according to some criteria of the modeler. A generalization relates some types (the subtypes) to another type (the supertype); the subtypes are generalized by the supertype and the supertype is specialized by the subtypes.

Definitions

Generalization means a collection of relationships with two roles, generalized and specialized, with a type in specialized and a collection of types in generalized.

A is a ***specialization*** of *B* and *B* is a ***generalization*** of *A* both mean *A* is a type in the collection, specialized, in a generalization and *B* is the type in the role, generalized, in that generalization.

This definition does not rule out that the specification of multiple supertypes for a type. Of course, single generalization can easily be defined.

Modeling invariants

There are no circular generalizations.

If A is specialization of B then A is a subtype of B and B is a supertype of A.

5.11.3 Specialization

We do not propose any modeling invariants for specialization. We will reconsider this again after review of the next revisions of the MOF Core submissions.

The Superstructure submission will define other relationships, including:
—viewpoint correspondence,
—platform independent to platform specific model correspondence.

5.11.4 Abstraction

Abstraction is used to declare that a collection of model elements is a simplified model produced from some other more complex collection of model elements by suppressing some detail. The more complex collection of elements is abstracted into the simpler collection.

A simple abstraction is a single relationship between one abstract model element and a collection of of model elements.

Hiding (not identifying) the operations of a class in a class diagram is a simple form of abstraction.

Example: a UML multi-object.

Example, a UML subsystem, or, anyway, something like that.

And there are many others.

There are several simple forms of abstraction, including, for example:

- an abstraction of a collection of similar elements into a single element of that kind, which suppresses the number of elements
- an abstraction of a structure of elements into a single element representing the structure, which suppresses the details of the structure and its parts
- an abstraction of a collection of interactions of several objects that accomplishes some purpose into a single interaction that suppresses the details of those interactions.

The partition of a set of components into a smaller number of subsystems is a complex abstraction, in which each subsystem is itself an abstraction of a collection of components.

A complex abstraction is a collection of other abstractions. When a collection of model elements is abstracted into a collection of abstract elements, those abstract elements must themselves form a single, consistent, simpler model.

Definitions

An ***abstraction*** means a collection of relationships; each relationship has
— two roles, refined and abstract, with a collection of elements bound to each role,
— an invariant that specifies what is hidden by each relationship of the abstraction.

An element is ***hidden by*** an abstraction means that element appears in a refined role, but not in an abstract role of that abstraction.

When *x* is the collection of all the elements in the abstract roles of an abstraction relationship and *y* is the collection of all the elements in the refined roles of that abstraction, then ***abstraction of*** *y* may be used to mean *x* together with that abstraction relationship.

If *x* and *y* are each a collection of model elements, then *y* ***is a refinement of*** *x* means *x* is an abstraction of *y*.

If you construct a complex abstraction out of other abstractions, then the abstraction invariants are a constant part of the construction. That is, there is only one abstraction invariant that applies to each part and to the whole: this may be called the invariance of abstraction invariants under construction. Is this a desirable property of the language?

Modeling invariants

A refinement of *x* consists of more model elements than *x*.

Here is an opportunity for readers to contribute: What are the modeling invariants that apply to abstractions?
mailto:joaquin@acm.org

5.11.5 Language extension

Definition

LanguageExtension means a three part relationship with roles baseLanguage, extension specification, and extendedLanguage

Modeling Invariants

The specification of the extendedLanguage of a language extension is the specification of the baseLanguage and the extension specification.

Example: A specification of UML 1.5 is the specification of the UML 2 kernel followed by the UML 1.5 extension specification.

6 Model management

Model management is a matter separate from the language of the model being managed. So the definitions and modeling invariants in this section specify the management of models, not the UML 2 language.

Models are best managed in a repository. So the model management architecture of UML 2 should be suited to a MOF. If suited to a MOF, it will also be suitable for file based modeling tools.

6.1 Package

UML 2 uses the MOF 2 packaging specification. That is, a UML 2 package is a MOF package.

Our submission group will review the MOF 2.0 Core submissions before completing the model management specification. It was our intention to adopt MOF model management or a subset of it for UML 2,we see no reason for incompatibility in model management between the adopted modeling language technology and the adopted model repository technology.
However, both MOF submission teams presently intend to use the package structure of the adopted UML infrastructure. We have decided to step aside for the moment, and await developments.

6.2 Name and Namespace

UML 2 uses the MOF 2 naming specification.

6.3 The system

Definition

The system, when not otherwise qualified, means the system described or specified by the model.

Modelling invariant

Every model which is a software specification contains one object identified as the system described or specified by that model.

7 MOF extensions

These are extensions to the UML kernel to provide a subset of the UML 2 language that is identical with the MOF Model. These same extensions will be included in the UML 2 Superstructure submission as part of the UML 2 basic langauge.

7.0.1 Incremental modification

Incremental modification is a relationship between specifications of model elements.

Definition

An ***incremental modification*** means a relationship with two roles, with a collection of model elements in the role, baseElements, and a model element in the role, derivedElement.

Modeling Invariants

We do not find any invariants for incremental modification that should be placed in the UML 2 kernel.

7.0.2 Inheritance

Inheritance is a relationship between specifications of classes and other templates.

In UML 2, inheritance is the same as is described in terms of descriptors in the UML 1 specification. [UML 1.4 p. 2-75]

Inheritance is a specific kind of incremental modification.

Definitions

Inheritance means an incremental modification of templates.

Base element means an element in the baseElements of an inheritance.

Derived element means an element in derivedElement of an inheritance.

Single inheritance means an element may have only one base element.

Multiple inheritance means an element may have several base elements.

Strict inheritance means every statement of each of the base elements is a statement of the derived element.

For example, in strict inheritance every well-formedness rule that applies to a class, alos applies to all its subclasses.

Modeling invariants

We do not find any invariants for inheritance that should be placed in the UML 2 kernel.

7.0.3 Instantiation

Instantiation is a relationship between a model element, a template and an object called the instantiator. Instantiation is used to specify that model element is produced by an action in which the instantiator participates and which uses the template as specification of the element produced.

For example, a modeler may use an instantiation relationship between a particular object in a model and a particular class in that model, with that class in the role, template.

The object name myDarlingWife:Person can be used in UML 1 to specify an instantiation relationship: that the object, myDarlingWife, is an instantiation of the «implementation» class, Person. UML 1 does not distinguish this from the use of the object name myDarlingWife:Person to specify a classification relationship: that the object, myDarlingWife, is an instance of the «type» class, Person.

Definition

Instantiation means a relationship with four roles, element, template, instantiator and instantiation, with a model element in element, a template in template, an object in instantiator, and an action in instantiation.

The instantiator may be hidden by abstraction; that is, the model may show only the template and the element.

To instantiate means to produce using a template.

An ***instantiation action*** means an action with role, instantiator, by which the object in that role instantiates an element.

A UML 1 create action is an action of some object that causes an instantiation action by something else. That something else does not appear in the UML 1 model; the instantiator can not be shown in UML 1.

An ***instantiation*** of a template means an element produced using that template.

Modeling Invariant

The element of an instantiation is produced by an action of the instantiator using the template.

In Java, for example, the instantiator is the Java virtual machine (sometimes invoked via a class object); the action is the execution of a new *statement.*

The object in the role, instantiator, of an instantiation action is the object in the role, instantiator, of an instantiation relationship with the object produced by that action in the role, element.

7.0.4 Communication

Communication is used to represent the conveyance of information between parts of the system being specified or between that system and its environment.

Examples:
—An object returns to another object the result of an operation that was requested by that object.
—An object raises a signal.
—An object requests an operation.

A communication is an action that results in the conveyance of data between two or more objects as a result of one or more interactions, possibly involving some intermediate objects. The data conveyed is represented by an object.

Definition

A ***communication*** means an action that conveys an object between two or more objects.

A ***message*** means an object conveyed in a communication.

Modeling Invariant

We are not satisfied with the definition of communication and leave the modeling invariant(s) to be specified after the definition is improved, in a another revised submission or during finalization.

7.0.5 Operation

An operation is an action used to specify that one object is requested by another to perform some function, the other object may provide some data with the request, the object performs the function and may return data.

Definition

The role names, 'client' and 'object' are chosen to conform to the CORBA object model.

For further definition of 'operation' see [ODP 3-7.1.2]

'Provide' and 'return' will be defined in a revision of this submission.

An ***operation*** of an object means an action with
three roles, client, object and providedMessage
or an action with four roles, client, object, providedMessage and returnedMessage,
with that object bound to the role, object;
the client provides the providedMessage to the object, which participates in the action and may return a returnedMessage;
the action may or may not change the object.

7.0.6 Query

Definition

A ***query*** of an object means an operation of that object, which returns a message to the client and does not change the object.

7.0.7 Renaming

Renaming is used when combining two parts of a model to make another part of that model.

Renaming is used differentiate two elements, that have the same name in the different parts of the model they come from, by changing the name of one or both. It is also used to identify (that is, make the same) two elements, by giving them the same name.

Roles are often renamed when model elements with roles are combined.

Definition

Renaming means a relationship with three roles, element, with a model element in that role, and oldName and newName, with the name of a model element in those roles.

Modeling Invariant

When model elements are combined, two roles of a model that have the same name must be played by the same model element.

Example:
A model has the action type, a Makes b, and another, a Checks d, both well specified. Our modeler wishes to specify the MakerChecker action type: a Makes b and then c Checks b, an important business behavior pattern. She wants to do this by reusing the existing action types, instead of by re-specifying making and checking. So, starting with the action type, a Checks d, she renames a to c and renames d to b; she then combines the result with a Makes b and identifies the two roles named b. This differentiates one a in the original action types from the other and identifies the roles b and d of the original action types. She now has the action type, a Makes b and c Checks b. Finally she adds the invariants:
a makes b before c checks b.
the same object does not play both a and c.

The second invariant is why the MakerChecker pattern is effective.

8 Pattern application extensions

These are extensions to the UML kernel to provide a subset of the UML 2 language that is adequate for a variety of forms of pattern application. It is included in this submission to accomodate the techniques of the 2U submission. These same extensions will be included in the UML 2 Superstructure submission as part of the UML 2 basic langauge.

8.0.1 Pattern application

A pattern application is a relationship with three roles, each for a collection of model elements. It specifies the relationship between an original collection of model elements, a pattern applied to that collection, and a collection of model elements that is the result of the application of that pattern.

This may be changed after discussion in Helsinki with the 2U submission group, to ensure alignment of the two submissions.

Definition

A ***pattern application correspondence*** means a collection of relationships; each relationship has three roles:
— original, for a collection of model elements
— patternParameter, for a variable name in a pattern
— result, for a collection of model elements

An ***original element*** in a pattern application correspondence means an element in the collection in the original role of that correspondence.

A ***pattern parameter*** in a pattern application correspondence means a variable name from a pattern, which name is in the patternParameter role of that correspondence.

A ***result*** in a pattern application correspondence means an element in the collection in the result role of that correspondence.

An element is the ***result of a pattern application*** means that element is in the collection in the result role of some pattern application correspondence.

An collection of original elements ***corresponds to*** a collection of results means those elements are in the same correspondence of a pattern application correspondence.

8.0.2 Renaming

Renaming, as specified in the MOF extension, is also included in the patttern application extension.

9 Conformance

The purpose of a model is to describe an existing system or to specify either a modification to an existing system or a new system to be built.

We may say that the meaning of a model is either:
It is like this.
or
Make me one like this.

Conformance testing is determining whether a system conforms to a model.

If the model is a description of an existing system, and the system does not conform to the model, then the model is wrong.

If the model is a specification for a new system or a modification, and the system does not conform to the model, then the system is wrong.

9.1 Representation

The possibility to compare a model to a system depends on the elements of the model representing things in the system and things that happen with the system.

9.2 Interpretation

An interpretation of a model relates elements of the model to what it is that the elements represent.

9.3 Testing

With an interpretation of a model, it is possible to test conformance. Testing is the process of exercising the system with the intent to produce errors. A successful test is a test that does produce an error.

9.4 Approach to conformance

[The following text is a slightly revised version of the ODP approach to conformance text.

community-ML.org/RM-ODP/Part2/15.html

Conformance relates a system to a model. Any statement in the model must be true when it is interpreted in terms of the system.

A conformance statement is a statement that identifies conformance points of a model and the behavior which must be satisfied at these points. The conformance statement must be such that there is the possibility of testing.

UML 2 identifies certain reference points as potentially declarable as conformance points in models. That is, as points at which conformance may be tested and which will, therefore, need to be accessible for test. However, the requirement that a particular reference point be considered a conformance point must be stated explicitly in the conformance statement of the model concerned.

9.4.1 Testing and reference points

The facts about a system can only be determined by testing and is based on a mapping from statements in the model to observable aspects of the system.

At any specific level of abstraction, a test is a series of observable actions, performed at prescribed points known as reference points, and only at these points. These reference points are accessible interfaces. A part of a system for which conformance is claimed is seen as a black box, testable only at its external interfaces. Thus, conformance is not dependent on any internal structure of the part of the system under test.

In some testing arrangements, access to what is ordinarily considered as internals of the system (for example, private attributes of an object) may be provided to the tester. This acces is provided by additional external interfaces. As a result, in this case the system under test is not identical to the system that will be deployed.

9.4.2 Classes of reference points

A conformance point is a reference point where a test can be made to see if a system meets some conformance criteria. A conformance statement must identify where the conformance points are, and what criteria are satisfied at these points. Four classes of reference points at which conformance tests can be applied are defined.

Programmatic reference point

A programmatic reference point is a reference point at which a programmatic interface can be established to allow access to a function. A programmatic conformance requirement is stated in terms of a behavioral compatibility with the intent that one object be replaced by another. A programmatic interface is an interface which is realized through a programming language binding.

Perceptual reference point

A perceptual reference point is a reference point at which there is some interaction between the system and the rest of the world.

NOTES

1 - A perceptual reference point may be, for example, a human-computer interface or a robotic interface (specified in terms of the interactions of the robot with its physical environment).

2 - A human-computer interface perceptual conformance requirement is stated in terms of the form of information presented to a human being and the interaction metaphor and dialogues the human may be engaged in.

Interworking reference point

For example, CORBA technology specifications are based on the interconnection of interworking reference points (the physical medium).

An interworking reference point is a reference point at which an interface can be established to allow communication between two or more systems. An interworking conformance requirement is stated in terms of the exchange of information between two or more systems. Interworking conformance involves interconnection of reference points.

Interchange reference point

An interchange reference point is a reference point at which an external physical storage medium can be introduced into the system. An interchange conformance requirement is stated in terms of the behavior (access methods and formats) of some physical medium so that information can be recorded on one system and then physically transferred, directly or indirectly, to be used on another system.

9.4.3 Change of configuration

The testing of conformance may take place at a single reference point, or it may involve some degree of consistency over use in a series of configurations involving several reference points. This may involve the testing of conformance to

a) the requirement for a part of the system to be able to operate after some preparatory process to adapt it to the local environment;

b) the requirement for a part to operate according to its specification at a particular reference point from initialization onwards;

c) the requirement for a part to continue to work when moved into a similar environment during operation.

The properties being tested above give rise to properties of the objects or interfaces involved, as follows.

Portability

Portability means the property that the reference points of an object allow it to be adapted to a variety of configurations.

NOTE - If the reference point is a programmatic reference point, the result can be source-code or execution portability. If it is an interworking reference point, the result is equipment portability.

Migratability

Migratability means the ability to change the configuration, substituting one reference point of an object for another while the object is being used.

9.5 The conformance testing process

Conformance is a concept which can be applied at any level of abstraction. For example, a very detailed perceptual conformance is expected to a standard defining character fonts, but a much more abstract perceptual conformance applies to screen layout rules.

The more abstract a model is, the more difficult it is to test. An increasing amount of system-specific interpretation is needed to establish that the more abstract statements about the system are in fact true as interpreted. It is not clear that direct testing of very abstract models is possible at reasonable cost using currently available or foreseeable techniques.

The testing process makes reference to a model. To be complete, the model must contain:

a) the behavior of the object being standardized and the way this behavior must be achieved.

b) a list of the primitive terms used in the model when making the statements of behavior.

c) a conformance statement indicating the conformance points, what systems must do at them and what information implementers must supply.

There are two roles in the testing process: the implementer and the tester. The first role is that of the implementer, who constructs an system on the basis of the model. The implementer must provide a statement of a mapping from all the terms used in the model to things or happenings in the real world. Thus the interfaces corresponding to the conformance points must be indicated and the representation of messages given.

If the model is abstract, the mapping of its basic terms to the real world may itself be complex. For example, in a computational viewpoint specification (see Recommendation X.903 | International Standard 10746-3), the primitive terms might be a collection of interactions between objects. The implementer wishing to conform to the computational viewpoint specification would have to indicate how the interactions were provided, either by reference to an engineering specification or by providing a detailed description of an unstandardized mechanism (although this course limits the field of application of the system to cases in which there is an agreement to use the unstandardized mechanism).

The second role is that of the tester, who observes the system under test. Testing involves some shared behavior between the testing process and the system under test. If this behavior is given a causal labelling, there is a spectrum of testing types from

a) passive testing, in which all behavior is originated by the system under test and recorded by the tester;

b) active testing, in which behavior is originated and recorded by the tester.

Often, the specification of the system under test is in the form of an interface, as is the specification of the tester and test procedures. When testing takes place, these interfaces are bound.

The tester must interpret its observations using the mapping provided by the implementer to yield facts about the system which can then be checked to show that they correspond to statements in the model. Testing is the process of examining and exercising the system with the intention to find something wrong.

9.6 The result of testing

A successful test is one that finds something wrong; that is, a a case when the facts do not correspond to the statements in the model, as interpreted.

The testing process fails if all the facts match the model. In this case, the testing process has failed to find anything wrong.

However, the testing process can succeed because

a) the model is logically inconsistent or incomplete, so that the facts about the system cannot be checked (this should not happen);

b) the mapping given by the implementer is logically incomplete, so that it is inconsistent or observations cannot be related to terms in the model; testing is impossible.

c) the observed behavior cannot be interpreted according to the mapping given by the implementer. The behavior of the system is not meaningful in terms of the model, and so the test succeeds.

d) the behavior is interpreted to give terms expressed in the model, but these happen in such a way that they yield facts that do not correspond to the statements in the model, and so the test succeeds.

9.7 Relation between reference points

. The flow of information between modelled system parts may pass through more than one reference point. For example, a distributed system may involve interactions of two parts A and B, but communication between them may pass in turn through a programmatic interface, a point of interconnection and a further programmatic interface.

A refinement of the same system may show interconnected parts that have more than one part on the communication path between them.In either case, conformance testing may involve

a) testing the information flow at each of these reference points;

b) testing the consistency between events at pairs of reference points.

In general, testing for correct behavior of a configuration of objects will require testing that statements about communication interfaces are true, but it will also require observation of other interfaces of these objects, so that the statements about the composition can also be checked.
The general notion of conformance takes into account the relation between several conformance points. Since the model related to a given conformance point may be expressed at various levels of abstraction, testing at a given conformance point will always involve interpretation at the appropriate level of abstraction. Thus, the testing of the global behavior requires coordinated testing at all the conformance points involved and the use of the appropriate interpretation at each point.

In particular, conformance of a template to a given programmatic interface can only be asserted when considering the language binding for the language in which the template has been written and compliance of the written template to the language binding specification, which must itself be conformant with the abstract interface specification.

10 Notation

10.1 UML 1 Notation

UML 2 uses the same notation as UML 1.

The notation consists of drawing elements and property sheets. The appearance of drawing elements is specified here. Property sheets contain data determined by all statements not represented by a drawing element. They may also contain data that is represented by a drawing element. The appearance of property sheets is not specified.

An object is shown as a box with the name of the object underlined. The attributes of an object may be shown as in UML 1. The type of the object and a role may be shown as in UML 1: a colon before the type name and a virgule or forward slash before the role name. The name of the object template may replace the type name, if the object is an instantiation of a template.

As with all elements, for every object there is a property sheet. That property sheet contains data determined by the statements specifying that object.

An object type is shown with the same notation as for an object, but with name of the object type not underlined and no object or role name.

A role for an object is shown with the same notation as for an object, but with a role name.

An object template is shown with the same notation as for an object, but with name of the object not underlined.

An association is shown as a path, as provided for links in UML 1, 3.42.2 Link Notation. Other association adornments may be shown on the path ends.

Associations (which correspond to UML 1 links) ***do*** have names. The name of an association may be shown as a name string near the path (but not near enough to an end to be confused with the name of a role). The name string may have a small black solid triangle in it. The point of the triangle indicates the direction in which to read the name. (The reading intended to be indicated by the triangle can be ambiguous in the case of an n-ary association. The ordering of the roles is not ambiguous in the model: it is the order of first appearance in the statement.)

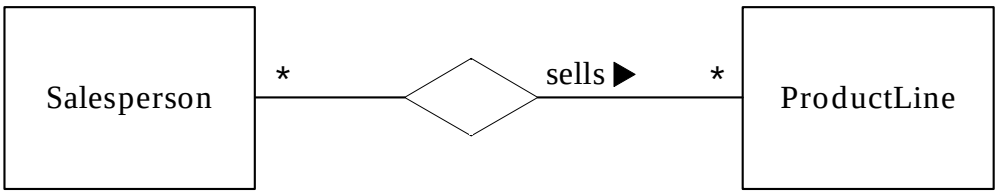
The name of a role of an association may be shown as a name string near the end of the path indicating the role of the object attached to the end of the path near the rolename.

The notation for association types is the same as that for associations. Association types may be shown between any of object types, classes, and other object templates.

The notations for an object in a role of an action are the several arrow notations of UML 1.

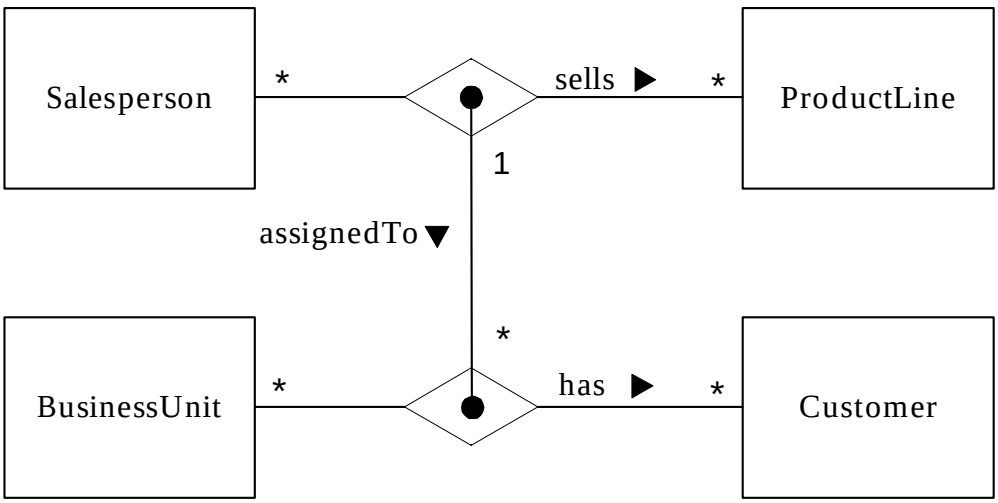
10.2 Additional notation

The UML 1 diamond notation for an n-ary association may be used in UML 2 for a binary association.



A language extension may provide icons to be included in the diamond to indicate different generic association types.

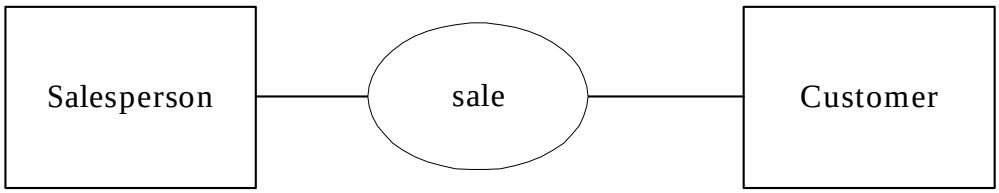
The notation for an association with another association bound to a role is the diamond notation for the bound association, with a dot in the center of the diamond, to which the end of the line from the binding association is connected



The illustration is is not a contrived example, but an actual model from a system built for MCI. A salesperson *may be trained to* sell *several* product lines. *A* business unit has *many* customers, *and a* customer *may do business with more than one* business unit. *For each relationship of a* customer *to a* business unit, *one* salesperson *who* sells *a particular* product line *is* assigned *to handle that relationship for that partictular product line.*

The notation for an action in which several objects participate is based on the one of the UML 1 notations for a collaboration. The action is presented as an oval, with lines to the participating objects.

A tool may provide the capability to zoom in from an action to a more detailed specification of that action.



Depending on the purpose of the model, the drawing might represent, for example:

In in a business model: a type of joint action *of in which a* customer *and a* salesman *participate.*

In a software model, a type of transaction *between a* salesperson proxy object *and a* customer purchasing system.

In another software model, a type of operation execution*, in which a* salesperson object *invokes the* sale *operaton of a* customer object *and receives a result.*

11 Differences between UML 1 and 2

This section discusses differences between UML 2 and UML 1. None of these differences affect the backward compatibility of UML 2, as UML 1 will be specified as an extension of the UML 2 kernel.

11.1 Generalization of UML 1 by UML 2

The following concepts are generalized.

11.1.1 Object

Objects are use to represent any thing the modeler regards as existing in the system or its environment.

11.1.2 Association

This submission uses ‘association’ for what UML 1 calls ‘link,’ and ‘association type’ for what UML 1 calls ‘association.’ (See the section Alternative Vocabularies.)

Association is generalized to allow, not only objects, but also actions and other associations to participate in an association.

Associations, which correspond to UML 1 links, ***do*** have names.

11.1.3 Constraint

Constraint is generalized to statement. Statements may constrain, but may also be positive. Expressions may be used in statements, but a statement is not an expression that is evaluated to obtain a value.

Notice that any constraint may be expressed as a statement. The statement 'o is constrained so that c' has the same practical value as the constraint 'c' attached to o.

For backward compatibility, constraints may continue to be use wherever they are used in UML 1 and wherever OCL 2.0 specifies that they may be used.

11.2 Refactoring of UML 1 by UML 2

The following concepts are refactored.

11.2.1 Class

Class is separated from type.

11.2.2 Relationship

Relationship is a term of convenience without any specific [meaning]. [UML 1.4–2.5.2]

Association is separated from relationship. This is because of the significant difference in meaning. This separation will not affect implementations. There will be no required change in the way association, generalization, dependency and flow are used by modelers.

11.3 Repositioning of UML 1 by UML 2

The following concepts are moved.

11.3.1 Use case

A use case is moved from a kind of classifier to a kind of behavior specification.

11.3.2 ElementOwnership

ElementOwnership is changed from an association class to an ordinary association.

The function of the attribute, visibility, of ElementOwnership is provided by the invariant of the association.

(The attribute, isSpecification, of ElementOwnership is not used in the UML 2 kernel, nor in the UML 1 metamodel.)

Part III

There seems little point in requiring compliance if no vendor intends to comply.

Summary of optional versus mandatory interfaces

An implementation must support all the interfaces. However, the submitters are not aware of any current implementations claiming UML 1 compliance that support the interfaces required by the UML 1 adopted technology. For this reason, support of the interfaces is a separate compliance point.

Proposed compliance points

All implementations claiming to provide UML modeling must comply with the entire UML infrastructure specification.

There are two basic compliance points, limited compliance and full compliance. All implementations claiming to provide UML modeling must conform to one of the two basic compliance points.

There are three additional compliance points, MOF compliance, model checking and CORBA interfaces support.

Limited compliance

The implementation:

- allows entry as drawings of all elements of the basic notation;

- allows entry as OCL 2.0 text of all elements of the basic language which it does not allow to be entered as drawings;

- generates XMI from models and models from XMI.

Basic compliance

The implementation provides limited compliance and also:

- allows entry as OCL 2.0 text of all elements of the basic language;

- will present all model elements as OCL 2.0 text.

Full compliance

The implementation provides basic compliance and also:

- allows entry as OCL 2.0 text of all elements of the language extensions included in this specification;

- will present all model elements as OCL 2.0 text.

MOF compliance

The implementation:

- Obtains its language definition from a MOF compliant repository.

- Will use language definitions from a MOF compliant repository that are extensions of the UML 2 kernel

Tools claiming MOF compliance must specify the MOF version or versions with which they interoperate.

Model checking

The implementation:

- Provides assistance in checking the consistency of the OCL 2.0 taken together with those model elements entered as drawings and not represented as OCL 2.0 by the tool.

In the case of tools that provide full compliance, the checking is on the model presented by the tool as OCL 2.0 text.

Tools not expected to provide a proof of consistency. Tools claiming compliance with model checking are expected to detect and report many inconsistencies, for example, by generating counter examples.

We do not expect to include compliance tests for model checking in our final submission.

CORBA interfaces support

The implementation:

- Complies with the CORBA Core specification as an object supporting all the interfaces specified by the IDL definitions of this UML 2 specification.

Changes or extensions required to adopted OMG specifications

The final submission will include a full specification of any changes or extensions required to existing OMG specifications.

We do not expect any changes will be required, except that our final submission will include a discussion of possible changes to the OMG MetaObject Facility specification.

Complete IDL definitions

Our final submission will include a complete listing in compilable form of the IDL definitions proposed for standardization.

12 Appendix: UML 1 in UML 2

A extension specification to the UML 2 kernel that provides a complete specification of the part of the UML 1 language that corresponds to the UML 2 kernel will be included in our final Infrastructure submission.

A extension specification to the UML 2 kernel that provides a complete specification of UML 1 language, as we understand it, will be included in our final Superstructure submission.

13 Appendix: Alternative Vocabularies

Some of the names use in this submission for modeling concepts are different from those used by UML 1. This is because we are object-oriented: of what Reenskaug calls the West Coast school.

This submission does not propose to change the UML 1 concept names. That is a decision that the Task Force and Technical Committee must make, and on which the Architecture Board may have an opinion.

The decision whether to change the UML 1 names to a smaller and more uniform set of concept names is an issue for the Task Force to discuss and decide.

The table shows the UML 1 names that correspond to the names used in this submission. The reason that we use these different names in this submission is that our useage enables the explanation of UML 2 to be more exact, clearer, easier to understand and smaller.

Table 1: Alternative Vocabularies

1.4	Submitted 2.0	Adopted 2.0
Object	Object	Object
Class «type»	ObjectType	?
Link	Association	?
Association	AssociationType	?
none?	Action	?
?	Joint Action	?
Action	Action Type	?
Class		?
Class «implementation»	Class	?
Generalization	Generalization	?
Generalization	Inheritance	?

The question marks in the first column indicate our uncertainty about the meaning of the UML 1 specification.

The question marks in the third column indicate our feeling that the UML 2 vocabulary needs a great deal of open discussion in the Task Force.

Note that in the CORBA Object Model "an object type is a type whose members are object references."

Class is defined so as to be practical for use in specifying software classes. This is dictated by the design principles of practicality and utility.

This table will be extended to cover the superstructure in that submission.

14 References

We use an order n language, not an object calculus, but we draw on the work of Abadi and Cardelli and others who use an object calculus.

[Abadi] Martín Abadi and Lucca Cardelli, A Theory of Objects, New York: Springer, 1996. ISBN 0-387-90092-6

Doing it right.

[Bennett] Douglas Bennett, Designing Hard Software, Greenwich, Connecticut: Manning, 1997. ISBN 0-13-304619-2

The source of the ontology we use as the foundation of UML 2. (See [Smith] below.)

[Bunge] Mario Bunge, Treatise on Basic Philosophy, Dordrecht, Boston, London: D. Reidel, 1979. In particular: Volume 3 : Ontology I : The Furniture of the World ISBN 90-277-0780-4

Software architecture patterns.

[Buschmann] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, Pattern Oriented Software Architecture—A System of Patterns, Chichester: Wiley, 1996. ISBN 0-471-95869-7

An approach to formal semantics for UML.

[Clark] Tony Clark, Andy Evans, and Stuart Kent, "The Meta-Modeling Language Calculus: Foundation Semantics for UML," Fundamental Approaches to Software Engineering, FASE2001, Heidelberg: Springer, 2001. LNCS 2029

st72095.inf.tu-dresden.de/START/program/Abstracts/34.html

*Yes, it's true: inheritance is **not** subtyping.*

[WCook] William R. Cook, Walter Hill and Peter S. Canning, "Inheritance is not subtyping," POPL '90, Proceedings of the seventeenth annual ACM symposium on Principles of Programming Languages, New York: ACM, 1990.

www.acm.org/pubs/contents/proceedings/plan/96709/

Where we learned of the idea of preface, which we adopt as extension specification.

[SCook] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer and Alan Cameron Wills, "Defining UML Family Members Using Prefaces," TOOLS'99 Technology of Object-Oriented Languages and Systems—Pacific, Piscataway: IEEE, 1999.

dlib.computer.org/conferen/tools/0462/pdf/04620102.pdf

A short and sweet description of the method used in defining UML 2.

[Corcoran] John Corcoran, "Axiomatic Method," in Robert Audi, Cambridge Dictionary of Philosophy, Cambridge: Cambridge University Press, 1999. ISBN 0-521-63722-8 and 63136-X

Himself.

[Dijkstra] Edsger Dijkstra, Selected Writings on Computing: A Personal Perspective, Berlin: Springer, 1982. ISBN: 0-387-90652-5

[Génova] Gonzalo Génova, Juan Llorens and Paloma Martínez, "Semantics of the Minimum Multiplicity in Ternary Associations in UML," Proceedings of the Fourth International Conference on the Unified Modeling Language, Berlin: Springer, 2001 LNCS 2185 ISBN 3-540-42667-1

An explanation of why UML needs inner and outer multiplicities, just as CDIF said.

[Gosling] James Gosling, Bill Joy and Guy Steele, The Java Language Specification, Reading, Massachusetts: Addison-Wesley, 1996. ISBN 0-201-63451-1

An example of a well written language specification.

[Halmos] Paul Halmos, Naïve Set Theory, College Park, Maryland: Litton, 1960. ISBN 0-387-90092-6

The classic (in case a reference is wanted).

[Harel] David Harel, D. Kozen and J. Tiuryn, "Dynamic Logic," Handbook of Philosophical Logic—Volume 4, Dordrecht: Reidel, 2001.
ISBN 1-4020-0139-8
www.kap.nl/series.htm/HALO

A logic of actions.

[Kent] William Kent, Data and Reality
www.1stbooks.com

Clear.

[Mates] Benson Mates, Elementary Logic, New York: Oxford University Press, 1972. ISBN 0-19-501491-X

A classic text (in case a reference will be useful).

[MOF1] Meta Object Facility (MOF) Specification, Needham, Massachusetts: OMG, 2000.
formal/00-04-03

[MOF2] Joint Meta Object Facility (MOF) Revised Submission, Needham, Massachusetts: OMG, 2002.
ad/2002-05-10

The MOF submission which we support

[Odell] James Odell and James Martin, Object Oriented Methods: A Foundation, Englewood Cliffs: Prentice Hall, 1995. ISBN 0-13-905597-5

Discusses the basic concepts needed for object modeling.

[ODP] X.900 Reference Model of Open Distributed Processing IS 10746, Geneva: ISO, 1995, 1996
www.community-ML.org/RM-ODP/

A standard and widely used set of concepts.

[Padulo] Louis Padulo and Michael Arbib, System Theory—A Unified State-Space Approach to Continuous and Discrete Time Systems, Philadelphia: W.B. Saunders, 1974.

A standard on state.

[Ramackers] James J. Odell and Guus Ramackers, "Toward a Formalization of OO Analysis," in James J. Odell, Advanced Object Oriented Analysis & Design Using UML, Cambridge: Cambridge University Press, 1998. ISBN 0-521-64819-X
Journal of OO Programming, July 1997.

Work towards the goal that: "Everything must be clear and unambiguous."

[Reenskaug] Trygve Reenskaug, P. Wold, O. A. Lehne, Working With Objects : The Ooram Software Engineering Method: Prentice Hall, 1995.
ISBN 0-134-52930-8

Role models.

An excellent introduction to how to specify a modeling language.

[Rumpe] David Harel and Bernhard Rumpe, Modeling Languages: Syntax, Semantics and All That Stuff-Part I: The Basic Stuff, Technical Report MCS00-16, Rehovot: Weizmann Institute, 2000. **www4.informatik.tu-muenchen.de/papers/HR00.ps.gz**

Elements of software architecture

[Shaw] Mary Shaw and David Garlin, Software Architecture, Upper Saddle River, New Jersey: Prentice-Hall, 1996. ISBN 0-13-182957-2

A careful and thorough deconstruction of the ontology we use as the foundation of UML 2. (See [Bunge] above.)

[Smith] Brian Cantwell Smith, On the Origin of Objects, Cambridge: The MIT Press, 1996. ISBN 0-262-69209-0

Several of the problems with UML 1 associations.

[Stevens] Perdita Stevens, "On associations in the Unified Modelling Language," Proceedings of the Fourth International Conference on the Unified Modeling Language, Berlin: Springer, 2001 LNCS 2185 ISBN 3-540-42667-1

Components.

[Szyperski] Clemens Szyperski, Component Software, Harlow: Addison-Wesley, 1997. ISBN 0-201-17888-5

The classics on the axiomatic method and formal semantics.

[Tarski] Alfred Tarski, Introduction to Logic and the Methodology of the Deductive Sciences, Oxford University Press, 1936. ISBN 0-486-28462-X

Alfred Tarski, Logic, semantics, metamathematics: Papers from 1923 to 1938, London: Oxford University Press, 1956. ISBN 0-915-14476-X

An analysis of inheritance.

[Wegner] Peter Wegner and Stanley B. Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like," Proceedings of ECOOP'88 (Oslo, Norway, 1988), Berlin: Springer, 1988. LNCS 322

Another standard on state.

[Zadeh] Lofti Zadeh and Charles Desoer, Linear Systems Theory—The State Space Approach, New York: McGraw-Hill, 1963. ISBN: 0-882-75809-8

Some peeks into reality.

[Zave] Pamela Zave and Michael Jackson, "Four Dark Corners of Requirements Engineering," ACM Transactions on Software Engineering and Methodology, New York: ACM 6:1, 1-30, January 1997 **www.acm.org/pubs/contents/journals/tosem/1997-6/#1**