UML Semantics FAQ

Stuart Kent, Andy Evans and Bernhard Rumpe:
editors on behalf of the pUML group

pUML@york.cs.ac.uk,
WWW home page: http://www.cs.york.ac.uk/puml

Abstract. This paper reports the results of a workshop held at ECOOP'99. The workshop was set up to find answers to questions fundamental to the definition of a semantics for the Unified Modelling Language. Questions examined the meaning of the term *semantics* in the context of UML; approaches to defining the semantics, including the feasibility of the meta-modelling approach; whether a single semantics is desirable and, if not, how to set up a framework for defining multiple, interlinked semantics; and some of the outstanding problems for defining a semantics for all of UML.

Introduction

This paper describes the results of a workshop held at ECOOP 1999, in Lisbon. The aim of the workshop was to identify and answer key questions concerning the semantics of the Unified Modelling Language (UML [8]). A list of the questions discussed here is given below:

1. What does the term *semantics* mean in the context of UML?
2. Why is it desirable to make semantics for UML explicit? What are the different ways in which a UML semantics might be made explicit?
3. Is a precise semantics desirable? For what purpose?
4. What is the current status of UML semantics? What are the reference documents?
5. Should UML have a single semantics? Should UML have a single *core* semantics?
6. Is it possible to express a semantics of UML in UML (the meta-modelling approach)?
7. Is it feasible to construct a semantics for all of UML? What are the main outstanding issues?

Specific aspects of UML were explored in attempting to answer these questions. There was broad agreement on questions 1-4; it was generally felt that individual contributions submitted before the workshop could be polished in answer to these questions. Participants broke out into groups to discuss the remaining three questions. Two groups considered the last question by exploring two specific areas of UML, respectively: concurrency, events, and dynamic behaviour in general; and aggregation.

This report is a snapshot of the state of affairs at the time. The UML Semantics FAQ will continue to be maintained by the pUML group, who encourage contributions from practitioners and academics interested in the UML as a standard modelling language. To this aim we are continuing to organise workshops at major conferences such as ECOOP and OOPSLA. However, this does not preclude other forms of contribution: it is always possible to improve current statements or add new topics. If you have any comments, or would like to offer alternative answers or suggest clarifications to existing ones, then please visit the website of the precise UML group (pUML), located at

`http://www.cs.york.ac.uk/puml`
where an updated FAQ will be maintained.

Q1. What does the term *semantics* mean in the context of UML?

Bernhard Rumpe, Technische Universität München, Germany
Stuart Kent, University of Kent at Canterbury, UK
Andy Evans, University of York, UK
Robert France, Colarado State University, USA

Q1.1. What does the term *semantics* mean at all?

Today, a lot of confusion arises from the fact that the word "*semantics*" itself has many different semantics! Developers tend to use the word "*semantics*" when they talk about the behavior of a system they develop. This kind of usage is almost contradictory to the *semantics* in scientific areas like Mathematics or Logic. There, "semantics" is a synonym for "meaning" of a notation – this is regardless of whether this notation deals with structure or behavior.

Basically, a semantics is needed if a notation (syntax) is given or newly developed, and its meaning needs to be defined. Almost all approaches define the semantics of its elements by relating it to another already well understood language.

This is comparable to natural languages. For example Chinese can be (roughly) understood if a Chinese-English dictionary is available. Of course grammar, or the definition of how elements of a language are modified and grouped together, also need to be mapped.

In computer science, the pattern is similar. A new language is given a meaning in three steps:

1. define precisely the *syntax* of the new language, which characterises all the possible expressions of that language
2. identify a well understood language, herein called the *semantics language*, and
3. define a mapping from expressions in the syntax of the new language to the semantics language.

The semantics language is often called the *semantics domain*. The mapping from syntax to semantics is usually intensional rather than extensional, which means that the mapping is not explicit, but by example. If a language is to be automatically processed and/or be unambiguous then the syntax, the semantics language, and mapping from one to the other must be completely, precisely and unambiguously defined.

Q1.2. What is special about UML *semantics*?

UML does have some specific characteristics, which makes the task of semantics definition interesting:

1. a substantial part of UML is visual/diagrammatic.
2. UML is not for execution, but for modeling, thus incorporating abstraction and underspecification techniques.
3. UML is combined of a set of partially overlapping subnotations.
4. UML is of widespread interest.

Whereas the last issue leads to the sociologically interesting question, how to reach agreement for a semantics definition, the other three topics lead to problems of a technical nature.

The fact that a large part of UML is diagrammatic makes it somewhat more difficult to deal with its semantics, but it is no problem in principle. Currently, its semantics is explained in English: UML semantics is ambiguous and imprecise. We speak of a *formal* or *precise* semantics for UML if the semantics domain of this translation is a formal language and – very important – the translation itself is precisely defined. This goal can be achieved, as several graphic formalisms, like Statecharts, Petri-Nets, or dataflow-diagrams have shown. The first step for UML is to precisely define its syntax. In the standard, this has been done by using the meta-model approach, which in the UML documents is mainly used to describe the abstract syntax of the UML [8] itself. Thus a meta-model for diagrams replaces the abstract syntax tree of textual notations.

The usage of UML as a modeling language and not as a programming language has an important impact that is often poorly recognized. A UML model is an *abstraction* of the real system to be developed. The model is used to capture important properties, but to disregard unimportant ones. As an effect, a UML model typically has a set of more than one possible implementation. A semantics definition must reflect this by making the *underspecification* of the model explicit.

Third, the UML is composed of a set of notations that partially overlap. For example [4] shows how (a subset of) the state diagram notation can be used to express the same information that could be expressed in terms of pre/post conditions on operations in a class diagram; but there are other aspects of state diagrams which can not. This adds an extra problem, as semantics definitions for each of the UML notations need to be consistent with each other. Only then will an integrated use of these notations be feasible. To check the consistency

of semantics definitions, it is necessary either to have a common semantic domain for all of them, or to establish precise mappings between different semantic domains.

A more detailed discussion of these topics can be found in [13].

Q1.3. What is a UML semantics good for?

Semantics of UML is a means to understand how UML should be used, and to ensure that when UML models are communicated there is a common shared understanding of what they mean. On the other hand, the actual practice of applying UML is necessary to get a *feeling* for it. A semantics definition is a necessary prerequisite, but certainly not sufficient. Furthermore, it is not necessary to understand the complete language to start using it.

Semantics is a vehicle for people who speak the same semantic language $\mathbb{D}$ (formal or informal) to discuss certain UML subtleties and improve the notation and use of UML in terms of $\mathbb{D}$.

Semantics can be for automating certain tasks by machine: for example, tools which can do more than simply process syntax, such as simulating or (partially) executing models, checking models are consistent, etc.

It is important to clarify the purpose of a semantics definition. There may be different semantics definitions to suit different purposes: the definition for explaining semantics to users of the notation may be different to that required to perform sophisticated automatic processing tasks, and both may be different to a semantics definition whose purpose is to demonstrate properties about the language, such as a measure of how expressive it is compared to other languages.

Q1.4. Common misunderstandings about semantics

The UML documents contain a paper called the "*Semantics of UML*". However, this paper does not focus much on semantics, but mainly on syntactic issues. The meta-model of UML gives a precise notion of what the abstract syntax is. However, it currently does not cope with semantics. Analogously, the semantics of C++ can not be understood from the context free grammar (without knowledge of similarly structured languages).

Furthermore, context conditions are by no means *semantic conditions*, but purely constrain the syntax. They give well-formedness rules, e.g. each variable must be defined before use, without telling you what a variable is. In the UML case, context conditions are usually explained using OCL. A context condition tells us what is constrained, not why it is constrained. The latter is a task of the semantics definition.

As explained earlier: semantics is not behavior. A structural description technique, like class diagrams, need an adequate semantics in the same way as do behavior description techniques.

Q2. Is a precise semantics desirable? For what purpose?

Stuart Kent, University of Kent at Canterbury, UK
Bernhard Rumpe, Technische Universität München, Germany
Andy Evans, University of York, UK
Robert France, Colarado State University, USA

Q2.1. Degrees of precision

A semantics definition consists of three parts:

1. define the syntax,
2. define the semantics domain, and
3. define the semantics mapping, as a relationship between syntax and semantics domain.

The degree to which each of these three parts is made explicit and/or precise may vary. Whenever a natural language, like English, is involved, we speak of an informal semantics definition.

The semantics definition gains much on precision, if at least its syntax is precisely defined. One can distinguish between a *concrete* and an *abstract* syntax. A concrete syntax provides the rules for stating exactly how the language will appear when written; the abstract syntax identifies the main concepts onto which the concrete syntax maps. These concepts are then given a semantics. For example, the following equations define the *concrete* syntax for numerals:

$$Character \supseteq Digit = \{'0','1','2','3','4','5','6','7','8','9'\}$$
$$String \supseteq Numeral = Digit \cup \{d \frown n \mid d \in Digit \wedge n \in Numeral\}$$

The concrete syntax talks about concrete things: digits are characters, numerals are strings. In an abstract syntax we would just assume that we have a set $Numeral$ of arbitrary tokens. The semantics would map members of this set uniquely to the natural numbers in the (well understood) mathematical language of arithmetic.

For UML, and other diagrammatic languages, the trend has been to define an abstract syntax. So, rather than, say, talk about boxes and lines between them (the equivalent of digits and strings) the syntax talks about class and association.

For textual languages, a context free grammar (BNF) is used, though this can be viewed as just a short hand for set theory. There is no reason why similar techniques could not be used to describe both the concrete and abstract syntax for diagrammatic languages such as UML. The UML standard has chosen to use a meta-modelling approach based on class diagrams to characterise the abstract syntax of the language. The concrete syntax seems only to be defined by example.

A precise definition of the semantics domain is usually given either by explicitly defining the notion of "system" using mathematical terms, or by using a formal language, like Z or Object Z, as the semantics language. However, precision does not require the language to be mathematical in the traditional sense.

Finally, to get a fully precise semantics, the semantics definition must also be represented precisely. This is feasible using mathematics, as done many times for other notations. The mappings can also be encoded in a meta-model — see FAQ 5 for details.

An alternative way to write down the mapping is algorithmically — a recipe for converting expressions in the (abstract) syntax to expressions in the semantics language. This would be useful where it is intended that the mapping is to be automated, for example where the semantics domain is an OOPL such as Java, and the mapping corresponds to code generation. Unfortunately, using a programming language as a semantics domain leads to a severe problem which needs to be considered: any model defined in an executable langauge can only describe one implementation and therefore cannot exhibit any form of underspecification. As discussed earlier, modeling languages like UML need to allow underspecification. Thus code generation necessarily involves a selection of one of the possible implementations – possibly a wrong one.

Q2.2. Abstraction versus precision versus detailedness

In the UML reference book [12] there is a detailed definition of the nature and purpose of models given. Abstraction is mentioned there as a key concept to yield understandable models, conveying the essentials of a view.

Different levels of abstraction allow information to be revealed about the systems on different levels of detailedness. So abstraction and detailedness are complementary – adding details makes a model less abstract. However, the word "precision" is ambiguous in that context. Sometimes it refers to the amount of details a model has and sometimes it is used as the degree of formality the modeling notation has. These two kinds of "precision" need to be distinguished. "Precision of a notation" refers to the precision of the definition of its syntax and semantics and is the same for all models, not to the amount of detail included in the model.

Physics gives us a good example. "About 42" is a vague definition for a number, it is neither very detailed nor precise. One cannot exactly determine whether 41.7 is included or not. "About 41.34" is more detailed, but still not precise. It seems likely that 41.7 is excluded, but we cannot be sure, if we don't precisely know what "about" means. Physics gives us a precise technique: "42.0" determines the exact interval $[41.95, 42.05]$. The notation is fully precise, but we can make it more detailed: "42.010" is a specialization conveying $[42.0095, 42.0105]$.

Of course this example is simple compared to the situation with UML. However, it is important to recognize that with the UML we can specify precisely, but still in an abstract manner, using underspecification wherever appropriate.

Q2.3. Why is precision important? What do you lose?

A Benchmark 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.

Establishing properties of the syntactic language Some important, but nevertheless largely neglected, issues in defining semantics are wrapped up in the question: Is the semantics mapping appropriate? Let us assume we denote the syntactic notation by $\mathbb{N}$, the semantics domain $\mathbb{D}$ and the semantics mapping as function

$$\mathbb{S} : \mathbb{N} \to \mathbb{D}.$$

By defining $\mathbb{S}$ precisely, we can use mathematics to prove properties about the syntactic language. For example, let us assume $\emptyset \in \mathbb{D}$ describes the invalid (or non-implementable) system, then it is an important result if we can show the following property:

$$\text{for all models } m \in \mathbb{N} \text{ it holds that } \mathbb{S}(m) \neq \emptyset$$

Another test of the appropriateness of a semantics definition is the question regarding what it distinguishes and what it identifies. Is the following possible:

$$\text{there are models } m_1, m_2 \in \mathbb{N} \text{ with } m_1 \neq m_2, \text{ but } \mathbb{S}(m_1) = \mathbb{S}(m_2)$$

Models that only differ in local variables should indeed have identical semantics, as the semantics should not rely on hidden or local properties. On the other hand, does the semantics distinguish models that are inherently different in nature?

Of greatest interest is the transfer of refinement, abstraction, composition and similar techniques from the semantic domain to the syntactic domain. Assume $\rightsquigarrow$ denotes a refinement relation on the semantics domain. Then it would be interesting to define transformation techniques $T : \mathbb{N} \to \mathbb{N}$ on the syntax with the following property:

$$\text{for all models } m \in \mathbb{N} \text{ it holds that } \mathbb{S}(m) \rightsquigarrow \mathbb{S}(T(m))$$

If this property is ensured once and for all models, then the transformation T can be applied, and we can be sure to get a refinement as a result, without dealing explicitly with the semantics anymore.

Thus representing the semantics mapping $\mathbb{S}$ precisely allows the notation developer to prove properties on his/her notation and associated development

techniques, such that the user of the notation later need not explicitly deal with the semantics anymore. The syntactic transformations may also be built into automated tools to support the developer.

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. This is the ultimate goal of a semantics definition.

Losses 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) develop a precise, yet rather widely agreed, semantics; (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.

Q3. What is the current status of UML semantics? What are the reference documents?

Martin Gogolla, Mark Richters and Oliver Radfelder,
University of Bremen, Germany

The current status of the UML semantics is that it is decribed in an informal manner. The 'UML Notation Guide' document gives an overview on the concepts, and the 'UML Semantics' document presents the abstract syntax together with context-sensitive conditions in form of class diagrams and OCL expressions. Both documents as well as the semi-official books by Booch, Rumbaugh, and Jacobson do not use formal techniques for explaining the semantics.

Concerning the reference documents, when studying the UML and especially the UML semantics, one has to take into account the official OMG UML definition, especially the 'UML Notation Guide' and 'UML Semantics' documents. But the problem is that the UML is an evolving language. Therefore many versions of these documents exist. The current version is version 1.3 [8] but there is already a call for contributions for version 2.0. In addition, there are also many books and papers on the subject, including the semi-official ones by Booch, Rumbaugh,

and Jacobson, especially the 'UML Reference Manual'. Because of publication lead times, laziness of researchers and so on, one has to be very careful when reading a paper or book to identify on exactly which version of the UML it is based.

For example, one is likely to come up with a very different semantics for signals, depending on whether you read the UML standard or the three amigos reference guide:

- *Signals ... have no operations.*
  UML Notation Guide, Version 1.3, page 3-138, line -4..-2.
- *A signal ... may have operations.*
  UML Reference Manual, page 428, line 3.

In our view, semantics work must only use the official UML definition, which can be obtained from the OMG pages at `http://www.omg.org`, must try to use the latest version possible, and must make it clear which version has been used. And, as an aside, the author for this document is the OMG, not Booch, Jacobson and Rumbaugh.

Finally, anyone working on UML semantics should look up existing work in this area. An up-to-date bibliography is maintained at

`http://www.db.informatik.uni-bremen.de/umlbib/`.

Also see the pUML web pages.

Q4. Should UML have a single semantics? Should UML have a single *core* semantics?

John Howse, University of Brighton, UK
Shusaku Iida, Japan Advanced Institute of Science and Technology (JAIST)
Richard Mitchell, InferData Corp, USA and University of Brighton, UK
Bernhard Rumpe, Technische Universität München, Germany

The advantage of having a single, standard semantics for UML is that it is easier for one person to understand another person's UML models. The advantage of having a variety of semantics is that you can choose what works best in your current project. We believe it is possible to support both standardisation and variation. Therefore, we answer 'no' to the question 'should UML have a single semantics?' The following sections elaborate on this answer. First, we explore a world in which variety is allowed. Then we explore how it could be actively supported.

Q4.1. Who can give UML a semantics?

In practice, there are many individuals and groups who can contribute to discussions on the semantics of UML, and many ways for them to disseminate their proposals. Here are some:

Who?	**Disseminate how?**
OMG task groups	UML documents
The three amigos	Books
Tool vendors	Tools
Methodologists	Books, Research papers
Development teams	Shared experience

As long as no single group or individual has control of the semantics of UML, there will be a variety of semantics for UML. Some will be more popular than others, meaning that more people understand them and use them. Popularity would be influenced by a number of factors, including:

- the fame and power of those proposing a particular semantics (for example, you might pay particular attention to proposals from an OMG group)
- strengths and weaknesses of a particular semantics as highlighted by scientific research (for example, a semantics with proven flaws might become less popular)
- strengths and weaknesses of a particular semantics as highlighted by practical experience (for example, you might be influenced by favourable reports from several large projects)
- the effectiveness of tool support (the availability and quality of tools might influence your choice).

In practice, if variation was supported, we would expect there to be parts of UML for which there is widespread (but not universal) agreement on the semantics, and other parts for which there is less agreement.

And finally in this section, it is appropriate to note that UML is extensible, through such mechanisms as stereotypes. Modelers who introduce their own stereotypes must define the semantics of their models. Therefore, the problem we address in the next section must be addressed even if the UML community opts for central control of semantics.

Q 4.2. How can we support variation?

There are two parts to our answer to the question of how can we provide support for a variety of semantics for UML. The first concerns a style of writing semantics. The second concerns a way of organizing shared semantic models. For us, supporting variation includes both supporting the sharing of parts of UML, when that is appropriate, and supporting different variants of UML, when that is helpful.

A semantics for a language, L, maps elements of L to some other language, S, using a semantic function. The language S describes a semantic domain. A semantic function could map elements of L to elements of S. Alternatively, a semantic function could map each element of L to a set of elements of S, yielding a set-based semantics, illustrated in Fig. 1.

A set based approach to defining semantics has a number of advantages. First, it helps to make any underspecification in a UML model explicit. The

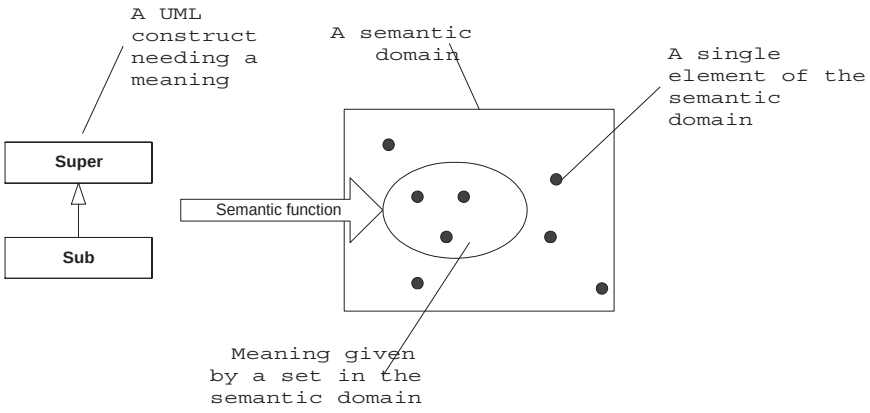

Fig. 1. Set-based Semantics for UML

more abstract (i.e., underspecified) a UML model is, the more elements (implementable systems) can be found in the set. The more detailed a model is, the fewer elements can be found.

Secondly, a set based approach allows us to have variants of semantic definitions. For example, we could build two different semantic mappings such that the second is a specialization of the first: for each model, the semantics given by the second mapping is a subset of the semantics given by the first mapping.

Thus, a hierarchy of semantics definitions can be built, having a very general standard semantics (which has a lot of possible implementations), and very specialized semantics for particular projects.

We do expect that it will be possible to get large groups of people to agree on a semantics for many parts of UML. How might such agreement be conveyed, whilst still allowing variation? We suggest that every UML model has no meaning until someone prefaces it with a definition of the semantics of the modeling language (see Fig. 2). Think of the preface as being a semantic definition of the UML (chosen to be appropriate for your project) that conceptually comes before the model your project is building.

Prefaces will be large documents. They could be organized into a hierarchy of packages. Then a project team could begin with a popular package agreed by the OMG that defines a core of UML, and tailor it with a package from the vendor of the tool they plan to use, then with a package that defines company-specific meanings to certain parts of UML, and finally with a package that defines the project's own variations.

Further details of how packages might be used in this way can be found in [2] from which the illustrative diagram in Fig. 3 is taken.

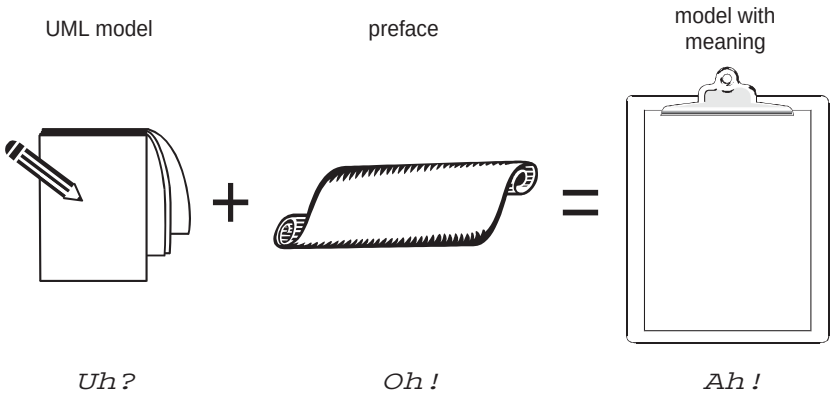

Fig. 2. Prefaces

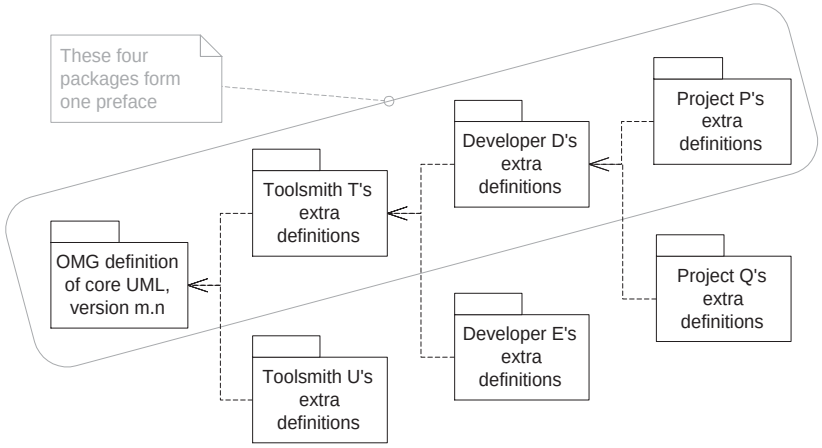

Fig. 3. A Hierarchy of Semantics Packages

Q5. Is It Possible to Express a Semantics of UML in UML (the meta-modelling approach)?

Martin Gogolla, University of Bremen, Germany
Stuart Kent, University of Kent at Canterbury, UK
Tom Mens, Vrije Universiteit Brussel, Belgium
Mark Richters and Oliver Radfelder, University of Bremen, Germany

Our answer to this question is: Yes, this is possible for a large part of UML. We call this approach to define the UML semantics the meta-modeling approach.

Q5.1. Why Use UML to Explain UML?

Before we show the details of this approach, we explain why we think it it useful to express UML in terms of UML itself. If one wants to describe the semantics of a language L_{start}, then one has to deal with at least one other language L_{target}, which is the language in which the semantics is to be given. Thus in order to understand the semantics one has to know both languages L_{start} and L_{target}.

The main advantage we see in the meta-modeling approach is that people wanting to understand the semantics of UML do not have to learn another language L_{target}. They just see the language whose features they want to see explained and use it for the translation. Proceeding this way, people who are not experts in formal semantics or formal specification languages will be given the chance to reason about UML without the burden of learning another language L_{target}.

The main danger of this approach, however, lies in the question of whether the expressive power of UML is suitable to express its semantics. We argue that a large part of UML can be expressed in this way but we currently do not know whether all UML features can be treated. On the other hand, it is also a demanding and interesting question for us to see which parts of UML can be transformed into UML and which parts not. Another danger of the approach is the possibility that confusion may arise because there is only one single language: one has to point out very clearly whether one is speaking of (1) a UML element which is currently being translated or (2) a UML element which occurs as the result of the translation.

Q5.2. Central Steps of the Approach

We now show the main steps to be taken for our meta-modeling approach. The general idea is presented in Fig. 4.

1. The first step to be taken is to develop the syntax of the core meta-modelling langauge, which is akin to what has been dubbed the *MOF*, essentially a diagrammatic modelling language for defining other languages (though we note that the current MOF is not as precise as it should be, and much larger than it needs to be). It is described in some detail below. This language will

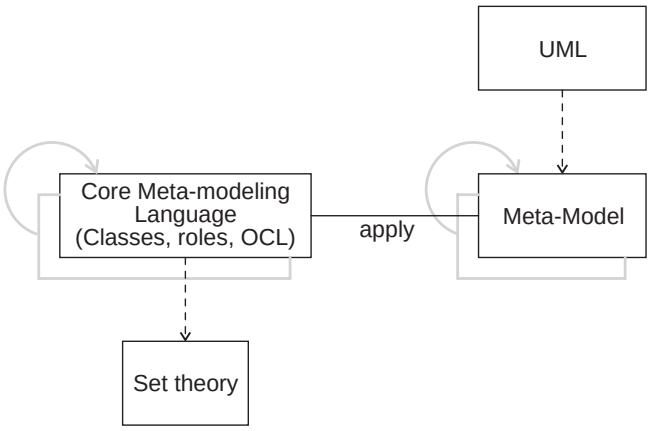

Fig. 4. General Approach to Answer Q5

for sure include class diagrams in which classes and binary associations are allowed. As a central description vehicle we will also use the Object Constraint Language OCL. Important restrictions on models must be expressed by using OCL for class invariants.

2. The second step is to develop the semantics of this core. As Fig. 4 shows, we propose to use set theory as the basis for the semantics. For us, set theory is the most natural choice here, because it is a widely accepted and the most popular formal foundation. Of course there are other approaches around (like temporal and dynamic logic, algebraic specification, streams, process theory, etc.) with advanced properties, but the drawback of these approaches are that they are hard to learn or that there is at least an additional learning effort for the average UML modeler. It is also important to note that the core should be kept as small as possible. It should be possible to gain an intuitive, yet unambiguous, understanding of the language without having to read its precise, set theoretic definition.

   With reference to FAQ 4, the core meta-modeling language and its semantics constitute a preface.

3. The third step is to produce a meta-model description of the set theoretic semantics of the UML core, using the UML core itself. This will include concepts such as *object configuration* or *snapshot*, which will contain objects and links, etc.

4. The fourth step of the meta-modeling approach is to transform the syntax of the complete UML language along the lines presented in the UML semantics document into the UML meta-model, i.e. into an abstract syntax. This step converts each concrete syntactic element into an element of the UML meta-model.

5. The last step is to relate the meta-model description of the syntax to the meta-model description of the semantics. This can be done directly (e.g. as is done for the UML core) or indirectly.

   Directly Syntactic concepts are associated with semantic concepts. Thus models are associated with snapshots and traces (sequences of snapshots, interspersed with action invocations), and OCL constraints are written to ensure that all the semantic mapping rules are in place, e.g. that, in a snapshot, the number of links from an object of class A to objects of class B via association ab, must lie between the cardinality bounds imposed on ab. Or that the ordering of action invocations in a trace match the ordering of actions in sequence diagrams, that the transformation in snapshots through a trace satisfy the pre and post conditions of the actions involved, and so on. More detailed descriptions of this approach can be found in [6, 5].

   Indirectly Features of the meta-model characterization of the UML abstract syntax are mapped into the meta-modeling core developed before. A very simple example for a transformation of UML language features into the meta-modeling core is given by Fig. 5. This shows how explicitly notated multiplicities in class diagrams can be transformed equivalently into a more basic class diagram without the muliplicities but with an additional restricting OCL invariant. The OCL formula (using a slight generalization of <=) requires that all instances of class A are connected to at least low and at most high objects of class B. An example of defining aspects of UML within UML can be found in [7]

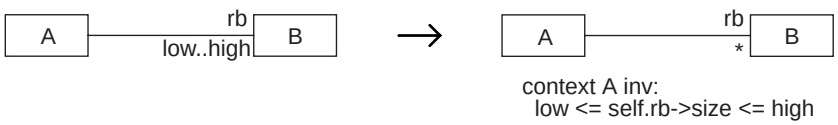

Fig. 5. Transformation of Multiplicities into OCL Invariant

The grey rectangles and the grey arrows in Fig. 4 indicate that both the core meta-model and the UML meta-model could be extended in a bottom-up manner so that more advanced features become available for meta-modeling. Such extensions could even be iterated, i.e. there could be extensions relying on already defined extensions.

Q5.3. Features of the meta-modeling Core

In Fig. 6 we have displayed the main features of the meta-modeling core in some detail as a class diagram. Roughly speaking, the upper part shows classes responsible for a generic description of UML language elements, also called descriptor

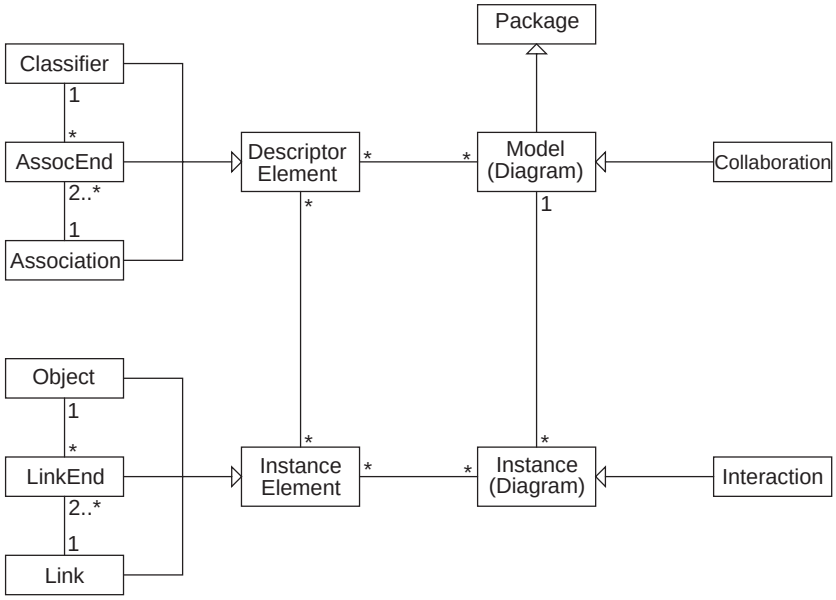

Fig. 6. The basis of the meta-modeling Core

elements in the UML. The lower part describes classes responsible for instantiations of the generic elements, also called instance elements in the UML. The upper and lower part are linked by (binary) associations establishing the connection between generic descriptor elements and instance elements by describing which elements of the instance level belong to which element of the generic level. (Note that concepts of descriptor and instance element do not currently appear in the standard UML meta-model – we believe they must.)

For example, the upper part mentions classifiers (i.e. in particular classes) and the lower part mentions objects. One class in an application domain (e.g. the class Person) will be described by one object of class Classifier. For a concrete system state, each Person instance, i.e. each Person object, will be represented by an object of class Object. The information that an instance in a concrete system state belongs to class Person will be provided by a link belonging to the association between InstanceElement and DescriptorElement.

Note that the meta-modeling core only includes very basic modeling features (i.e. classes, binary associations and the Object Constraint Languge). The core does not include UML class diagram language features currently in controversial discussion (controversial at least from the formal semantics point of view – see FAQ 6) such as aggregation and composition.

Q5.4. What can be Treated Nicely? Where are the Problems?

The ideas presented above can be applied to a variety of UML diagram forms. In general, we have the feeling that the approach works well for static and basic dynamic aspects in UML, but we are unsure about the advanced dynamic aspects. The approach works for:

- Language features of the UML core language,
- Pre- and postconditions formulated in OCL as part of class diagrams,
- Advanced class diagram features like aggregation and composition,
- A subset of statechart diagrams,
- Combinations of certain actions in statechart diagrams like parallel composition and concatenation, and
- A subset of sequence and collaboration diagrams.
- Model (package) extension and refinement (at least partially).

Areas with UML diagram features where problems are expected and further intensive research must take place are:

- Concurrency,
- Events and triggering,
- Active objects,
- Full treatment of refinement, and
- Deployment.

Q6. Is it feasible to construct a semantics for all of UML? What are the outstanding issues?

The first of these questions can only be answered constructively: the answer will be unknown until a semantics for all of UML is constructed. However, the more that is constructed the more confident we can be of a positive answer. The best that can be done, then, at this stage, is to examine the current state of affairs [8], and identify the places where it is incomplete, ambiguous or inconsistent. Improvements have been suggested where possible.

Q6.1. Dynamic behaviour and concurrency

Roel Wieringa, University of Twente, the Netherlands
Egidio Astesiano and Gianna Reggio, Universita' de Genova, Italy
Alain Le Guennec, IRISA, France
Heinrich Hussman, TU Dresden, Germany
Klaas van den Berg, Pim van den Broek, University of Twente

The issues discussed below rely upon an understanding of the distinction between active and passive (classes) objects. An active object is defined as one that has its own thread of control, a passive object is defined as one that has not. This

means, roughly, that an active object has its own flow of control to dispatch requests sent to it. A passive object only performs a computation within the thread of control of an active object. State machines can be defined for active as well as passive objects, that constrain the order in which events can be handled by the object.

Objects communicate by means of call events and signal events, in which, respectively, an operation is called or an asynchronous signal is received. In an *operation call*, the caller calls an operation of another object, the callee. Control passes to the callee and returns when the callee has finished performing the body of the called operation. This is called *run-to-completion* (RTC) semantics. A *shape signal* is a message sent by one object to one or more other objects. Communication by means of signals is asynchronous: The caller can continue processing when the signal is sent. Each active object has a queue of events that have been received but not yet processed.

Event Queue on active objects. The event queue of an active object may be emptied in any order. This opens the possibility that an event in the "queue" is *never* processed by the receiving object. A similar problem exists for the so called change events, which are changes in a condition of the model: these too may be handled any time after their occurrence, or they may never be handled. The concept of event queue should therefore be made more precise by including a fairness requirement, which says that every event in the queue will be handled in a finite time after it entered the queue.

Concurrency semantics of operations. An object may receive several operation calls simultaneously. The UML allows the definition of several concurrency semantics for each operation. One of these is the sequential semantics, which does not guarantee the integrity of the object if several calls to this operation occur simultaneolusly; another is the concurrent semantics, which guarantees the integrity of the object under any number of simultaneous calls of the operation. But consider two operations of an object, one of which is sequential and the other concurrent, that both update the same variable. What happens if both are called simultaneously? Can the concurrent operation still maintain its guarantee if the sequential operation messes up the object? This suggests that the concurrency semantics should not be a property of each operation, but of the object as a whole, that applies to all its operations. This is confirmed by one remark in the UML documentation that seems to suggest that a passive object could be implemented by a monitor (page 2-149); but nevertheless, concurrency semantics is declared for operations rather than for the complete object.

Activity states. An activity state of an object is a state in which the object performs a non-atomic computation. It is defined by entering "do: activity" in the statechart node of that state. Activity states open a can of worms:

- Can a passive object have activity states? We think not. Since a passive object returns control after performing a transition, the activity of such a state would never be performed. For non-basic states in a passive object, however, activities can be defined: During an inter-level transition in a state hierarchy, activities of non-basic states may be started, and when control leaves the passive object, these activities are terminated.
- A state of an active object may have an activity. When is this activity performed? If, after a step, the object has another event in its queue, it will immediately proceed processing this event. But then the activity has no time to run.
- When an object is in an activity state, it will perform the activity defined for the state. But this activity may update local variables (attributes). Does this mean that some active objects may never be in a stable state?
- What happens if two concurrent basic states of an active object are both running activities? Can they update the same attributes? But then activities should have sequential, guarded or concurrency semantics just as operations of passive objects have.
- Suppose a state machine enters a hierarchy of nested states, for all of which activities are defined. A thread can only do one thing at a time. Is the activity of the superstate interleaved with that of the substates? Should it terminate before the activities of the substates commence?

Managing redundant and/or conflicting descriptions of dynamic behaviour. The behaviour of an operation can be specified in many overlapping ways: Transitions on State Machines, Collaborations and associated Interactions, ActivityCharts, pre/post conditions, via Methods, and so on. For example, one can implement an operation by both a method and a state machine transition. The UML does not prohibit this. The method body may involve updates to state variables that conflict with actions triggered by the transition. And even if the method and the transition actions do not conflict, in which order are the actions in the method body and along the transition executed? Are they interleaved? The description on page 2.101 of the semantics document is extremely vague on this.

Further, a class may have a state machine but still have an operation that is defined by a method rather than by a state transition in the machine. In that case, the state machine does not specify the set of all possible behaviors of the class instances, *contra* what is said about state machines on page 2.136, namely that each state machine specifies *all* possible behaviors of the model element to which it is attached.

The previous observation is compounded by the fact that one object may have several state machines (page 2.136). Again, what happens if these two state machines have conflicting behaviors?

The query attribute. A query operation is an operation that does not update its owning object when called. However, the operation may be realized by a

method that nevertheless modifies the state. An obvious fix to this inconsistency is to require the method to be read-only with respect to the owning object. This cannot be statically checked, but we can deal with this similarly to what has been done in the Ada semantics definition, namely by calling models that violate this constraint "erroneous" and leaving tools free to handle these errors in whatever way.

However, this addition is not enough, for the operation can also (even additionally) be realized by a transition in the state machine of the object; and even if no update actions are triggered by the transition, the transition causes a change of state and this itself is an update. A query operation should not be defined by a state transition.

Q6.2. Refinement and extension

Tom Mens, Vrije Universiteit Brussel, Belgium

Intuitively, *refinement* is a mechanism that allows us to gradually add more detail to an arbitrary software artifact. In the case of UML, these software artifacts could be single model elements, parts of a diagram, an entire diagram, or even combinations of (parts of) several diagrams.

Unfortunately, there is some confusion about the exact meaning of the term refinement. Some researchers specifically mean the relationship between model elements at different semantics levels, such as analysis and design. Other researchers only use stepwise refinement to gradually add detail to software artifacts within the same phase. Sometimes, a combination of both is allowed as well. Regardless of the exact definition, a well-defined notion of refinement is crucial in the software engineering process, e.g., to provide better automated tool support for forward engineering as well as reverse engineering.

In [8], there are essentially four different Relationships that can be used as a refinement mechanism: Extend, Include, Generalization and Abstraction.

- The first two relationships are only meaningful in the context of use case diagrams. A UseCase can be an extension of another one, which is represented by an Extend relationship, requiring an ExtensionPoint in the UseCase that is being extended. A UseCase can also be included in another one, which is represented by an Include relationship.
- Another important relationship for expressing some kind of refinement is Generalization, which is defined between two GeneralizableElements (the child and the parent). Examples of GeneralizableElements are Classifiers, Associations, Stereotypes, and Packages. A Generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information. Generalization is a *subtyping mechanism*, i.e., an Instance of the more general GeneralizableElement may be substituted by an Instance of the more specific GeneralizableElement. Generalization also

serves as an *incremental modification* or *inheritance mechanism*. The complete description of a GeneralizableElement is obtained incrementally by combining the descriptions of all its ancestors.
- Finally, an Abstraction is a special kind of Dependency relationship that relates two (sets of) elements that represent the same concept at different levels of abstraction or from different viewpoints. If the stereotype ≪refine≫ is attached to the Abstraction, it specifies a Refinement relationship between model elements at different semantic levels, such as analysis and design. The exact mapping between these different levels, however, is outside the scope of the UML semantics. Alternatively, a ≪realize≫ stereotype can be attached to the Abstraction to model stepwise refinement. Again, the exact mapping of such a Realization is not specified by the UML.

Many problems can be identified with the way in which UML deals with refinement:

Unclear distinction between Refinement and Realization. While the definition of Refinement seems to indicate that Refinement serves specifically as a relation between different semantic levels, parts of [8] contradict this view. For example, there is a detailed description of state machine refinement that specifies how a state machine can be refined stepwise by gradually adding more detail. On the other hand, Realization seems to overlap with Refinement, since it also addresses the link between different semantic levels (specification and implementation). To avoid this confusion, a clear distinction should be made between the two kinds of Abstraction. For example, Refinement could be reserved to express the relation between elements at the same semantics level but at different levels of detail, while Realization could be reserved to express the relation between elements at different semantics levels.

One of the reasons for the above confusion is that the semantics of both Refinement and Realization are not specified by the UML. This task is left entirely to the user or CASE tool developer. To find a commonly accepted definition for refinement and realization, it might be useful to look at existing proposals in the literature. For example, an automatic link between sequence diagrams and statecharts is proposed in [15]. Similarly, in the research community of message sequence charts [11] (a more sophisticated variant of sequence diagrams), attempts are being undertaken to semantically link them to use case diagrams.

Undesired interaction between subtyping & incremental modification. Because the Generalization relationship is actually used for two completely different purposes, subtyping and incremental modification, it suffers from a lack of orthogonality. This leads to the problem that a GeneralizableElement can only be incrementally modified by adding items to it, but not by deleting items from it. Otherwise, the modification will not be substitutable anymore for the more general element. Nevertheless, in many situations it would also be useful to incrementally remove items from a GeneralizableElement.

The main reason for the above problem is that the UML has chosen for a specific variant of inheritance, namely subtype inheritance. As a result, other interesting variants such as implementation inheritance are excluded. In [3], some strong evidence against subtype inheritance is provided. A possible solution would be to separate the orthogonal aspects of Generalization (subtyping and inheritance) into two different relationships.

Lack of orthogonality between Refinement and Generalization. Generalization could be regarded as a specific kind of Refinement. They are both used to incrementally add more detail to a given element. The main difference is that Generalization imposes an extra substitutability requirement. In the metamodel, this commonality should be reflected by defining Generalization as a special kind of Refinement.

Extend and Include versus Generalization. While Extend and Include are used as refinement mechanisms for use cases, these use cases can also be connected by means of a Generalization relationship. The question arises whether such a Generalization relationship is still necessary between use cases. As an answer to this question: the subtyping aspect of Generalization is probably still useful for use cases, but maybe the incremental modification aspect is not, since it can be alternatively achieved via Extend and Include. In any case, the overlap in functionality between Generalization, Extend and Include leads to a lot of confusion.

Unclear definition of Generalization. While it is more or less clear how Generalization is defined for classes and interfaces (it is described in natural language in [8]), the same is not true for other GeneralizableElements such as Nodes, Components, Associations and Packages. [8] does not specify how Generalization should be defined for these elements.

Scaleability of Generalization. The basic notion of Generalization is fairly primitive, because it can only be applied to single GeneralizableElements. However, by using a Generalization relationship between Packages it becomes possible to define a refinement of more complex software artefacts, since Packages can be used to group arbitrarily complex sets of ModelElements. The question arises whether a general meaningful and useful definition can be given to the notion of Generalization between Packages.

Refinement of UML diagrams. UML does not specify how entire diagrams can be refined or realized. For example, when is a class diagram a refinement of another class diagram? The same question holds for all other UML diagrams. Part of the problem is that the UML metamodel does not have an explicit representation for each kind of UML diagram. The metaclasses Collaboration,

Interaction, StateMachine and ActivityGraph can be used for representing collaboration diagrams, sequence diagrams, statechart and activity diagrams, respectively. However, there is no corresponding metaclass for representing a class diagram, object diagram, use case diagram, etc ... Another problem is that the existing metaclasses that represent entire UML diagrams are defined as direct subclasses of ModelElement. To be able to refine diagrams through Generalization, they should at least be subclasses of GeneralizableElement. An even better idea would be to define them as subclasses of Model (a subclass of Package that is intended to describe the modeled system at a certain level of abstraction and from a specific viewpoint). An important open problem that remains is how Generalization (or refinement) should be defined for each kind of diagram. For different UML diagrams, refinement needs to be defined in a different way. State machine refinement will be defined in a completely different way than, say, collaboration refinement or use case diagram refinement. For most kinds of UML diagrams, there is currently no clear or commonly accepted idea of what refinement should look like. It might be useful to look at other approaches (formal as well as informal) where refinement has already been investigated. Especially in the area of theoretical computer science, there is a significant number of people working on formal refinement techniques for many different kinds of models, and using many different kinds of underlying formalisms. Many of these ideas might be relevant to the UML.

Q6.3. Aggregation

Andy Evans, University of York, UK
Robert France, Colarado State University, USA
Guy Genilloud, Swiss Federal Institute of Technology
Brian Henderson-Sellers, University of Technology, Sydney, Australia
Perdita Stevens, University of Edinburgh, UK

Aggregation has long been a thorny issue; the problem is that there seem to be so many subtle nuances to the concept. In UML things appear to be more clear cut at first: there are, essentially, two forms, represented by the black diamond and white diamonds. The intention would be to partition *all* whole-part relationships into two distinct, non-overlapping and exhaustive kinds. Furthermore, examination of the standard suggests that the black diamond *was intended* to have precise semantics, whilst the white one was probably intended to cover all the other cases (of whole-part).

Although the semantics of the black diamond should not be difficult to provide (the UML standard suggests a *deletion* semantics, where, if the whole is copied or destroyed, the parts are as well), a semantics for the white diamond will open the same can of worms that have been opened in the past, which, in turn, may raise questions about whether the black diamond is a useful or appropriate concept to treat as a special case. Problems with the definitions of UML's black and white diamonds are discussed further in [9]. Attempts to tease out the

various subtle aspects of whole-part are provided in e.g. [1, 16, 9]. Attempts to formalize such analyses are provided in e.g. [14, 10].

References

[1] F. Civello. Roles for composite objects in object-oriented analysis and design. In *OOPSLA'93 Conference Proceedings*, ACM SIGPLAN Notices 23:10, October 1993.

[2] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. Defining uml family members using prefaces. In C. Mingins and B. Meyer, editors, *Proceedings of TOOLS Pacific 99*. IEEE Press, 1999.

[3] W.R. Cook, W.R. Hill, and P.S. Canning. Inheritance is not subtyping. *ACM Transactions on Programming Languages and Systems*, pages 125–135, 1999.

[4] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[5] A.S. Evans and S. Kent. Meta-modelling semantics of UML: the pUML approach. In B. Rumpe and R.B. France, editors, *2nd International Conference on the Unified Modeling Language*, 1999.

[6] Robert Geisler. Precise UML semantics through formal metamodeling. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[7] Martin Gogolla and Mark Richters. Equivalence rules for UML class diagrams. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998*, pages 87–96. ESSAIM, Mulhouse, France, 1998.

[8] Object Management Group. UML specification version 1.3. Technical Report ad/99-06-08, June 1999.

[9] B. Henderson-Sellers and F. Barbier. Black and white diamonds. In B. Rumpe and R.B. France, editors, *2nd International Conference on the Unified Modeling Language*, 1999.

[10] B. Henderson-Sellers and F. Barbier. What is this thing called aggregation? In R. Mitchell, A.C. Wills, J. Bosch, and B. Meyer, editors, *TOOLS29*, pages 216–230. IEEE Computer Society Press, 1999.

[11] E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on message sequence charts (msc'96). Tutorials at First Joint Conference FORTE/PSTV'96, Kaiserslautern, Germany, October 1996.

[12] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Guide*. Addison-Wesley, 1998.

[13] Bernhard Rumpe. A note on semantics (with an emphasis on UML). In Haim Kilov and Bernhard Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 177–197. Technische Universität München, TUM-I9813, 1998.

[14] Monika Saksena, Robert France, and Maria Larrondo-Petri. A characterization of aggregation. In C. Rolland and G. Grosz, editors, *Proceedings of OOIS98*, pages 11–19. Springer, 1998.

[15] S. Schnnberger and R.K. Keller. Algorithmic support for model transformation in object-oriented development. Publications of DIRO, August 1997.

[16] M. Snoeck and G. Dedene. Existence dependency: the key to semantic integrity between structural and behavioural aspects of object types. *IEEE Trans. Software Eng.*, 24(4):233–251, 1998.