Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002
ã 2002 Society for Design and Process Science

LOOSE SEMANTICS FOR UML/OCL

Heinrich Hussmann
Fakultät für Informatik, Technische Universität Dresden, Germany
Email: hussmannh@acm.org

ABSTRACT

This paper deals with formal foundations for a subset of the UML notation (subset of class diagrams and constraints in OCL). There are already various proposals for semantics of UML and a few for OCL. Nevertheless, it is argued that these approaches are not fully adequate for building a conceptual bridge between the programming artifacts produced from UML/OCL and the formal semantics. A different approach is suggested which builds on one hand on the existing work on UML/OCL semantics, and which on the other hand re-introduces ideas, which were discussed for abstract data types under the term "loose semantics". The advantages of this approach are presented and a relationship to the meta-modeling approach is outlined.

1. INTRODUCTION

The Unified Modeling Language (UML) [UML1.4] has quickly established itself as the prevailing standard for software modeling in industrial practice. There is no need to further discuss the advantages and problems of UML here. Moreover, basic familiarity of the reader with UML is assumed.

From the beginning, it was one of the goals of the UML development to provide a more precise description of the language than it was available for other software modeling notations in the past. Specifically, an OMG press release from 1997 [OMG97] mentions as number 4 of the seven design goals for UML: "Provide a formal basis for understanding the modeling language." This goal has been reached by now only in a very limited way, that is regarding the *syntax* of UML. The UML standard contains the notation of the Object Constraint Language (OCL), and this language is used to define well-formedness rules on a meta-model for the UML abstract syntax. However, until now a formal semantics is not part of the standard. There exist several proposals for defining such a semantics, some of which are reviewed further below. However, the practical acceptance of these approaches is low. The general claim of this paper is that one reason for this is that the existing approaches to UML/OCL semantics do not use the right abstraction level. In particular, existing semantics are not very helpful for people who want to define practical tool for UML, for instance code generation.

The paper is structured as follows. First, the variety of different possible implementations for a UML class diagram is discussed, in order to get an idea of the needed abstraction level. Then the existing approaches are revisited and analyzed. This completes section 2. In section 3, a novel approach to UML semantics based on so-called "object algebras" is presented. It is shown that this approach is a generalization of the various ideas for code generation, and that it can be used to precisely define semantics for OCL expressions as well. Finally, it is outlined how the meta-modeling approach can be combined with the described technique.

2. EXISTING APPROACHES TO UML /OCL SEMANTICS REVISITED

In order to explain various ways of interpretation of a given UML class diagram, we are using a simple example UML class diagram, which is depicted in Figure 1.

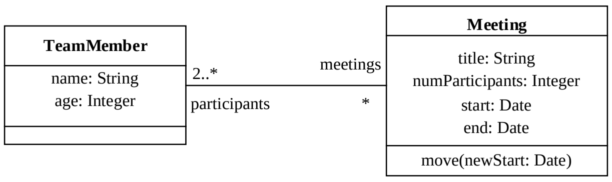


Figure 1: Example UML Class Diagram

2.1 PROGRAMMERS' APPROACH

As a start to the discussion of semantic models, it is helpful to analyze how class diagrams are mapped to programming languages in practice. First let us assume the viewpoint of a Java programmer. A standard mapping to Java code, which can be achieved with CASE tools (at least in principle, ignoring some of the weaknesses of most existing UML CASE tools) is as follows. We assume here a user-defined type *Date*. The type name *java.util.Set* refers to a standard library interface for finite sets from the so-called Java Collection Framework.

```
class TeamMember {
  private String name;
  private int age;
  private java.util.Set meetings;
   //element type Meeting
}

class Meeting {
  private String title;
  private int numParticipants;
  private Date start;
  private Date end;
  private java.util.Set
     participants;
   //element type TeamMember
```

```
    public void move (Date newStart)
     {}
  }
```

This is just one of the possible implementations of the UML class diagram. Let us now assume that another programmer provides a handcrafted implementation of the same class diagram. It features a special class, which manages the association between *TeamMember* and *Meeting* objects.

```
  class TeamMember {
    private String name;
    private int age;
  }

  class Meeting {
    private String title;
    private int numParticipants;
    private Date start;
    private Date end;
    public void move (Date newStart)
     {}
  }

  class MeetingManager {
    static java.util.HashMap
        participation;
      //maps Meeting objects onto
      //sets of Participant objects
  ...
  }
```

Even without going into the details of the second implementation, it should be obvious that the rather abstract UML model is refined in rather different ways in the two approaches. The situation becomes more complex if one considers different target languages than Java. For instance, in some applications the *Meeting* class may be mapped onto a CORBA interface definition, which is suitable for accessing *Meeting* objects independently of their concrete implementation language. The corresponding code in CORBA IDL (Interface Definition Language) essentially looks as follows.

```
  interface Meeting {
    attribute string title;
    attribute long numParticipants;
    attribute Date start;
    attribute Date end;
    void move (in Date newStart);
  }
```

Besides the minor syntactical differences, there is also a semantic difference to the Java code from above. The keyword *attribute* in CORBA IDL does not explicitly prescribe the existence of a variable in the implementation, which stores the relevant information. Rather the *attribute* declaration in the interface states the existence of two operations, which are called the *getter* and *setter* operation of the attribute. The exact syntactic representation of these operations is dependent on the actual programming language used for the implementation. It is possible, but not required, that the implementation provides a variable corresponding to the attribute, The key point is here that the UML class diagram contains an attribute, e.g. *numParticipants*, but that the implementation may implement this in a different way than just by providing a variable.
A final example of a completely different mapping to code is a translation into a database schema. For instance, the following SQL code may be generated from the UML class diagram in Figure 1. The attribute *numParticipants* is not represented directly since it can be derived from other information. (Of course, this decision is too difficult for most automated translation tools, but nevertheless it is adequate.)

```
  create table TEAMMEMBER
  (TMID integer primary key,
   NAME varchar not null,
   age integer
  )

  create table MEETING
  (MTID integer primary key,
   TITLE varchar not null,
   START DATE,
   END DATE,
  )

  create table TM_MTGS
  {TMID integer references
     TEAMMEMBER,
   MTID integer references MEETING,
   primary key (TMID,MTID)
  )
```

The above database schema consists of three tables, one for each class and one for the association. The association is a table just holding pairs of foreign keys to the class tables. Please note that there exist many different strategies for mapping a UML class diagram to a database schema.

The examples given above just cover a few of the possible mappings of UML to target languages. There are for instance approaches to map UML class diagrams to sets of Web pages, to XML structures or to physical or organizational entities.

One of the main advantages of the UML notation is that it provides a common abstraction for all the very different implementations, which were sketched above. Using the OCL language of UML, semantic constraints can be formulated, which have to be translated into the various possible implementations in a very different way. A simple example for the class diagram from Figure 1 is the rule that the UML attribute *numParticipants* of class *Meeting* has always to be consistent with the actual number of associated *TeamMember* objects. In OCL syntax, this reads as follows. Please note that the OCL keyword "inv" denotes an invariant.

```
context Meeting inv:
numParticipants =
  participants -> size
```

2.2 NAIVE SET-THEORETIC SEMANTICS

Now let us contrast some of the existing approaches to UML/OCL semantics with the wide spectrum of implementations discussed in the previous subsection. For instance in [RG98], it is suggested to use a simple set-theoretic interpretation for UML class diagrams. This basic idea has also been chosen as the basis for the first approach to officially define the semantics of OCL in the context of the OMG [OCL2]. In this approach, the semantics of the class diagram from Figure 1 is described as a set of hypergraphs, where a hypergraph corresponds to a concrete configuration of object instances (a system state). The nodes correspond to objects and the edges to association links. The nodes are labeled with attribute/value mappings and the edges with names for association ends.

This approach has the strong advantage that it is accessible in principle to everybody who has a decent background in mathematics and that it is precise enough to define formally the semantics of OCL, as it is done in [RG98, RG01, OCL2].

There is, however, a serious drawback of this naive approach. Objections come from two sides: academics criticize the low abstraction level (and this paper somehow belongs to this category), but also practicioners are not happy with the fact that the semantics of UML class diagrams is fixed to a particular structure. For each one of the implementations mentioned in section 2.1 above, a non-trivial mapping is required to understand it as a hypergraph structure. For instance, attribute/value mappings do not exist in all implementations of a UML class diagram (take the translation to CORBA interfaces). So we have the situation that the concepts of UML itself are more abstract than the formal semantics given to it. This situation is common also in other areas (for instance in operational semantics of programming languages), but it is also considered unsatisfactory there (which is the reason why denotational semantics for programming languages were invented).

2.3 TRANSLATION SEMANTICS

There is a second group of approaches to the semantics of UML, which tries to keep the right abstraction level. These approaches define translations from UML class diagrams to traditional specification languages. There exist, among others, translations to Z [FB+97], Object-Z [KC99], Larch [BG98] and CASL [HCB00]. Also other UML diagram types have been translated to formal notations, e.g. in [BCR00] using Abstract State Machines. The exact abstraction level reached in these translations differs. For instance, the Z-based work tends to describe the structure of the implementation more closely, according to its "model-based" paradigma.

There are a number of serious problems, however, with these approaches. A pragmatic, but nevertheless vary important, issue is that the semantics is not accessible to anybody who does not have the necessary background in the chosen formal specification language. Due to the low acceptance of formal specification languages in the practically-oriented software engineering community and due to the large variety in formal specification languages, this reduces the practical impact of the approach already almost to zero. Moreover, there is the technical problem of maintaining two versions of the same specification, one in the UML formalism and one in the formal specification language, which is its translation. Techniques known from code generation can be used to keep both versions in synchronicity. Significant overhead is required to re-translate results of analysis tools from translated version back to UML. And finally, there is a third source of problems which stems from the notoriously weak relationship of formal specification languages to real-world programming languages. If, for instance, a CASL specification is created from a UML specification, the relationship to the various practical implementations, as they were presented in section 2.1 above, remains either unclear or at least relatively complex. So in summary, this style of semantics provides only little value to UML tool builders.

2.4 METAMODELING SEMANTICS

The third group of approaches for UML semantics we discuss here is based on the application of a "bootstrapping" principle [CEKS01]. The semantics of UML is described using UML itself (and its constraint language OCL). This is particularly appealing to the UML community, since the standard document defining UML itself relies on a meta-modeling approach, where the syntax of UML is defined by a meta-model in UML (in fact, in the so-called MOF framework, but this will become a proper sublanguage of UML soon).

Since the meta-modeling principle will be discussed in more detail below, a simple example for meta-modeling semantics may be helpful. In principle, a general (meta-)model for possible object configurations (system states) is defined in UML. For instance, the UML class diagram in Figure 2 describes on the meta-level a class of object instances which contains so-called slots, each of which has a name and a value (of a universal type *Any*). The slots are used to define the semantics of attributes and associations.

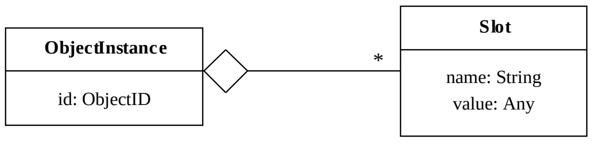


Figure 2: Simple Semantic Meta-Model for UML Classes

Based on such a semantic meta-model, an object configuration of the class diagram from Figure 1 representing one meeting with three participants will consist of the following objects:

- Four instances of the *ObjectInstance* meta-class (one for the meeting and three for the team members)
- Ten instances of the *Slot* meta-class to represent attributes (four slots for the attributes of the meeting object, three times two slots for the attributes of the team members)
- Depending on the actual way how semantics of associations is defined, a certain number of additional *Slot* objects and possibly further objects to represent the association links between the meeting and the three participants.

It is obvious that instantiations of the semantic meta-model become extremely large. But this is not the most serious problem, since this information can be processed by machine. The main problem is again in the level of abstraction. In fact, the meta-modeling approach is not much different in its level of abstraction from the naive approach discussed above. For instance, in the current proposals for meta-modeling semantics, attributes are mapped to meta-level object instances like slots. But is this really adequate for implementations that possibly implement attributes by pairs of operations (as it was discussed above)? And moreover, there is a variety of possible meta-modeling semantics for UML class diagrams, for instance depending on the way how association links are mapped to slots (and on the question whether they are mapped to slots or some other meta-class). This is also strange for an abstract semantics, which should provide a common abstraction for many different concrete implementations.

3. OBJECT ALGEBRAS AND LOOSE SEMANTICS

Following the harsh criticism of existing approaches in the previous section, it is now time to outline a new approach which tries to overcome the disadvantages of the existing approaches but builds on their advantages. The difference to the other approaches even may look small at first sight, but there are serious advantages which will be explained below.

The problem of the first and third group of approaches to semantics discussed above was that the abstraction level was inadequate. In terms of software engineering, the structure of the semantic domain is overspecified in these approaches. A counter-measure is to adhere to the old principles of modular design and to apply information hiding. So we define only the necessary access operations to the semantic domain and their essential properties. This is very similar in spirit to the translation approaches (second group in the discussion above). However, we avoid the overhead of using a formal specification language. Instead, we give a more abstract semantics in two versions: First a direct semantics based on plain mathematical set theory, then a sketch of a meta-modeling approach to the same concept.

3.1 UML CLASS DIAGRAMS AS ALGEBRAIC SIGNATURES

Before proceeding with any formal definitions, the information contained in a UML class diagram has to be represented as a formal structure. In order to maintain good compatibility with the other approaches, we use here a definition very similar to [RG98]. In our definition we try to stay a little bit closer to the syntactic information given in the diagram and avoid to include information pointing already to the intended semantics. Moreover we extend the definition to cover non-query operations.

Definition. A *UML class model* over a set *P* of primitive type symbols is a structure *M* = (*C*, <, *ATT*, *OP*, *ASSOC*) where

- *C* is a set of class names. We use below the abbreviation *T* = *C*   *P* for all type names.
- < is a partial order on *C* reflecting the generalization hierarchy.
- *ATT* is a set of attribute symbols. An attribute symbol is a triple (*c*, *a*, *t*) where *c*   *C*, *a* is an attribute identifier and *t*   *T* is a type symbol.
- *OP* is a set of operation symbols. An operation symbol is a quintuple (*c*, *op*, *args*, *t*, *q*) where *c*   *C*, *op* is an operation identifier, *args*   *T** is the list of argument types, *t*   *T*   {void} is the result type ("void" indicates the non-existence of a result). *q* is a Boolean value indicating whether the operation is a "query" (which leaves the system state unchanged).
- *ASSOC* is a set of association symbols. An association symbol is a triple (*c1*, *ass*, *c2*) where *c1*   *C*, *c2*   *C* are the two related classes, and *ass* is the name of the association. '

As it was observed for instance in [RG01], the information contained in a UML class model is very similar to a signature in the sense of algebraic specifications. The next subsection takes this observation one step further and defines specific algebras for this type of signature.

3.2 OBJECT ALGEBRAS

The following definition can be seen from two viewpoints. From the viewpoint of universal algebra, it defines a specific kind of heterogeneous algebra, which is specifically adjusted to the purposes of UML models. From the viewpoint of a person interested in the semantics of a UML model, it constitutes an attempt to provide an optimally abstract definition of what an implementor of a class diagram has to provide.

Definition. Given a UML class model $M = (C, <, ATT, OP, ASSOC)$ over a set P of primitive type symbols, an *object algebra* for M is a structure

$A = (U_A, ID_A, P_A, A_A, OP_A, LNK_A)$ where

- U_A is a set called the *(object) universe*, the elements of which are called *object configurations*.
- ID_A is a countably infinite set of *object identifiers* such that for each identifier in a given object configuration the class of the identifier is known, i.e. there is a function

  $class\colon\ U_A \times ID_A \to C.$
- P_A is a family of sets containing for each primitive type $p \quad P$ a carrier set p_A.

  The set V_A of *values* is defined as an abbreviation for the rest of the definition:

  $$V_A =_{\text{def}} ID_A \quad \bigcup_{p \quad P_A} p_A \quad \{\text{null}\}$$
- A_A is a family of functions containing for each attribute name a in ATT a partial function

  $a_A\colon U_A \times ID_A \to V_A$.
- OP_A is a family of functions containing for each operation name op in OP a partial function $op_A\colon U_A \times ID_A \times V_A \times \ldots \times V_A \to U_A \times V_A$ such that for a query operation $op(u, i, x_1, \ldots, x_n) = (u, v)$ holds (i.e. u is unchanged) and such that only the "null" value is returned for the "void" type.
- LNK_A is a family of functions containing for each association name *ass* in *ASSOC* a total function $ass_A\colon U_A \times ID_A \times ID_A \to$ Bool.

This definition still looks complex, but the remaining complexity is unavoidable due to the inherent complexity of UML class diagrams. It does not talk at all about the structure of an object configuration (the system state), but just says that there are some observations which are applicable to such an object configuration (an element of U). One can ask for the value of an attribute, execute an operation for an object and ask whether two objects are linked via an association. The term *object universe* denotes all possible object configurations (global system states) for a given class diagram. Please note that an operation always transforms a whole object configuration and not an individual object, and therefore operations may modify other objects than the one on which the operation was invoked, and may for instance establish or remove links between objects.

An example may be helpful at this stage. Assume that object configuration u_1 contains two objects, one for each of the classes *meeting* and *TeamMember*. The observation $class(u_1,1)$ delivers, for instance the class *TeamMember* and $name(u_1,1)$ delivers for instance the value "john", indicating that the object identifier 1 denotes an object of class *TeamMember* which carries the name "john". Analoguously, other attribute values are available, and , for instance, identifier 2 may denote a *Meeting* object. Assume the objects are not yet linked, as stated by $meetings(u_1, 1, 2) = false$ (taking *meetings* as the name for the association shown in Figure 1). Now an operation *addMeeting* on class *TeamMember* may establish a link between the two objects. In the object algebra this means that $addMeeting(u_1, 1, 2)$ gives a new object configuration u_2 which has exactly the same properties as u_1 but where $meetings(u_2, 1, 2) = true$.

The above definition does not yet contain all conditions one wants to put on a semantic structure for UML class diagrams. It contains already a way to determine the class of the object an identifier is intended to refer to, but of course accesses to the attributes etc. need to be consistent with this information. (Side remark: We are assuming here a subset of UML, where objects always belong to a single class. For full UML, the function *class* from above needs to be slightly generalized.)

In order to decide whether some access function is type-correct for an object identifier, we have to deal, of course, with the complex issues of inheritance and dynamic binding. The following definition is also in this respect more open than [RG98] (where a simple inclusion semantics for subclasses is assumed). The exact binding mechanism is kept open; just minimal conditions are specified.

Definition. Given a UML class model $M = (C, <, ATT, OP, ASSOC)$ over a set P of primitive type symbols, an *object algebra* $A = (U_A, ID_A, P_A, A_A, OP_A, LNK_A)$ for M is called *type-conformant* if the conditions given below are satisfied.

For the following conditions, we use an auxiliary predicate *isKindOf* which is defined as follows:

$isKindOf\colon\ U_A \times ID_A \times C \to$ Bool

For every $u \quad U_A,\ i \quad ID_A,\ c \quad C$:

$isKindOf(u,i,c) \Leftrightarrow\ class(u,i) <^* c$

where $<^*$ denotes the transitive-reflexive closure of the syntactic inheritance relation.

- For each attribute in ATT with name a of each class $c \quad C$ with result type $t \quad T$:
  - $a_A(u, i)$ is defined if $isKindOf(u,i,c)$
  - $a_A(u, i)$ is conformant to t in u, if it is defined;
- For each operation in OP with name op of each class $c \quad C$ with argument types

  $w = < t_1, \ldots, t_n > \quad T^*$ and result type $t \quad T$:
  - if $op_A(u, i, x_1, \ldots, x_n) = (u', x)$ is defined, then $isKindOf(u,i,c)$ and for all k $\quad \{1,\ldots,n\}$ the value x_k is conformant to t_k in u and the value x is conformant to t in u';
- For each association in *ASSOC* with name *ass* and starting class *c1* and end class *c2*
  - $ass_A(u, i1, i2)$ = true implies $isKindOf(u,i1,c1)$ and $isKindOf(u,i2,c2)$.

The above definition makes use of the notion of a value being conformant to a type. We avoid here to go into the details of this question, since this is well-known area.

3.3 LOOSE SEMANTICS

The term *loose semantics* is borrowed from the theory of algebraic specifications [BW82]. The basic idea can be easily transferred to UML:

> The semantics of a UML class diagram is constituted by *all* object algebras that are type-conformant to the class diagram.

The basic idea of the loose semantics is that all possible implementations of the UML class diagram are covered. There is no single definition of the structure of the semantic domain. So for instance,

· The set theoretic semantics from [RG98] constitutes an object algebra which is contained in the loose semantics of the class diagram.
· All other approaches to code generation as mentioned in section 2.1 above constitute specific object algebras , which differ in their way how the algebra functions are implemented.

Using the loose semantics approach, the statements above can be verified in a formal setting. For instance, the fact that the semantics of [RG98] provides an object algebra can be proven mathematically (although this proof is relatively trivial). The definition of object algebras was chosen in such a way that standard object-oriented programs or semantic models for database schemas are covered as well. So it is just a matter of checking the conditions form above whether the implementation belongs to the semantics of the UML class diagram or not.

At this point, it may be helpful to clarify the relationship between the approach described here an the approach of so-called *evolving algebras* [Boe95], more recently renamed to *Abstract State Machines*. The general idea is very similar, since also evolving algebras provide a built-in global state. However, in our approach we assume UML(+OCL) to be the language which is used by the specifier, and do not see the need for another language layer. Moreover, a technical difference is that in our approach an algebra maintains all possible states, whereas in evolving algebras the state is identified with an algebra. This makes it easier for us to define loose semantics with multiple models for a specification (a single model/algebra contains all states in a certain implementation style) whereas in evolving algebras there is essentially one fixed model.

3.4 SEMANTICS OF OCL CONSTRAINTS

Since the information carried in plain UML class diagrams is very much limited, it is of course necessary to extend the discussed approach to more sophisticated specifications involving OCL constraints. Some of the aspects of class diagrams which were ignored above can also be easily handled through OCL constraints (e.g. multiplicities, see [GR01]).

Using an analogy to the classical theory of algebraic specifications, OCL constraints play the very important role of "axioms" which further restrict the admissible algebras in the semantics of a syntactical specification for a specification, starting from the type-conformant algebras. Exactly the same approach is used here.

The following definition gives a simple interpretation for a significant subset of OCL expressions in terms of object algebras.

Definition. Assume a given UML class model, a set X of typed variables and a binding $Env: X \rightarrow V$ of variables to values. Then for any type-conformant object algebra A based on this class model, the interpretation of OCL expressions is a function

$\mathbf{I}_A : Exp \times U \times Env \rightarrow V \quad$ {undefined}

The value of the interpretation function is defined recursively for OCL expressions as follows.

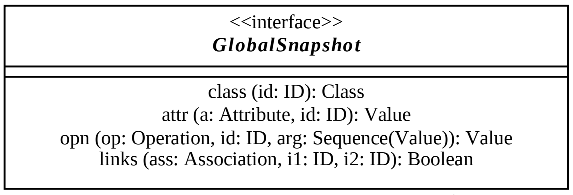
· Variable: $\mathbf{I}_A\,(x, u, e) = e(x)$
· "let": $\mathbf{I}_A\,(\mathbf{let}\ x = exp1\ \mathbf{in}\ exp2, u, e) =$
$\mathbf{I}_A\,(exp2, u, e+\{x \rightarrow \mathrm{I}_A\,(exp1, u, e)\})$
· Attribute: $\mathbf{I}_A\,(exp.a, u, e) = a_A(u, \mathbf{I}_A\,(exp, u, e))$
· Query: $\mathbf{I}_A\,(exp.op(\mathrm{e}xp_1, \ldots, \mathrm{e}xp_n), u, e) =$
$op_A(u, \mathbf{I}_A\,(exp, u, e),$
$\mathbf{I}_A\,(exp_1, u, e), \ldots,$
$\mathbf{I}_A\,(exp_n, u, e))$
· Navigation along an association *ass* using an association end ass_i:
$\mathbf{I}_A\,(exp.ass_i, u, e) =$
$\{j \quad ID_A \mid ass_A(\mathbf{I}_A\,(u, exp, u, e), j) = \mathrm{true}\}$
· Primitive operations:
$\mathbf{I}_A\,(op(\mathrm{e}xp_1, \ldots, \mathrm{e}xp_n), u, e) =$
$op_A(\mathbf{I}_A\,(exp_1, u, e), \ldots, \mathbf{I}_A\,(exp_n, u, e))$

In order to be practically applied, the semantics of OCL expressions needs to be combined with a semantics for the way how the OCL constraints are attached to the UML model (e.g. invariant, precondition, postcondition, guard). Since the precise semantics of this aspect is a difficult topic in its own (see [HHB01]) and there is the additional technicality of providing an appropriate comparative interpretation involving pre- and post-states for an operation (see e.g. [RG01]), we omit the whole topic here. However, the extension exists and works smoothly (see [Hus02]).

The main contribution of the above definition of OCL semantics is that it is independent of the concrete implementation chosen. So it can be taken as the basis, for instance, to define detailed mappings from OCL constraints to programming languages or other frameworks (see e.g. [HDF00, DHL01]).

3.5. LOOSE SEMANTICS AND META-MODELING

It may appear as a disadvantage of the object algebra approach that it is again based on plain mathematics. The number of indices and brackets in the formulae alone probably makes the style of presentation inaccessible for some of the audience interested in UML. Therefore, it is interesting to see whether the same approach can also be formulated by using meta-level UML notation. Fortunately, this is not difficult. In fact, the notion of an object algebra can be encoded as a UML *interface*, as shown in Figure 3.



```
<<interface>>
GlobalSnapshot

class (id: ID): Class
attr (a: Attribute, id: ID): Value
opn (op: Operation, id: ID, arg: Sequence(Value)): Value
links (ass: Association, i1: ID, i2: ID): Boolean
```

Figure 3: The GlobalSnapshot Interface

The *GlobalSnapshot* interface provides a high-level view of a system state from the outside. It makes reference to classes from the syntactic meta-model of UML (like *Class*, *Operation*, *Attribute*) and to special sorts, which are assumed to be available in the implementation (like *ID* for object identifiers and *Value* as a supertype of all possible values). The parameter *U*, which was present in the object algebra definition, can be omitted from the signatures here, since the interface always works on some given snapshot (in object-oriented style). The *opn* operation, which invokes a model-level operation, may change the snapshot by side effect, leading to new values for the internal state of the snapshot. Of course, some more information from the definition above have to be added, and this can be done by using OCL constraints on meta-level (for instance to ensure type-conformance). Due to space limitations, it is not possible to go into further detail here. The most important aspect is that under this viewpoint now a meta-modeling semantics as shown in Figure 2 above can be understood as the meta-model of a specific implementation approach, for instance one targeting at the implementation in an objetc-oriented programming language. Precise constraints relating the implementation meta-model and the *GlobalSnapshot* in OCL can be given which specify the chosen semantic variant of class diagram implementation. This approach may provide a useful framework for the definition of *implementation mappings*, as it is discussed for the 2.0 version of UML.

4. CONCLUSION AND OUTLOOK

In this paper, we have provided an analysis of existing approaches to the semantics of UML class diagrams and an approach, which tries to apply the right level of abstraction for such a semantics, such that the full intended spectrum of implementations can be covered. We have also shown that the approach is not in contradiction to the meta-modeling approach but can be used as a helpful tool for relating the abstract general semantics of class diagrams to more specific semantic domains of individual implementations.

Of course, the work presented here provides more a starting point than a complete result. For instance, the practicability of the specification of implementation mappings has to be proven. One clear drawback of the presented approach is that is considers all classes being *passive*, and avoids all issues related to concurrency. The most challenging topic is therefore the transfer of the idea to a more general understanding of UML, including other diagrams than class diagrams. However, experience from previous work, which has followed similar ideas in a different setting [Hus97] gives some indications that this transfer will be possible.

Another interesting direction of further research is to consider the embedding of traditional specification techniques, like abstract data types, into the newly proposed approach. In fact, classical axiomatic specifications of data types should be in some sense a special case of the semantics as described above. Together with proof calculi for UML/OCL (as they are being developed at various places), there is a chance that a sublanguage of UML/OCL may be defined which is a full-featured formal specification language in itself. And therefore may replace in many practical applications the usage of traditional formal methods. This paper has tried to contribute to this goal by giving a direct and abstract semantics for such a sublanguage of UML/OCL.

REFERENCES

[Boe95] Egon Börger, Why Use Evolving Algebras for Hardware and Software Engineering?, in M. Bartosek, J. Staudek, J. Wiedermann (eds.). Proc. SOFSEM '95, Springer LNCS Vol. 1012, pp. 236-271, Berlin 1995.

[BCR00] E. Börger, A. Cavarra, and E. Riccobene, An ASM Semantics for UML Activity Diagrams, in T. Rus (ed.), AMAST 2000 Proc., Springer LNCS Vol. 1816, pp. 293-308, Berlin 2000.

[BG98] M. Brickford, D. Guaspari, Lightweight Analysis of UML. Draft Technical Report, Odyssey Research Associates, 1998.

[BW82] M. Broy, M. Wirsing, Partial abstract data types. *Acta Informatica* 18(1):47--64 (1982).

[CEKS01] T. Clark, A. Evans, S. Kent, P. Sammut, The MMF Approach to Engineering Object-Oriented Design Languages, Proc. Worksop on Language Descriptions, Tools and Applications (LDTA2001), 2001. (Available via http://www.puml.org)

[DHL01] Birgit Demuth, Heinrich Hussmann, Sten Loecher: OCL as a Specification Language for Business Rules in Data Base Applications, in: Gogolla/Kobryn (Eds.), <<UML>>2001 Proc., Springer, LNCS Vol. 2185, Berlin 2001.

[FB+97] R. France, J.-M. Bruel, M.M. Larrondo-Petrie, M. Shroff, Exploring the Semantics of UML Type Structures with Z, In: Proc. 2nd IFIP FMOODS Conference, pp. 247-260, London 1997.

[GR01] M. Gogolla and M. Richters. Expressing UML Class Diagrams Properties with OCL. In Clark/Warmer (eds): Advances in Object Modelling with the OCL, Springer LNCS Vol. 2263, pp 86-115, Berlin 2001.

[HHB01] R. Hennicker, H. Hussmann, M. Bidoit, On the Precise Meaning of OCL Constraints. In: Clark/Warmer (eds): Advances in Object Modelling with the OCL, Springer LNCS Vol. 2263, pp. 70-85, Berlin 2001.

[Hus97] H. Hussmann, Formal foundations for Software Engineering Methods, Springer LNCS Vol. 1322, Berlin 1997.

[Hus02] H. Hussmann, Formal Specification of Software Systems, Lecture Notes, http://www-st.inf.tu-dresden.de/fs, 2002

[HCB00] H. Hussmann, M. Cerioli, H. Baumeister, From UML to CASL (Static Part), Technical Report of DISI - Università di Genova, DISI-TR-00-06, Italy, 2000.

[HDF00] Heinrich Hussmann, Birgit Demuth, Frank Finger, Modular Architecture for a Toolset Supporting OCL, In: Evans/Kent/Selic (eds). <<UML>>2000 Proc., Springer LNCS Vol. 1939, pp.278-293, Berlin 2000.

[KC99] S.-K. Kim, D. Carrington, Formalizing the UML Class Diagram Using Object-Z. In: France/Rumpe (eds.), Proc. <<UML>>'99, Springer LNCS Vol. 1723, pp. 83-98, Berlin 1999.

[OMG97] OMG Press release "What is OMG-UML and Why Is It Important?", 1997. http://cgi.omg.org/news/pr97/umlprimer.html

[RG98] M. Richters and M. Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, Proc. 17th Int. Conf. Conceptual Modeling (ER'98), pages 449-464. Springer LNCS Vol. 1507, Berlin 1998.

[RG01]M. Richters and M. Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, Advances in Object Modelling with the OCL, pages 43-69. Springer LNCS Vol. 2263, Berlin 2001.

[UML1.4] Unified Modeling Language Specificationb, Version 1.4, OMG Document formal/01-09-67, 2001.

[OCL2] J. Warmer et al., Response to the UML 2.0 OCL RfP, Initial Submission, Version 1.0, OMG Document ad/2001-08-01, 2001