Response to UML 2.0 Request for Information

Submitted by the precise UML group

© December 1999

Tony Clark
Andy Evans
Robert France
Stuart Kent
Bernard Rumpe

17 December, 1999

1 Introduction

This submission is a response to the Object Management Group's request for information regarding version 2.0 of the Unified Modeling Language. The response is organised into four main sections, in line with the objectives of the RFI. These are:

- strengths and weaknesses of UML 1.3
- determination of major revision
- comparison with roadmap recommendations
- recommendations for revisions

The response is primarily focussed on the definition of the UML semantics, and on the object constraint language. This coincides with the group's aims, interests and expertise.

> Just so that there is no confusion, *semantics*, here, refers to the *meaning* of a language (in this case UML) not to behavioural constraints that can not be expressed e.g. on a class diagram, so need to be expressed e.g. in OCL.

A draft paper is provided in support of the main response. This provides concrete examples of our proposed approach.

1.1.1 The pUML group

The precise UML (pUML) group was formed in 1997 to provide an open forum for researchers in industry and academia who are interested in developing the UML as a precise, i.e. formally defined modelling language. Since then the group has been actively involved in encouraging collaboration and carrying out research at an international level. This has been achieved through the organisation of a number of workshops on the semantics of UML and by providing an internet forum for discussion on matters relating to the precise features of UML.

Members of the pUML group have extensive experience of applying UML in practice and investigating its semantics. Experience relating to its application includes consultancy for major users of UML, including Nortel Networks, British Aerospace Plc., Siemens, Daimler-Chrysler, and BMW. Investigation of the UML semantics is documented in over 30 published papers and through a number of organised workshops and meetings. These describe work relating to the completeness of the UML semantics, formalisation of UML using specification languages such as Z, Object-Z and discrete mathematics, the development of proof and refinement techniques for UML models, and the development of visual constraint languages for UML.

Further details of the group's activities, and other online resources, can be found at the group's web-site (http://www.cs.york.ac.uk/puml).

2 Strengths and Weaknesses of UML 1.3.

2.1 Semantics chapter

2.1.1 Strengths

2.1.1.1. Architecture of semantics

The meta-modelling approach has proved to be an excellent medium for describing syntax and well-formedness rules of the language. The fact that it is written using UML has the benefit of making it accessible to non-technical readers.

There has been some attempt to organise the semantics into sensible groupings using packages. We think this is the right approach, but disagree with some of the detail. See Section 3.1 for our reasons, and how we would go about architecting the meta-model.

2.1.1.2. Use of OCL

OCL has proven itself in the definition of well formedness constraints and it, or something similar, has an important role to play in defining the UML semantics.

2.1.1.3. Semantic richness

The English part of the semantics document, which describes the meaning of the notations whose abstract syntax is described by the meta-model, introduces a rich variety of ideas and concepts, and provides some important insights.

2.1.1.4. Some formalisation of semantics

There has been some attempt to make precise some aspects of the semantics in the core behaviour package. See section 2.1.2.3 for more detail on this.

2.1.2 Weaknesses

2.1.2.1. Syntax

Essentially, only an abstract syntax has been defined in the meta-model, though there are hints at a concrete syntax, e.g. the presentation of a model element.

> A concrete syntax comprises the rules on how to put together boxes, lines etc. that make up a diagram. An abstract syntax talks in terms of classes, associations etc. that those boxes and lines represent. For more detail see Kent et al. (1999a) and/or section 3.2.

The (informal) definition of the concrete syntax is not given in the semantics document, but in the UML notation guide. There is no written down mapping between the concrete syntax and the abstract syntax. The idea that you can leave this to tool vendors is flawed, for two reasons:

1. Without making this mapping explicit and precise, there is no way of checking that a tool conforms to the language specification, i.e. that it produces the correct expressions in the abstract syntax for a particular concrete syntax expression.
2. Making this mapping explicit and precise is a good way of checking that the abstract syntax does really support the concrete syntax, i.e. that the concepts in the abstract syntax are the right concepts. Conversely, that the concrete syntax is appropriate for expressing the required concepts. For example, we note, in the current meta-model that there are concepts of *association end*, *attribute*, and *query operations*. When modelling at the analysis level, the distinction between these three is irrelevant: there is a single underlying concept, that of (public) accessor. Closer to the implementation, association ends and attributes are usually interpreted as private storage; query operations are interpreted as public accessors. This suggests that the concepts in the abstract syntax should be *(public) accessor* and *(private) storage*, and that the same concrete syntax may be mapped to different abstract syntaxes depending on the modelling context. Making the mapping between abstract and concrete

syntaxes explicit and precise is likely to identify cases such as this (some of them may be quite subtle) and get them fixed.

3. There are already areas where UML provides multiple concrete syntaxes for the same abstract notions. The most well known is sequence versus collaboration diagrams. Making the concrete syntax, and its mapping to the abstract syntax explicit will make the relationships between different concrete syntaxes more clear, which will, amongst other things, support automated translation.
4. Providing an explicit and precise mapping requires, of course, a method for doing it. The same method can be reused to provide, in a systematic way, alternative, domain specific, concrete syntaxes for the same concept. This is one way in which the stereotype mechanism is used (Berner et al., 1999), and providing such a method is equivalent to stating the semantics of this use of the stereotype mechanism.

2.1.2.2. Precision (lack of)

To define a semantics requires (at least) an abstract syntax, a semantics domain and a relationship between the two to be defined (see Kent et al. (1999a) for more details). In the semantics document, the abstract syntax is defined using a meta-model approach (class diagrams + OCL constraints), the semantics domain is English, and the relationship between the two is also expressed in English.

Thus the semantics document is not a precise or formal description of the language.

A precise description of the language is important for reasons explained in the following sections (adapted from Kent et al. (1999a)).

Benchmarking
A precise semantics provides an unambiguous benchmark against which a developer's understanding or a tool's performance can be measured: Does the developer use the notation in ways which are consistent with the semantics? Does a tool generate code as the semantics would predict, or does it check the consistency of a model in accordance with the semantics?

Machine Processing

For a machine to process a language, that language must be defined precisely. If it is to perform semantics-oriented tasks, then its semantics must be defined precisely. Examples of semantics-oriented tasks are: model simulation or (partial) execution; checking that different views on a model (class diagrams, invariants, state diagrams, sequence diagrams, etc.) are consistent with one another; checking that the behaviour of a superclass is preserved in a subclass; and so on.

> Is should also be pointed out that different semantics can be constructed to suit different purposes. Thus a semantics which has been constructed with the sole purpose of laying down the definition of a language, may not be appropriate for mechanisation. If alternative semantics are constructed, then it is important that they are proved to be equivalent.

Establishing properties of the syntax

Some important, but nevertheless largely neglected, issues in defining semantics are wrapped up in the question: Is the semantics mapping appropriate? Briefly, a semantics mapping is appropriate if it supports the reasoning or transformation rules on the syntax that one would expect. For example, I would want to be sure that a class diagram with an association end with cardinality 1 could be transformed to a class diagram with no cardinality on that association end plus an OCL constraint stating that the size of the set resulting from navigating that association end was always 1. For more examples see France et al. (1998) and Evans (1998).

Representing the semantics precisely allows one to check that the semantics is sound and complete with respect to the reasoning rules. If one is confident that the reasoning rules are correct, then this increases one’s confidence that the semantics is correct. Conversely, if one is confident that the semantics is correct, then this increases one’s confidence that the reasoning rules are correct.

Having precise semantics and reasoning rules gives the following advantages:
- It provides the user of the notation a choice of whether or not to deal explicitly with the semantics. To draw an analogy, having a Chinese-English translation at hand, I begin to learn Chinese words, but also how to build useful Chinese sentences. When I have learnt to do this I can directly deal with Chinese, without any translation to English anymore.

- The syntactic transformation or reasoning rules may be used in a systematic way by a modeller to establish whether a model has certain properties or not.
- The reasoning rules can be used in a constructive way, to develop a new model from an existing one. For example, one could have proven transformation rules that can be used to develop a concrete model from a more abstract one (refinement), which will guarantee that the concrete model behaviourally conforms to the abstract one.
- If the rules are precise, they can be mechanised, saving the modeller from laborious work. Even the choice of which rule to apply in a particular context can be mechanised in some cases (known as theorem proving).

Complexity and clarity

Making the semantics precise is likely to lead to a reduction in the number of concepts in the meta-model. This reduction will come about by making clear and exact the differences and similarities between what are currently identified as different concepts. In particular, there seem to be a large number of concepts in the behaviour model, some of which seem to overlap. Specifically, it is not clear why events (in state diagrams) and actions are distinct? What in their semantics make them different? Surely a CallAction is no different semantically to a CallEvent, etc.?

On the other hand, experience of constructing semantic models for systems significantly simpler than UML suggests that the *structure* of components in the semantic domain can be surprisingly complex - there may be a set of minimal concepts, but the work together in sophisticated and interesting ways. English is at best cumbersome and at worst inadequate for describing such structures.

The arguments against defining a precise semantics

Defining a precise semantics is hard work and time consuming. If the only purpose of the semantics is to show developers (roughly) how to use the notation, it could be argued that a precise semantics is not necessary. This is probably not (or will not be in the near future) the case with UML.

A precise semantics can also be hard to read, especially if it is written in a hard to understand mathematical language. We believe this can be mitigated in three ways: (1) use a notation that will be readable by general engineers; (2) develop a reference implementation of that semantics that can be used actively by developers and tool builders to gain a deep understanding, rather like how a programmer gets to understand a programming language by writing programs, compiling them (checks that their programs are syntactically and type correct) and observing the effects when they are executed (checks their understanding of the semantics against what actually happens); and (3) write an informal semantics to complement, not replace, the precise one. It is important to stress that (3) is an exercise in *explaining* the semantics, it is not that effective in defining the semantics.

2.1.2.3. Unclear distinction between syntax (concrete and abstract) and semantics

Although we have stated that the larger part of the semantics has been defined in English, there is evidence that some of the semantics has been represented in the meta-model using a denotational approach.

Specifically, in the common behaviour package notions of attribute link, link, linkEnd, instance and stimulus have all been introduced. As far as we can tell from their English descriptions, these are all concepts from the semantic domain: they represent particular instances of behaviour. Furthermore, there are even some OCL constraints which start to make precise the semantics mapping, by stating what constitutes a valid instance of some specified behaviour, e.g. what constitutes a valid instance of an association.

Unfortunately, that this has been done is not made clear in the semantics document, as evidenced by the fact that concepts in the abstract syntax and concepts from the semantics domain, all subclass from model element: surely, there should be a distinction between model element (instantiable) and instance of model element (instance).

We believe the semantics would be much easier to read, if packages were used to clearly distinguish between what is abstract syntax, what is semantics domain, what is the mapping between the two, and similarly to separate concrete from abstract syntax. However, this will require more sophisticated model management constructs, such as those alluded to above.

As an aside, we have done some work (Kent et al., 1999b, Evans and Kent, 1999) which take this approach to defining semantics much further than in the current meta-model.

2.1.2.4. Testing the semantics

The semantics model remain untested. As far as we are aware, there is no tool that claims to be UML 1.3. compliant. Experience from software development suggests that it is only when a model is made precise and checked by a machine (e.g. by implementing, compiling and executing it as a program) that one establishes whether it is appropriate and correct. Experience also suggests that, in most cases, the model is found to be incorrect, inappropriate and incomplete in, at least, some areas. We suspect that the UML meta-model is no different.

2.1.2.5. Conformance of tools against semantics

Currently, the only way of checking that a tool or method conforms to UML definition, is by inspection: compare the source code/test results of the tool with the semantics definition on paper. This is certainly infeasible without having a method for systematically transforming the source code/test results to something that can be compared directly with the semantics definition, and probably infeasible without automating this mapping and the checks in some way.

2.1.2.6. Behaviour

As indicated earlier, there seems to be an over abundance of concepts to represent behaviour (why both action *and* event).

There are also some concepts missing. For example, Catalysis (D'Souza and Wills, 1998) provides an interpretation of state diagrams to pre/post conditions (transitions), and dynamic classes (states), which is particularly useful when building analysis or business models. This requires (a) pre/post conditions to be included in the model, supported by OCL, which might also lead one to reflect further on the relationship between operation and method, (b) a characterisation of state diagrams which permits this mapping.

Indeed, the latter suggests (to give another example of 2.1.2.3) that perhaps the state diagram concrete syntax can be mapped two ways into the abstract syntax, and further, that having separate abstract syntax to represent a state diagram may be unnecessary - actions & pre/post conditions would probably be enough to capture the two main interpretations.

Finally, Kent et al. (1999a) details some specific problems with the definition of the behaviour as it currently stands in version 1.3.

2.1.2.7. Model Management

This is a weakness of the semantics that, perhaps, reflects the general lack understanding in the community of notions such as refinement, realisation, model templates, extension of models, composition of models, reuse of models, etc. We note that Catalysis has some interesting ideas in this area, which it manages to accommodate within the existing concrete syntax.

There are three areas to which we draw your attention:

Package imports. This is conceptually similar to e.g. the Java model of package imports, which we believe to be too weak. In particular it does not allow a model to define one aspect (e.g. operation A) of a model element (e.g. a class), another model to define another aspect (e.g. operation B), and then for both these models to be imported into another model, in which the two aspects of the model element are merged together. Such a device is required (as will be shown in 3.1) to structure the meta-model itself, especially one that includes semantics and aims to support profiles. D'Souza et al. (1999) puts forward a similar argument.

Catalysis proposed a much stronger notion of package imports which, effectively, does for packages (models) what a sophisticated inheritance mechanism (such as that implemented in Eiffel) does for classes. In particular, it allows multiple imports with renaming and merging. Catalysis & D'Souza et al. (1999) also shows how this mechanism can be used to support model templates, and, to a limited extent, patterns.

Model/System Model. The definitions of these two concepts are vague and unclear. They seem to be saying that a model is a particular kind of package that represents an "abstraction" of some physical system, and that a "system model" is a model that contains a collection of models of the same physical system, together with all the relationships and constraints between model elements contained in the different models. No where is it defined what a physical system is and how this differs from an abstraction of the physical system.

Then there is the concept of **Subsystem** which “represents a behavioural unit in a physical system” whatever that is. It seems that a subsystem is split into two parts, its specification and its implementation.

There is a simpler, yet more general, way of viewing things:

- There are models.
- Using a more sophisticated notion of model imports (see above), models can be built from other models.
- There is a concept of refinement/realisation between models, that allows an abstract model to be realised down onto a more concrete one. The refinement can, itself, be represented as a model which imports from the abstract and the concrete model and adds new stuff (e.g. associations, constraints, sequence diagrams, etc.) which say how any instance of the abstract model relates to instance(s) of the concrete model. We do not care which of the terms refinement or realisation is used. Also note that refinement/realisation only needs to be between models. The relationship only exists between model elements as part of the construction of the relationship between models. Our experience suggests that the model element relationships are expressed using things like associations, sequence diagrams and so on.
- Actually, refinement/realisation just conforms to *a particular pattern of modelling*. As we have indicated, this relationship can be expressed as a model. There may be other generally useful modelling patterns that it is worth identifying and codifying into the meta-model.

Thus, different models of the same physical system will either be at the same level of abstraction, or they will be at different levels of abstraction. If they are at the same level of abstraction, then they can be merged through package imports to produce a complete model *at that level of abstraction* of the system. If they are at different levels, then they can be related by refinement/realisation.

Thus a system can have models at different levels of abstraction, and each of those models may be constructed from some smaller model pieces using package imports (note a model piece is itself a model). Since refinements are also models, a multi-level model of a system (where, here, level refers to a level of abstraction) may be constructed by package imports from multi-level model pieces. Model pieces may be predefined, reusable chunks.

We have developed examples within the telecomms domain (not yet published) showing how this all works. It expands on ideas from Catalysis.

> Catalysis draws many of its ideas from a large body of work relating to the composition of models in Algebraic Specification languages such as OBJ and CLEAR (Goguen). Work on parameterisation from the ML community where they have functors and packages is also appropriate (ML has a calculus of package composition that determines the meaning of importing, composing, diamond-importing etc.). As is work on scope issues from the functional language community.

The above suggests that a System Model is a multi-level model, that may be constructed by importing from a number of refinements (which are themselves models), and the abstract and concrete models of those refinements may be built up from other models via package imports and so on.

A subsystem is just two models, one which is a specification of the system, and one which is the implementation. The implementation imports from the specification. That latter statement may seem surprising, but a little reflection should convince you that you can’t write an implementation of a set of interfaces, which does not itself include those interfaces. For example, if one defines a set of interfaces in Java that then get used in some larger system, the implementation of those interfaces must include them (when one defines a class A which implements interface IA one has to write “class A implements IA”, and this won’t compile unless IA is also supplied.

Alternatively, one might identify a model comprising just the declarations of interfaces that are then imported into a specification model, that adds things like pre/post conditions etc., and into an implementation model that adds things like classes, with implemented methods. The subsystem could then be the model constructed by importing both the specification and the implementation models.

As for the issue of instantiation (seemingly only subsystem models can be instantiated), there is, we think, a misunderstanding here. If you accept what was said earlier about semantics domains, all models can be instantiated - one can construct instances (e.g.visualised through object diagrams (snapshots), filmstrips, etc.) in the semantics domain which conform to the model.

There is another notion of instantiation, which distinguishes between those models which are executable (one can create instances inside a machine and run them) and those which can't. We are willing to believe that there is a subset of UML which is executable (lets call it the Unified Programming Language - UPL), and only models written using this language would be instantiable in the sense just described.

2.1.2.8. Architecture & Profiles

The current version of UML provides a large number of modelling facilities. Because of this, there is a danger of becoming overloaded with too many concepts, many of which are not widely used except in very specific circumstances. For example, the definition of class diagrams (static model elements) supports a wide variety of facilities for expressing constraints. In practice, these facilities are rarely used, or may be used inappropriately.

There has already been some attempt to architect the meta-model into packages so that pieces are brought in stage by stage. The limited power of the current UML package imports has meant that this is relatively coarse grained. It is not possible to define pieces of a class or in one package, then add more pieces in another package that imports it (and that adding might be by importing another package that supplies other pieces of that class). Thus when a concept is introduced in a particular package (e.g. the concept of *Operation* in the core package), one is forced to introduce all the different facets of that concept, even if they are not relevant at that stage. If all we are interested in is a subset of UML for specification modelling, then we probably do not need to know about whether an operation is abstract or not, whether it is a leaf or a root, and, or even what form of concurrency it uses. In fact, the only that is relevant for this purpose is its *specification* attribute.

This leads into the notion of profiles. It should be possible to have a core profile (introducing classes, associations, operations, etc.), then for an implementation modelling profile to import from the core, adding distinctions such as whether an accessor can be stored or not (see Section 2.1.2.1), whether an operation is abstract, isRoot, isLeaf and so on. The specification modelling profile might import from the core profile and add in notions of pre/post conditions and invariants.

In summary, to support a finer-grained architecture for the UML meta-model, which is also required to support profiles, one must have or more sophisticated notion of package imports, such as that outlined in Section 2.1.2.7. For more details on this approach see e.g. D'Souza et al. (1999). Thus, if one accepts this approach (and we do) it is clear that fixing some of the model management problems is essential, if one is going to handle profiles and have finer-grained control of the architecture of the meta-model.

2.1.2.9. Change and revision of the language

The current architecture is unlikely to support graceful refactoring and revision of the meta-model language definition. This drastically increases the vulnerability of the notation to extinction caused by changes in trends and emerging technology.

Providing a finer-grained architecture and a profile mechanism (as described above) significantly increases the likelihood of being able to refactor the language definition in a graceful fashion.

2.2 Object Constraint Language

A constraint language for UML is important because:

- It has proved essential in order to define the UML language using the meta-modelling approach
- It is important for some forms of modelling, specifically those where the desire is not to build an operational, executable model, but to build a declarative, possibly under-determined model. Examples include: business modelling, where it is used to express business rules; software specification, where it is used to make precise invariants, pre and post conditions; modelling telecomms networks and services, where it is used to express behaviour and constraints in a declarative fashion. Members of the pUML group have worked and are working with industry who are using the OCL with UML in anger.

2.2.1 Strengths

2.2.1.1. Standard syntax

OCL provides a standard syntax for writing constraints in a reasonably precise way.

2.2.1.2. Useability

OCL is more acceptable to engineers than bare predicate logic.

2.2.2 Weaknesses

2.2.2.1. Integration

OCL is not fully integrated into UML (e.g. it is used to describe, but is not itself described in the semantics document). This leads to some confusion about its status (is it part of UML?).

2.2.2.2. Useability

Experience suggests that OCL is an extremely powerful tool and essential for engineering large industrial software systems. The lack of a complete, rigorous and flexible model for OCL makes it difficult for Software Engineers to realise its potential.

2.2.2.3. Variation in concrete syntax

Related to point 2, there is no defined route for defining different concrete syntaxes, for use in general or for use in specific domains. For example, it would be useful to have a visual form of the language which allowed constraints to be built up using a series of network diagrams representing prototypical network configurations that are allowed or not allowed; it would be useful to have a variant of the syntax which used the symbols of "my favourite programming language" (e.g. Java) for the logical operators.

2.2.2.4. Semantics

OCL is currently informally and incompletely defined. Informality precludes the development of tools such as model checkers, type checkers and inter-model consistency checkers. Incompleteness leads to confusion in areas such as the context of an OCL expression and how it relates to overall system execution.

3 Determination of whether a major revision is required

A major revision is required if the deficiencies are such that the only way that they can be accommodated is through a substantial refactoring or extension of the existing definitions. We think this is the case for both the areas on which we have focussed: the constraint language and the semantics. The following lays out the arguments for this, in particular highlighting existing work which could be adapted to address the weaknesses.

Whether major revision is done through RFP's, or some other mechanism, is discussed in Section 5. One issue to consider is that, in order to address some of the roadmap recommendations (see Section 3.1.4) the basic approach to (architecture of) semantics and language definition needs to be settled. We do believe that addressing the weaknesses of the semantics will require significant revision to the approach currently taken to defining the language. On the other hand, we believe that this can not be addressed through an RFP, simply because this will delay the issuing of other RFPs. We have proposed, in Section 5, an alternative mechanism.

3.1 Semantics

It should already be clear from section 2.2.2 that a major revision is required. However, in order to ascertain more accurately how different the result of that revision will be from the current definition, and, therefore, the most appropriate way to plan that revision, requires a more detailed discussion of what is required and what might be done to address the various weaknesses.

In summary:

- The language definition is structured into profiles, based on a kernel library of language definition tools and components. A profile is itself a definition of a language that may specialise and/or extend other profiles, and incorporate components from the kernel library.
- Each profile is organised into abstract syntax, semantics domain and a satisfaction/denotation relationship between the two (see Figure 1). Both abstract syntax and semantics domain may have many concrete representations.
- A subset of UML, the meta-modelling sublanguage, can be used to characterise all aspects of a profile and the kernel library. This language includes class diagrams, a constraint language, packages (to represent models), an enhanced version of package imports, and a notion of package realisation.
- Language definition modelling is done concurrently with the systematic construction of a reference implementation (a tool) of the models produced. The reference implementation can be used (a) to test and explore the language definition models themselves, (b) by (a), to help users to explore and understand the form and meaning of the language being defined, and (c) to support automatic testing of conformance of CASE tools to the language definition.

3.1.1 Language definition essentials

The general architecture for language definition, described in this section, is adapted from programming language theory and definitions of logics. It addresses points made in sections 2.1.2.1, 2.1.2.2, 2.1.2.3, 2.1.2.8, 2.1.2.9, as follows:

- 2.1.2.1: makes the distinction between abstract and concrete syntax
- 2.1.2.3, 2.1.2.2: clarifies the relationship between syntax and semantics; provides a foundation on which to build a precise semantics; incorporates the notion of proof rules.
- 2.1.2.8, 2.1.2.9: provides a foundation on which to build a finer-grained and more flexible language architecture, thereby increasing the chance of graceful revision; provides the basis for the architecture of profiles.

This section builds on the 'architecture' for a language definition sketched in sections 2.1.2.1 and 2.1.2.2. There, five components to a language definition were identified: a concrete syntax, an abstract syntax, a semantics domain, a mapping from concrete to abstract syntax and a mapping (the semantics) from abstract syntax to semantics domain.

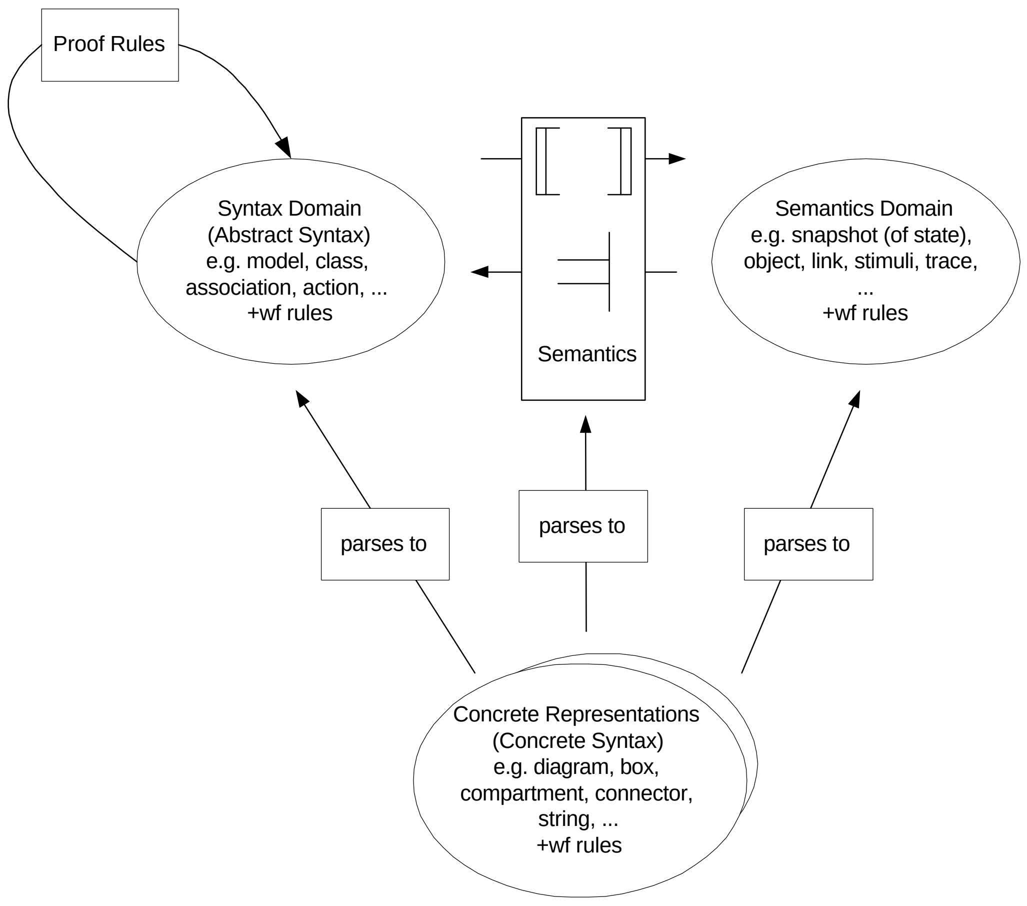


Figure 1: Components of a Language Definition

These components are illustrated by Figure 1, which also suggests examples of the concepts that might appear in each part. A more uniform terminology has been adopted, and another component has been added:

Terminology
A language is defined by its *syntax* AND *semantics*. The semantics is given by a mapping from the syntax (domain) to a *semantics domain*. Both syntax and semantics domains may have *concrete representations*. There may be many concrete representations for any particular element from either domain. Some concrete representations may be used to represent elements from both domains. For example, in the UML, a sequence diagram may be used to specify general behaviour of operations (it is a concrete representation of some 'sentence' of the UML), or it may be used to represent a specific trace of that behaviour (it is a concrete representation an element in the semantics domain). A set of concrete representations of elements from the syntax domain (often referred to as *expressions* and *sentences*) has traditionally been called a *concrete syntax*. The term *abstract syntax* has then been used to distinguish the syntax domain from concrete syntax. We will continue to use these terms. It is also worth stressing that every component of the language definition has *well-formedness (wf) rules*. For example, the UML 1.3. meta-model has a number of these rules expressed as constraints in OCL.

Additional Component
Transformation or *proof rules* map sentences of a language to (simpler) sentences of that language. They can be used to deduce new properties from a set of existing properties, where a property is expressed as a sentence.

We have deliberately not rendered Figure 1 using UML notation because we do not wish the reader to fall into the trap of thinking that this architecture is only for meta-modelling in UML. The components and their relationships could be described in a number of languages. We have used an invented language to identify the main components and their basic relationships; the details could be described using plain set theory/predicate logic.

The relationship between concrete syntax and concepts is well-understood by most people. Parsers implement this relationship.

The relationship between abstract syntax and semantics domain, is probably less well-understood. The basic idea is that the concepts in the semantics domain have a more direct correspondence to things in the world. Typically, this means they represent examples or instances of behaviours in the world - the conceptualisation of the state of memory in the computer or writing on bits of paper in a business (snapshots), which also requires the conceptualisation of individuals (objects) and connections between individuals (links), the conceptualisation of the transitions from state to state through time (traces), the conceptualisation of invocations of actions/events at particular points in time or in particular states (stimuli).

If you really wish to explore the philosophy of all this, take a look at Montague (1974).

The relationship between abstract syntax and semantics domain is one of *denotation* and *satisfaction*. What do expressions in the language denote in particular contexts (in particular states, at particular points in time)? What does it mean for (a suitably coarse-grained) instance to satisfy a sentence of the language (*sentences* of the language denote truth values - they are true or false in particular states or for particular traces). For UML, it is appropriate to ask, for example, what set of objects does a class denote at a particular point of time or in a particular snapshot, but it is not appropriate to ask whether a set of objects satisfies a class. However, it is appropriate to ask whether a snapshot or trace satisfies a model; alternatively, whether a model denotes true or false in the context of a particular snapshot or a particular trace.

The symbols $[[X]]$ and $x \models X$ are those typically used in mathematics to represent the denotation of a language element X, and the satisfaction relationship between an element of the semantics domain (instance x) and sentence (X).

Proof rules are rules which allow one to deduce properties of a sentence in the language by ascertaining what other sentences can be derived from it through transformation. Proof rules must be *sound* - all instances which satisfy the derived sentence must also satisfy the originating sentence. They may also be *complete* - every sentence which, semantically, follows from a sentence can be derived from that sentence by the proof rules.

Figure 1 shows the semantics of a language as a mapping to a semantics domain, which has certain characteristics. Proof rules provide a way of ‘revealing’ the semantics, whose details may be hidden, by indicating what sentences in a language follow from other sentences. An alternative approach to semantics is a translational one: expressions in the language are translated into another language which has a semantics, or is, perhaps, more widely understood than the language being translated. For example, one could give a semantics to UML by translating into a language such as Z. This is different to *characterizing* the language itself and/or semantics domain in Z.

The translational approach is illustrated by Figure 2. Language B might be UML, Language A might be Z, plain predicate logic, or even a programming language.

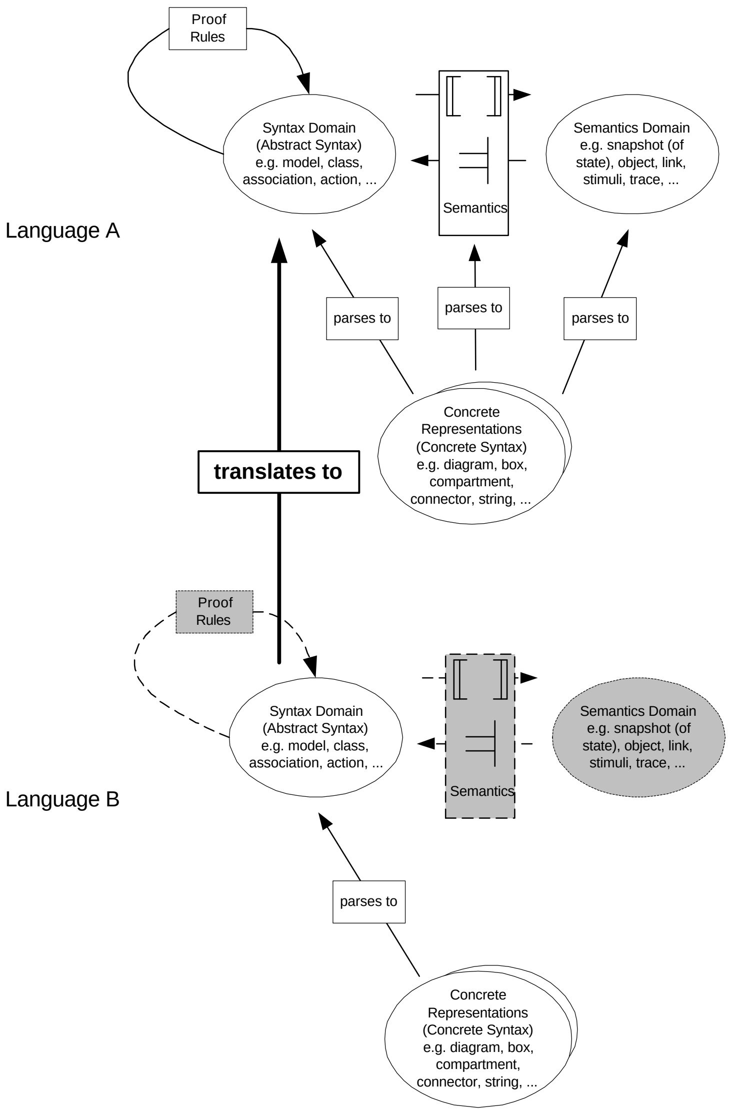

Proof Rules
Syntax Domain (Abstract Syntax) e.g. model, class, association, action, ...
Semantics
Semantics Domain e.g. snapshot (of state), object, link, stimuli, trace, ...
Language A
parses to
parses to
parses to
Concrete Representations (Concrete Syntax) e.g. diagram, box, compartment, connector, string, ...
translates to
Proof Rules
Syntax Domain (Abstract Syntax) e.g. model, class, association, action, ...
Semantics
Semantics Domain e.g. snapshot (of state), object, link, stimuli, trace, ...
Language B
parses to
Concrete Representations (Concrete Syntax) e.g. diagram, box, compartment, connector, string, ...

Figure 2 - Translational Approach to Semantics

Expressions/sentences of Language B are translated into expressions/sentences of Language A. Language A is presumed to have a semantics and, possibly, proof rules. A semantics/proof rules are induced for language B (hence shown in grey) from this translation.

3.1.2 Meta-modelling approach

The meta-modelling approach, described in this section, addresses points made in sections 2.1.2.2, 2.1.2.8, 2.1.2.9, as follows:

- 2.1.2.2: It delivers a language for defining other languages (a meta-modelling sublanguage of UML) which should be more accessible to engineers than plain maths (especially if some more “engineer friendly” concrete syntaxes for OCL can be devised).
- 2.1.2.8, 2.1.2.9: The proposed meta-modelling sublanguage has constructs (packages, package imports, package nesting) that allow language definitions to be constructed piecemeal and from reusable library components. This allows language definitions to be constructed as prescribed in Section 3.1.1, and supports profile definitions (the reader is referred to Section 0 to see how, exactly). It supports graceful revision, because the package structuring mechanisms mean that it is more likely that revisions can be achieved through (behaviour preserving) refactorings and extension.

3.1.2.1. What is the meta-modelling approach? Why use it?

We indicated, in the previous section, that one language that could be used to express the various components of a language definition is plain set theory/predicate logic. What are the alternatives?

Instead of set theory one could use one of any number of mathematical theories which have been developed/used over the years in defining languages, for example: BNF, attribute grammers, etc., well known ways of defining concrete and abstract syntax; domain theory, a useful characterisation of the semantics domain when defining programming languages; labelled transition graphs, used to characterise the semantics domain for process algebras such as CSP.

There are also a number of well-defined calculi which could be used to provide a semantics using the translational approach. Examples are Z and CSP.

Or there is the so-called meta-modelling approach, so far adopted in the UML standard. In this approach, a subset of the UML language is used to define itself and the rest of the language.

> There might be some debate as to whether the distinguishing feature of meta-modelling is that one defines a language using a subset of itself. A meta-model just indicates the level at which a model is perceived in a meta-model architecture. The semantics chapter in 1.3. states that the definition of UML is based on a 4 layer metamodel architecture. This can be related to the previous section (3.1.1) as follows:
>
> - User objects are expressions of the language used to characterise the semantics domain.
> - Models are expressions in the language being defined (UML).
> - The Meta-Model is the language being defined.
> - The Meta-Meta model is the language used to describe Meta-Models (e.g. set theory).
>
> But, there are a number of components of a language definition, as described in the previous section, which do not seem to have a counterpart in this metamodelling architecture. Where is the language used to characterise the semantics domain? Perhaps this also is a meta-model. But then expressions in this language would then be models, according to the architecture, but as far as we can make out they reside at the user objects layer. What about the distinction between concrete syntax and concepts? What about the mappings between the various components (layers)?
>
> There may be other ways to map the architecture described in section 3.1.1 to the four layer meta-modelling architecture. However, whatever mapping is made, it is clear that the concepts in the meta-modelling architecture need to be refined and better delineated for such a mapping to be made. We contend, therefore, that the architecture described in section 3.1.1 provides a richer and better delineated set of concepts than the 4-layer meta-model architecture.
>
> If so, what else can we use the term "meta-modelling" to mean. Well, it seems that in general usage the term is often used to mean that the language is defined using a subset of itself. We have adopted this usage.

There are some advantages to this:

1. The language definition is more likely to be understood by modellers and tool builders, who, we presume, will already understand UML to some extent (at least the meta-modelling sub-language).
2. The mapping from an object-oriented model of the language to an object-oriented program is more or less direct, providing greater confidence that any implementation of the language in a CASE tool conforms to the language definition.
3. The features of object modelling (e.g. polymorphism and delegation) that make it so flexible and useful in modelling a wide variety of systems, bring similar benefits when modelling in the domain of language definition.
4. This is the route already embarked upon for UML, so if a more complete language definition can be provided using the same approach then the transition from version 1.3. to version 2.0. will be much smoother.
5. It doesn't exclude alternative approaches. Anything one can define in the meta-model will have an equivalent external representation, just because the meta-modelling language itself will need an external semantics (e.g. in set theory). Thus the meta-modelling approach provides a framework in which different semantics can be 'plugged in'. If a translational semantics is desired, then the target language can be defined in the meta-model. For example, the abstract syntax of Z can be expressed in the meta-model with little difficulty.

We believe these to be sufficient reasons for adopting the meta-modelling approach. In particular, 5 means that we can always 'escape' to other paradigms if that proves necessary.

3.1.2.2. Mitigating the risks of the meta-modelling approach

There are three risks associated with using the meta-modelling approach:

- Risk of circular definition.
- Risk that the meta-modelling language is not rich enough to define the target language(s), including itself.

The first risk is mitigated by providing an external definition of the meta-modelling language. This could be done by a denotational or translational approach. For UML, the meta-modelling language will be a small subset of UML (see below). A denotational approach could be realised using plain set theory, which would be used to define the syntax and semantics domain, the semantics as a mapping between these, and, if deemed required, a concrete syntax and a set of concrete representations for the semantics domain. A translational semantics could be given based on existing work. France (1999) has already done some work on mapping to Z; Jackson (1999) has a language Alloy which is close, concrete syntax aside, to our proposed meta-modelling sublanguage, and he has defined a mapping into logic for this language, so that it can be run through a model checker. The translation itself need not be described in set theory; it could be embedded in an algorithm in some programming language.

The second risk is mitigated in two ways.

Firstly, by ensuring that at least the language is able to describe itself. We believe a mathematical proof, hinging on an external, denotational semantics, can be constructed demonstrating this.

Secondly, by point 5 made on page 13, which suggests that semantics built using existing mathematical theories can be ‘plugged in’ to the meta-model. This is possible, at least if the meta-modelling language has an external (denotational) definition in set theory. Then any definition of the abstract syntax in the meta-modelling language can be rendered in terms of sets, and these can be used as a starting point for a semantics written using plain math. It may even be possible to have part of e.g. a denotational semantics rendered in the meta-model with other parts expressed external to the meta-model. Of course, it may also be the case that the meta-modelling language is rich enough to render a complete definition of all aspects of UML - without trying, we won’t know.

3.1.2.3. A meta-modelling sublanguage of UML

So what should be in a meta-modelling sublanguage of UML? It needs the following components.

Object structures

A way to describe the structure of expressions in the syntax of a language, and a way to describe the structure of elements of a semantics domain. Sets and relations provide the basic structuring mechanisms in plain Math. In UML, the basic mechanisms are classes and associations. Additional power comes from the ability to relate classes through generalisation and the ability to construct sets of objects by navigating associations and filtering (as you do when writing constraints).

Constraints

A way to constrain the structure of expressions and elements of the semantics domain, so that only certain patterns of structure are admitted. These are sometimes known as well-formedness constraints. In plain Math, such constraints can be expressed using logical statements. In UML, constraints can be expressed using cardinalities on associations, using collaborations, using annotations to class diagrams, and, of course, using OCL. These are just various concrete syntaxes for a single underlying constraint language. The constraint language is not necessarily everything covered by OCL. For example, undefinedness, bags, sequences and real numbers can probably all be left out. Sets, navigation and logic are essential. A well-defined notion of transitive closure (as defined in Alloy (Jackson, 1999)) needs to be included.

Packages

A way to package up different parts of a language definition (abstract syntax, semantics domain, etc.) and relate those packages is required. An enhancement and tightening up of UML’s package mechanisms will suffice. A package is a collection of expressions/sentences of the meta-modelling sublanguage. A package is itself a sentence of the language (it is valid to ask whether a particular instance from the semantics domain satisfies a package). Thus packages can be contained in other packages. This is just the situation in

UML. Package containment essentially provides a useful naming scheme. Thus in Figure 6 the package `AbstractSyntax` actually has the name `AProfile.AbstractSyntax`. Package containment can also be used to control visibilities (Schürr and Winter, 1999).

Additionally, an enhanced version of package imports is required. In Version 1.3., the package imports mechanism is very basic. When a package is imported, the elements in the package that are imported can not be extended or altered within the importing package: any changes are enacted on the original definition. An enhanced version of package imports, such as that defined in Catalysis, does allow elements to be extended and renamed. This gives additional flexibility, but still ensures that the importing package is conformant to the imported package (instances of the importing package are still instances of the imported package, subject to appropriate renaming and hiding of extensions).

In Figure 3 `InstructorAllocation` and `RoomAllocation` both import from the same `ResourceAllocation` package, but via one imports relation `Resource` is renamed to `Room` and via the other imports relation `Resource` is renamed to `Instructor`. In both cases, `Request` is renamed to `CourseDelivery`. The package `CourseAllocation` imports `InstructorAllocation` and `RoomAllocation` with no renaming, and ends up with classes `Instructor`, `Room`, `CourseDelivery` and `Allocation`.

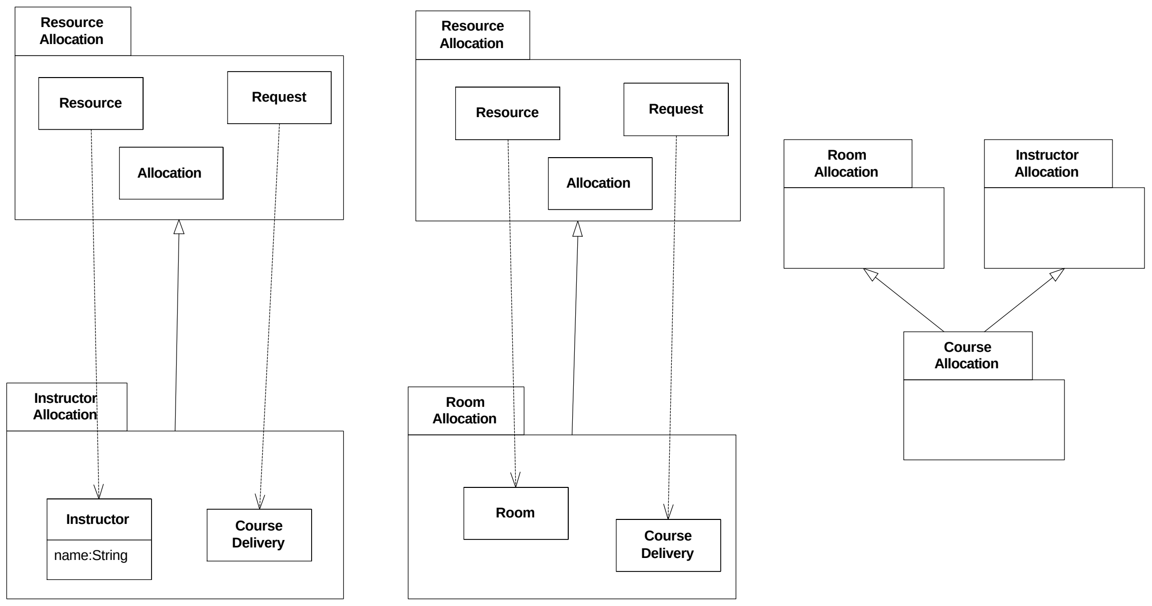


Figure 3 - Package imports with renaming

This relationship allows for fine-grained and powerful structuring of packages in three ways:

- Package elements can be renamed on imports. Thus generic terms can be used in packages intended for reuse, and these can be revised to the specific case when a generic package is imported to a specific context.
- Package elements can be added to in the importing package, without affecting their definition in the package imported. For example, `Instructor` has an attribute that is not in `Resource`. (Though note that it is possible to make elements fine-grained enough, so this can be equated to just adding elements. So the attribute can be treated as a distinct element from the class.)
- Elements which are distinct in imported packages but which are intended to be the same (e.g. `CourseDelivery`, `Allocation`) are treated as the same (they are merged) in the importing package.

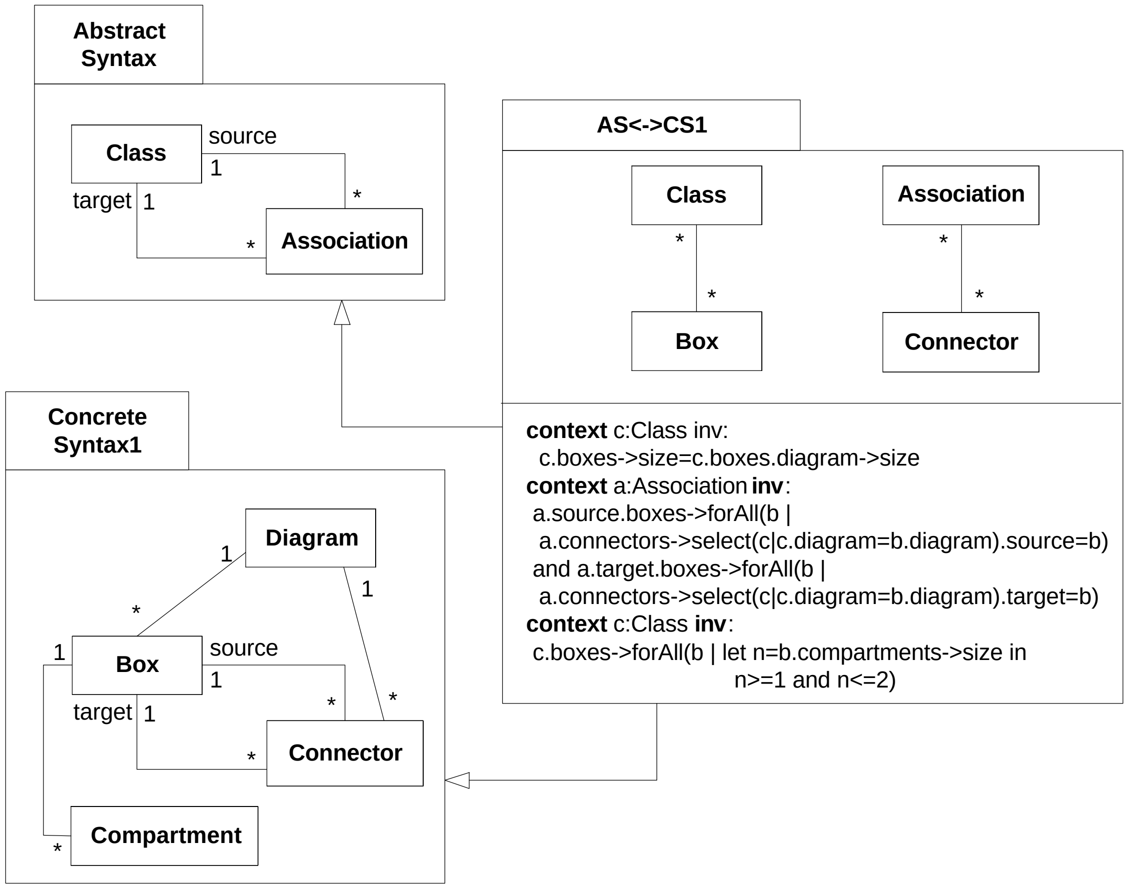
Realisation

Realisation (or refinement) is a relationship between packages, which says how instances (i.e. object configurations) of one package map to instances of another package. Realisation has been defined and used by one of the authors in telecomms modelling to say how service definitions can be mapped onto configuration information on networks. This work is yet to be published, but see the conclusions of Kent et al. (1999b) for a brief explanation. A realisation relationship can be expressed using a model that conforms to a particular pattern. A simple example of realisation is illustrated by Figure 4. The relationship is between two packages, one representing the concrete syntax of a simple language and the other the abstract syntax of the language. The details of the realisation relationship are recorded in a package which imports the concrete and abstract syntax. Specifically:

A class is realised by a box with two compartments, and an association is realised by a connector whose source and target boxes are the realisations of the source and target classes of the association.

There is an assumption here that a class can only ever be realised by a single box and an association a single connector.

- Two associations are included, indicating that a class is realised by a box and an association by a connector.
- OCL invariants are included, recording the fact that a class/association is realised by a single box/connector in any one diagram, a class must be realised by a box with one or two compartments, and that the connector realising the association must connect boxes that realise the classes at the source and target of the association.



```
context c:Class inv:
  c.boxes->size=c.boxes.diagram->size
context a:Association inv:
 a.source.boxes->forAll(b |
  a.connectors->select(c|c.diagram=b.diagram).source=b)
 and a.target.boxes->forAll(b |
  a.connectors->select(c|c.diagram=b.diagram).target=b)
context c:Class inv:
 c.boxes->forAll(b | let n=b.compartments->size in
                              n>=1 and n<=2)
```

Figure 4 - Example of realisation

Thus the package representing the realisation relationship conforms to a pattern, it must obey certain constraints: it is not allowed to add new associations between elements which are only from one of the imported packages; invariants in the realisation package must reference associations between elements of each of the importing packages.

Thus, in the context of language definition, the mappings between components can be expressed as realisations (in this case the concrete syntax is a realisation of the abstract syntax). Although (as we have shown) these can be expressed using other parts of the meta-modelling sublanguage, these mappings are so important to a language definition that they need to be defined as part of the meta-modelling sublanguage. It makes sense, then, to invent a concrete syntax to represent the concept of realisation. The syntax we have chosen mirrors that of association classes (the concepts are quite similar), and is illustrated by Figure 5.

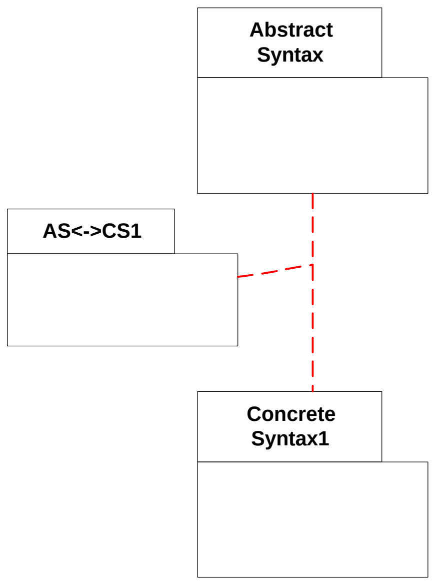


Figure 5 - Syntax for realisation

3.1.3 Architecture of the UML definition (using meta-modelling approach)

The proposed architecture of the UML definition, described in this section, addresses points made in sections 2.1.2.8, 2.1.2.9, as follows:

- 2.1.2.8: The kernel library provides a set of basic reusable library components for building language definitions in general, and the definition of UML, in particular. Each component is a package. Some components may be put together into larger components (packages) through package imports. Different kinds of component are identified, ranging from fundamental, generic pieces, and more specific UML pieces. The latter, in particular, are identified with concrete/abstract syntax and semantics domain.
- 2.1.2.9: A recipe is outlined for putting together a profile from pieces in the kernel library and from existing profiles. This enables graceful revision.

3.1.3.1. A recipe for building a profile

A UML profile will be a definition of a language. It will at least comprise an abstract syntax, a concrete syntax and a mapping between these. It should also provide a semantics in some form. This might be through:

1. translation to another language, for example, another, simpler UML profile, or a different language altogether;
2. through proof rules - we can imagine a profile which extends another, simpler UML profile with new abstract syntax concepts, but then provides rules that say how each of these new concepts can be rendered in terms of the concepts already provided in the simpler profile;
3. through a denotational semantics, by providing a semantics domain and a semantics (a mapping from abstract syntax to semantics domain), and, probably, some concrete representation of elements of the semantics domain;
4. some combination of all three (most likely 2 & 3).

We suppose that the more normal route will be 4, specifically some combination of 2 & 3. The template package for a profile is then given by Figure 6. (See Section 3.1.3.3 for a definition of the dashed line.)

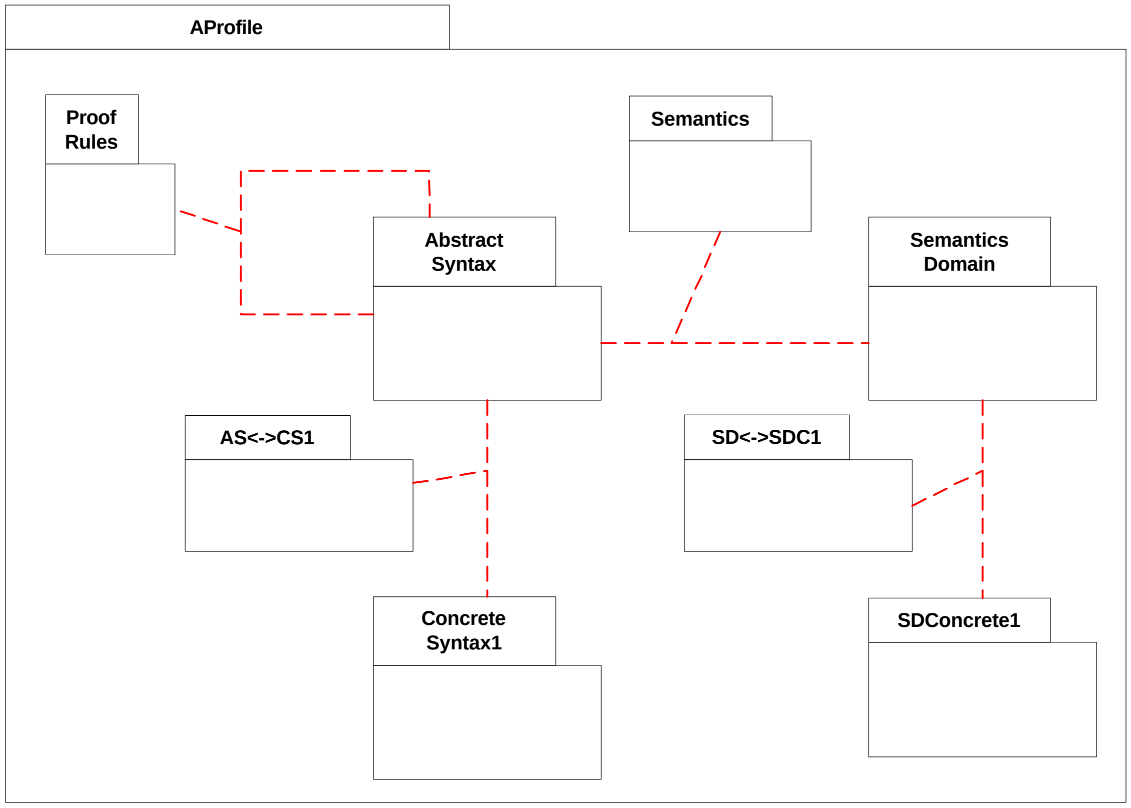


Figure 6 - Template architecture of a profile

Of course, these components in a profile will not be built from scratch every time. The package imports mechanism can be used to build up a profile from existing components, such as those described in subsequent sections. In addition, it may be possible to refactor an existing profile to isolate pieces required for a new profile, without affecting the behaviour of the existing profile.

This is illustrated in Figure 7 where `ProfileA` is refactored to import from a package `CommonToAB`, and then this is imported into `ProfileB` - the profile being constructed.

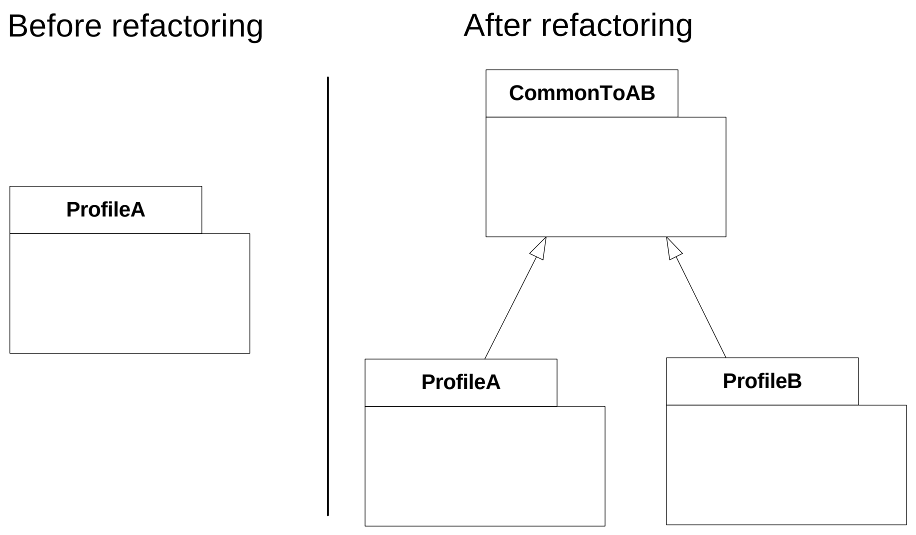


Figure 7 - Refactoring profiles

Finally, the reader is reminded that packages describing each of the components will involve not only class diagrams, but also a number of constraints. For example, an abstract syntax has well-formedness constraints which need to be expressed.

3.1.3.2. A kernel library to support profile definitions

There is no single kernel language. There is a library of components which can be combined and extended in different ways to suit purpose. This library is represented as a network of packages. Some combinations of these components, for example, languages for implementation and specification modelling, respectively, may be delivered as standard profiles (see next sections).

We'll call this library the "kernel".

We imagine at least the following main packages, which we expect to be further subdivided into packages to provide components that are fine-grained enough to build the required profiles.

Fundamentals: concepts such as relationship, relateble, instantiable, instance, container, contained, generalizable, generalisation (special form of relationship), etc.

Abstract Syntax: includes a number of components for defining abstract syntax, probably organised into sub-packages:

> **Model management:** concept of packages and package imports, including renaming and merging of elements in packages. Many of these concepts may be given a very general charecterisation in terms of relationships between containers (there are great similarities between the enhanced version of package imports and class inheritance).
>
> **Statics:** the same concepts used to characterise static structures in the meta-modelling sublanguage, i.e. class, association etc. Essentially a cut-down version of the core package in the UML 1.3. meta-model.
>
> **Constraint language:** the same concepts that underpin the expression of constraints in the meta-modelling sublanguage. This can be sourced from existing work such as Kent et al. (1999b), Richters and Gogolla (1999), Clark (1999).
>
> **Dynamics:** actions, operators for composing actions (in sequence, in parallel etc.), and place holders for specifying actions (pre/post conditions).

Concrete Syntax: concepts such as boxes, compartments, connectors (directed and not), folders, names, labels, paths, graphs, diagrams, charts. This will also probably be organised into sub-packages, though we are not yet sure what those will be.

Semantics Domain:

Statics: concepts such as object, link, snapshot (e.g. as visualised through an object diagram), and operators over these (e.g. merging two snapshots).

Dynamics: concepts such as trace, stimuli (invocation of an action), and operators for composing these (e.g. merging two traces).

Mappings:

Semantics: Some standard mappings (including well-formedness constraints) of elements of the abstract syntax to elements of the semantics domain. This will likely be subdivided into the following packages.

Model Management

Statics

Constraints

Dynamics

Concrete2Abstract: Mappings of concrete syntax to/from abstract syntax.

Model Management

Statics

Constraints

Dynamics

Concrete2SemanticsDomain: Mappings of concrete representations to/from the semantics domain.

Statics

Dynamics

Guidelines for determining which constructs should be defined in the kernel library are given below; we believe that the structure outlined above conforms to these guidelines.

1. The constructs should provide a set of fundamental modelling concepts to support the definition of UML profiles. If possible, it would be desirable to have a library that supports (at least) the definition of a set of standard, general purpose modelling profiles through merging/composition of pieces in the library and specialisation, not through extension (i.e. only adding constraints on package imports).
2. The meta-model profile should just be a specialisation of the kernel library. It should not be necessary to add any new constructs, just, possibly, constrain the ones that are there.
3. All constructs in the kernel should have a precise semantics.
4. Taken together, the constructs should define a conceptual object model that is sufficiently rich to act as a teaching and communication aid (understanding this model should facilitate understanding all other language elements).

3.1.3.3. Meta-modelling profile

The meta-modelling profile will define the meta-modelling sublanguage. It will select (and possibly specialise) components from all but the dynamics packages in the kernel (it is possible to import just a selection of packages from within a containing package). This profile will also be given an external semantics in set theory, in order to ground the meta-modelling approach.

3.1.3.4. Standard, general-purpose profiles

Standard profiles will be (fragments of) general purpose modelling languages that will not be tied to a particular domain or level of modelling.

- Aggregation & richer associations, will select components from the statics and dynamics (sub)packages (probably most of the former, but only a part of the latter).

- Constraint language. Will import all the constraint language packages from the kernel, and will likely provide alternative concrete syntaxes, as prescribed in Section 3.2.
- Model management, specifically refinement. This will extend the model-management, statics and dynamics packages in the kernel, providing detailed definitions of refinement (as patterns of models) not only on the static side, but also on the dynamic side. There may be other patterns of relationship to be defined, e.g. subsystems.
- Some number of more specialised dynamic modelling profiles, with different models of concurrency, real-time etc., and involving or not declarative styles of expressing dynamic behaviour (e.g. pre/post conditions, specification-style state diagrams) and more operational styles (e.g. action languages, operational-style state diagrams). These will extend the dynamic and static basics, and some variants may then themselves be combined with some of the other standard profiles, such as aggregation and model management.

3.1.3.5. Horizontal profiles

Horizontal profiles will be profiles delivering languages targeted at different levels of abstraction, for example:

- Business modelling
- Implementation modelling
- Design modelling
- Software specification

Horizontal profiles will extend (fragments of) the standard profiles and kernel library.

3.1.3.6. Domain specific or *vertical* profiles

Vertical profiles will be profiles delivering languages targeted at a particular domain, and may provide sub-profiles focussed on different levels of abstraction for that domain. For example:

- Insurance modelling
- Telecomms modelling
- Etc.

Vertical profiles will specialise and merge (fragments of) horizontal profiles.

An illustration of the architecture and components of the proposed kernel library and an example profile are presented in the attached Appendix.

3.1.4 Reference implementation

A reference implementation of the UML definition, described in this section, will address points made in sections 2.1.2.4 and 2.1.2.5, as follows:

2.1.2.4: It will provide a concrete test that the meta-model is correct. Language designers will be able to use the tool to explore its concrete/abstract syntax & semantics domain (does it allow the right expressions; are instances of the correct structure), semantics (do instances satisfy sentences as expected; are denotations of expressions those that are expected), and mappings to concrete representations (do the correct diagrams/text appear on the screen).

2.1.2.5: One can conceive of a setup where the language definition is encapsulated as a set of interfaces (e.g. in CORBA) and a test harness is written against those interfaces. An implementation of those interfaces must pass all the tests run by the harness. A reference implementation of the interfaces can be used to develop tests in the test harness. Alternatively, the harness can just run test data through the reference implementation at the same time as test data is run through the implementation be checked, and the results from each compared.

A reference implementation is a tool that, in some sense, embodies the language definition. It must be able to construct expressions of the abstract syntax, elements of the semantics domain, and concrete representations of these, and be able to check that these conform to the well-formedness conditions. It must be able to relate these various pieces according to the definition of the various mappings, and check that the relations respect the constraints on those mappings.

Assuming we have adopted an object-modelling approach to language definition, it should be possible to build a reference implementation for a single profile quite systematically. A profile defines a language. Although it will be structured into packages, this structure can be flattened into a single package comprising classes, associations and OCL constraints. This flattening is required because there is no programming language that directly supports the rich package constructs we have proposed for meta-modelling. A systematic mapping from this kind of package can be constructed into most OOPLs. The main addition required are methods (e.g. constructors) to build and edit expressions, elements of the semantics domain, etc. A key issue to consider is how to deal with invariants. We would recommend turning these into classes with their own check method that is able to check whether or not the invariant holds over the current state of the system.

One reference implementation it would be particularly useful to construct is for the meta-modelling profile. This would be useful because, with a little work on the user interface and persistence, it could be used to help build the UML definition itself. In particular, if the semantics of the meta-modelling profile was built into the tool, the tool could assist with exploring the properties of the language being defined, by looking at particular examples of expressions of that language, their mappings into the semantics domain, so on and so forth.

3.1.5 Migration & backwards compatibility

There remains a question of how to migrate to the proposed language architecture from version 1.4. Here are some guidelines to which we will need to pay attention:

- Any semantics built should reflect common usage and understanding. Essentially, this means formalising the English in the semantics chapter, with the understanding that the semantics work is likely to find errors and ambiguities with that English.
- The concrete syntax needs to be defined precisely, but not changed. The notation guide should be adhered to as closely as possible.
- The proposed architecture is much finer grained, not as monolithic, as the current language architecture. Combined with the enhanced form of package imports, this provides more flexible constructs for evolving the language gracefully.
- We envisage the following first steps:
    1. Select from the existing meta-model to build a definition of the meta-modelling sublanguage. (See accompanying paper for a first attempt at this.)
    2. Build a reference implementation for this.
    3. Use the reference implementation to refactor the existing meta-model to fit the new architecture.
    4. Define the meta-modelling profile in terms of the refactoring, adding/updating components only where strictly necessary.
    5. 5 continue with other profiles.

3.1.6 Structure of the UML Specification Document

The proposed architecture will impact the structure of the UML specification document.

Chapter 2 will be expanded (and possibly sub-divided) to describe the overall language definition architecture, to define the kernel library, and to provide guidelines for constructing profiles.

Then will follow a series of chapters defining the various profiles. The OCL chapter will be one of these, as will the XMI and CORBA mappings (these are just extensions of the kernel and/or profiles with additional concrete syntaxes (XMI and CORBA) mapped onto the language concepts.

The notation guide in Chapter 3 would probably be split into two: a precise definition of the concrete syntax, which may be spread across the kernel and various profiles, and a user guide to the language. It could be decided that every profile definition should also be “fronted” by a background section and short user guide. In which case, the second part of the notation guide will be split across the user guides accompanying profiles.

It may be convenient to separate out profile definitions into separate documents, both as a convenience to users (not all users will need all profile definitions), and to avoid a single ‘doorstop’ specification.

The specification document should be accompanied by freely-available reference implementations of the profiles.

3.2 Constraint languages

In summary:

- This area is important and substantial enough to warrant a major revision.
- Subject to acceptance of our approach to semantics, the requirements for the revision can be expressed quite precisely.
- The pUML group have been engaged in work which could be refactored as a response to a RFI in this area.

The semantics issue (2.2.2.4) must be dealt with as the semantics is made more precise, not least because this involves defining the language used to describe the UML meta-model, and, as we already know, OCL is a key part of that language. The integration issue (2.2.2.1) is also dealt with there, in the sense that an abstract syntax for OCL will have to be produced and integrated with the UML meta-model.

The other two weaknesses concern usability of the notation (2.2.2.2) and the ability to support different concrete syntaxes (2.2.2.3). The latter is a property that should improve the former. Both these problems concern the definition of concrete syntaxes for constraints and showing how a common abstract syntax is derived from those different concrete syntaxes.

We believe that it is a lack of engineer-friendly concrete syntaxes, combined with poor tool support, that mitigate against the use of precise constraint languages during modelling. In our experience, where industry has persevered, the quality of modelling, especially at the analysis and business levels, improves.

Therefore, we think this is a sufficiently important and substantial problem to warrant a major revision. On the other hand, if we architect the language definition and semantics appropriately (see section 3.1), the revision can be expressed very precisely in terms of providing a constraint language profile whose purpose is to define a series of concrete syntaxes for constraints, and map these to a common abstract syntax which will be provided as part of the kernel.

We (pUML) have been engaged in work (Kent, 1997; Kent and Howse, 1999) which seeks to address the usability issue by allowing different concrete syntaxes (in this case a visual and a textual one) to be mixed, so that the most appropriate concrete syntax can be used in any given context, where the context can change many times as one writes a single constraint. For example, navigation expressions in a constraint can be shown visually (using a notation called Constraint Diagrams), whereas numerical calculations are shown textually. Kent and Howse (1999) also indicates how certain parts of a constraint can be expressed as prototypical examples, using object diagrams, and suggests that it is a relatively simple matter to provide domain specific concrete syntaxes for these. This addresses the flexibility issue.

In addition, we note that existing UML notations also visualisation (weak) constraints: state diagrams can be viewed, at the analysis level, as visualising pre/post conditions, class diagrams visualise cardinality constraints, and collaborations are, essentially, a less expressive form of constraint diagram.

4 Comparison with Roadmap recommendations

The following sections indicate how what we have proposed addresses the roadmap recommendations. We also propose some additional recommendations, which probably comes under the behavioural modelling section:

Constraint languages

- Provide more user friendly syntaxes for constraint languages.
- Provide a mechanism for producing domain specific syntaxes for constraint languages.
- Formalise the semantics of constraint languages.

The first two of these are addressed by our proposals in Section 3.2. The semantics issue is addressed by the proposed approach to adding semantics into the meta-model, which has already been done in large part in Kent et al. (1999b).

The reasons for these recommendations should already be clear: a constraint language is essential to the definition of the meta-model and to some forms of modelling (analysis, business and specification modelling, in particular); more friendly and visual syntaxes are required for use by modellers.

4.1 Architecture

"Define a physical metamodel that is rigorously aligned with the MOF, etc."

We have proposed the definition of a sub-language of UML for meta-modelling. This is similar to the MOF meta-meta model, but defined in a more declarative, logical style. It is also, itself, more suited to a declarative style of meta-modelling as required for UML. We have given clear guidelines for determining what should go in this language, and have based the language on those guidelines. In particular, the language is intended to be small (but not too small), must at least be good enough to describe itself (completely and with semantics), and is targeted at defining the language in a declarative fashion. It also has constructs to support a fine-grained language architecture and development of profiles.

We have also described a richer, more precise and better delineated architecture for language definition than the 4-level meta-model architecture (at least as it is described in the semantics chapter of UML 1.3). We propose that this is used as the reference architecture for defining UML.

"Provide guidelines to determine what constructs should be defined in the kernel language, and what constructs should be defined in UML profiles and standard model libraries"

We have discarded the idea of a single kernel language, proposing instead a kernel library of sub-languages (represented as a network of models in the meta-model) which can be used to define profiles. We have proposed an architecture which separates concrete syntax, abstract syntax, semantics domain and the relationships between each pair.

As with every modelling exercise, what goes in the library depends on what profiles you need to support and how you decide to organise the various pieces. We have proposed one possible organisation, which we expect to need refactoring as the details are filled in. This follows experience in industry practice which is moving towards short, incremental development steps, and being prepared to refactor designs as new requirements dictate.

"Apply the aforementioned guidelines to improve the integrity and quality of the kernel language. The process should remove constructs from the kernel that are vaguely defined or rarely used, and should provide a basis for defining new profiles."

A key to our approach is the use of a Catalysis-like package imports mechanism. This is what allows new profiles to be built atop the kernel library and other profiles, and allows for fine-grained structuring of the meta-model. It allows rarely used constructs to be pushed into the profiles where they are needed and kept out of the profiles where they are not needed. The proposed concrete syntax/abstract syntax/semantics domain split considerably improves the integrity and quality of the language definition, as it makes precise and explicit the concrete syntax and semantics (which were previously implicit), and relationships between the three parts.

4.2 Extensibility

"Provide a first-class extensibility mechanism consistent with a 4-layer metamodel architecture."

Whenever a new piece of language is required, the meta-model library is extended with a new package defining that part of the language. This will include definition of concrete syntax, abstract syntax and semantics. Stereotypes can be seen as a simple way of extending concrete syntax, and support for them could be provided in the kernel library, e.g. by allowing every concrete syntactical element to have a stereotype associated with them. It would then be the responsibility of the language designer to say how any particular pattern of element(s) with its (their) stereotype(s) mapped to the abstract syntax and then, subsequently, to the semantics. That may require either or both of the abstract syntax and semantics to be extended and/or specialised.

“Improve the rigor of profile specifications so that they can support increased user demands for language customization”

The split between concrete syntax, abstract syntax and semantics, means that profiles may be customised in degrees. One may just extend or specialise the concrete syntax, without changing abstract syntax or semantics. Or it might be necessary to introduce new abstract syntax (with or without additional concrete syntax), which can then be mapped to existing abstract syntax and get its semantics via that route. Or it may be necessary to extend or change the semantics. In general, the first requires less expertise than the second, which requires less expertise than the third, suggesting that users may be given limited tool support to do the first, but perhaps not the other two.

The package extension mechanism allows pieces of profiles to be abstracted out (without affecting the behaviour of the profile) into a parent package that can then be reused to build a new package. This technique would be required, for example, if you wished to replace the semantics of a construct with something different, but keep the same syntax.

4.3 Relationships

“Provide a more complete semantics for refinement and trace dependencies ...”

We have sketched an approach to refinement (realisation) of models which we have tried out for real and is based on Catalysis ideas, which can be expressed itself as a particular pattern of package imports (an abstract model and a concrete model imported into a realisation model, which adds relationships between elements of each and constraints). We have argued that this is an important pattern for modelling languages themselves. We are aware that this pattern needs to be further explored, for example how to represent refinements of dynamic models.

4.4 Behavioural Modelling

Statecharts and Activity Graphs

All the recommendations here are to do with semantics. Our approach to semantics should be able to support what is required here, building on the current work of Rumbaugh, Selic et al. on defining an action language for UML. If it turns out that some difficult aspects of the semantics can not be encoded in the meta-model itself, the external semantics (e.g. in set theory) of the meta-modelling sub-language, should allow external semantics to be built using mathematics appropriate to the task. Alternatively, we just accept that some aspects of the semantics are not given a precise definition, or are left open to tool builders to fill in (with some carefully worded guidance in the semantics document). Whatever, we will be in a much better position than the current status quo.

Collaborations

“Define a more complete semantics for defining patterns.”

The proposed more sophisticated version of package imports supports model templates, which can be used to model patterns to a limited extent. Key is the ability to merge and rename model elements imported from parents. To completely represent patterns, one needs to be able to extend the meta-model itself - a pattern, at its most general, is a collection of constraints on what is and is not allowed in a particular kind of model. These can only be expressed in their entirety within the meta-model (e.g. define a subclass of model which represents abstract factory models; that subclass identifies a set of its classes as being factories, and a set as being products, and these must be related in a particular way, etc.).

4.5 Model Management

“Refine notation and semantics for models and subsystems to improve support for enterprise architecture views”

In our scheme, this could be represented using a new profile that extended the standard set. This would essentially identify a pattern of for building models suitable for modelling enterprise architectures. It would probably be one of the vertical profiles.

4.6 General mechanisms

“Define a metamodel and mechanism for model versioning.”

This can probably be handled in a similar way to refinement, i.e. as a model pattern. Version n will be one model, version n+1 will be another, and there will then be a model which records the relationship between these two versions. This would allow elements in a new version to be traced back to their counterparts in an older verison.

“Specify a metamodel and XMI mechanism for diagram interchange.”

Both diagrams and the XMI are different concrete syntaxes on the same abstract syntax. Thus models can be viewed as diagrams or XMI. However if, as we suspect, this is asking for diagram layout information to also be transferred, then an XMI syntax will need to be defined that maps to the diagram concrete syntax (i.e. treats the diagram concrete syntax as more abstract than itself). All these syntaxes and their mappings can be accommodated within the proposed meta-modelling approach.

5 Recommendations for revisions

We have indicated how what we propose addresses many of the roadmap recommendations. However, much remains to be done.

1. The UML specification needs to be restructured to reflect the proposed architecture of the language definition.
2. The meta-modelling sub-language needs to be fully defined.
3. The ideas about architecture need to be made concrete, by providing a kernel library, by providing some standard profiles and by building reference implementations of those.
4. Profiles dealing with particular aspects of the language, particular domains, etc. need to be built.

There is a natural ordering here, the ordering used above. Except that we won’t know for sure what needs to go in the kernel library until some standard profiles are built, and similarly we won’t know whether the standard profiles are the right ones and are appropriately structured until the profiles dependent on them are built.

In addition, the same group will not be building all the profiles, and those building profiles may need advice and support from those establishing the architecture and the kernel library.

So, this is what we propose:

- As part of the RFP writing process, as much work as possible is done by the semantics working group on 1, 2 and 3, using the existing meta-model as a basis. The attachment to this document has started this process. We soon hope to start construction of a reference implementation. A goal will be to ensure that the meta-modelling sublanguage is fully defined and implemented, and that the new structure of the UML specification document is defined by the time RFP’s are issued.
- This work continues during the 2.0 revision process.
- RFP’s for specific profiles are issued in March 2000 (or thereabouts). The RFP’s will be for some standard, horizontal and vertical profiles. The particular choice of profiles will depend on a combination of what is most important to industry, what is deemed feasible in the time, and what is in place by the time the RFP’s are issued. Submitters are required to adopt a meta-modelling approach, and are encouraged to work with the semantics working group to improve their submissions. They can use the reference implementation of the meta-modelling sublanguage to develop the meta-model, thereby ensuring that they do not stray from that language when building their definitions. Note that this will require the tool to be useable. They can also produce a reference implementation for their profile, if desired (suggests coordinated open source of reference implementations).

This model is similar to the Linux model, where the kernel is closely controlled by a small group (it has to be), but library components and flexible mechanisms are provided to support the development of specialised extensions.

The main risk we perceive with this approach is that it relies on the goodwill of the semantics working group: the amount of time required to do the work justice may mean that it may not be possible to rely on a purely voluntary contribution. Resources will also need to be found to build the reference implementation(s).

References

Berner S., Glinz M. and Joos S. (1999) *A Classification of Stereotypes for Object-Oriented Modeling Languages*, in France & Rumpe (1999).

Clark A. (1999) *Typechecking UML Static Models*, in France & Rumpe (1999).

D’Souza D., Sane A. and Birchenough A. (1999) *First-Class Extensibility for UML - Packaging of Profiles, Stereotypes and Patterns*, in France & Rumpe (1999).

D'Souza D. and Wills A. (1998) *Objects, components and frameworks with UML*, Object Technology Series, Addison-Wesley.

Evans A. (1998) *Reasoning with UML class diagrams*, in WIFT'98. IEEE Press.

Evans A. and Kent S. (1999) *Core Meta-Modelling Semantics of UML: The pUML Approach*, in France & Rumpe (1999).

France R. (1999) *A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts*, in Proceedings of OOPSLA’99, ACM Press.

France R. and Rumpe B. (eds.) (1999) *Proceedings of UML’99 - The Unified Modeling Language, Beyond the Standard: Second International Conference*, Fort Collins, CO, USA. LNCS 1723, Springer Verlag.

France R., Evans A., Lano K., and Rumpe B. (1998) *Developing the UML as a formal modeling notation*, Computer Standards and Interfaces: Special Issues on Formal Development Techniques, 1998.

Jackson D. (1999) *Alloy: A Lightweight Object Modelling Notation*, available from http://sdg.lcs.mit.edu/~dnj/abstracts.html#alloy

Kent S. (1997) *Constraint Diagrams: Visualising Invariants in Object Oriented Models*, in Proceedings of OOPSLA’97, ACM Press.

Kent S. and Howse J. (1999) *Mixing Visual and Textual Constraint Languages*, in France & Rumpe (1999).

Kent S., Evans A. and Rumpe B. (eds.) (1999a) *UML Semantics FAQ*, in ECOOP’99 Workshop Reader, Springer Verlag.

Kent S., Gaito G. and Ross R. (1999b) *A Meta-Model Semantics for Structural Constraints in UML*, in Kilov et al. (1999).

Kilov H., Rumpe B. and Simmonds S. (eds.) (1999) *Behavioural Specifications of Businesses and Systems*, Kluwer Academic Publishers.

Montague R. (1974) *Universal Grammar*, in Thomason R.H. (ed) “Formal Philosophy: The Selected Papers of Richard Montague”, Yale University Press.

Richters R. and Gogolla M. (1999) *A Metamodel for OCL*, in France & Rumpe (1999).

Schürr A. and Winter A. (1999) *UML, the Future Standard Software Architecture Description Language?*, in Kilov et al. (1999).