FORMALLY MODELING UML AND ITS EVOLUTION: A HOLISTIC APPROACH

Ambrosio Toval Álvarez
*Software Engineering Research Group, Department of Informatics: Languages and Systems, University of Murcia, Spain**
atoval@dif.um.es

José Luis Fernández Alemán
Software Engineering Research Group, Department of Informatics: Languages and Systems, University of Murcia, Spain
aleman@dif.um.es
To my father

Abstract Due to the pervasiveness of diagrams in human communication and because of the increasing availability of graphical notations in Software Engineering, the study of diagrammatic notations is at the forefront of many research efforts. The expressive power of these kinds of languages and notations can be remarkably improved by adding extensibility mechanisms. Extensibility, the ability of a notation or a modeling language to be extended from its own modeling constructs, is a feature that has assumed considerable importance with the appearance of the UML (Unified Modeling Language). In this paper, a holistic proposal to formally support the evolution of the UML metamodel is presented. To attain this aim, an algebraic formalization is provided which leads to a seamless formal model of the UML four-layer semantics architecture. These two characteristics - being holistic and seamless together with reflection are the most innovative aspects of the research with respect to formalizing the UML. In particular, a central role is played by reflection. A formal language supporting this feature called Maude is studied and put forward as the basis for the formalization of the UML extensibility mechanisms. Since Maude is an executable specification language, the final set of formal models can also be used as a UML virtual machine at the specification level. To illustrate the approach, a UML Class Diagram prototype is implemented using the Maude interpreter. The

*Granted by the CICYT (Science and Technology Joint Committee), Spanish Ministries of Education and Industry, research project TIC97-0593-C05-02 MENHIR, OM

integration of this Maude prototype with a UML commercial CASE has been developed, in Java, and is currently working.

Keywords: Extension Mechanism, Algebraic Specification, Maude, UML Evolution, Metaprogramming.

1. INTRODUCTION

Object-Orientation is one of the most prominent approaches in current software development. Within this paradigm, the UML [16, 17] is the standard OO notation adopted by the OMG[1] and is becoming an unavoidable reference in information systems Analysis and Design.

The UML graphical notation supports extensibility mechanisms by means of three extension mechanisms: *Stereotype*, *Constraint* and *Tagged-Value*. A stereotype represents a subclass of an existing modeling element, such as a message, an operation, a class, a use case, and a data type, that may have additional constraints and attributes, that is to say, it extends the vocabulary of the UML. A constraint extends the semantics of a UML element. A constraint is used to specify conditions and propositions that must be fulfilled. The constraint may be written in natural language or in a particular constraint language such as OCL (Object Constraint Language). The third extension mechanism is the tagged value. A tagged value extends the properties of a UML element. This allows properties such as attributes in the metamodel and both predefined and user-defined tagged values to be attached to a model element.

Although this paper can be read in a self-contained way, it is based on the previous formalization of the UML metamodel elements, relationships and diagrams [9, 10]. This research aims at obtaining a whole proposal to support the UML extension mechanisms by means of an algebraic specification, which is in accordance with the four-layer metamodeling architecture on which the UML definition is based [17]. Therefore, the semantic framework provided in this paper faces up to the unpredictable and changeable nature of the UML, without limiting its ability to adapt the UML to the needs of a concrete domain.

In this paper, we focus on one of the UML diagrams, the Class Diagram (probably the kind of UML diagrams most widely used among software practitioners) to illustrate the approach, but this is applicable to any other UML diagram or model. We consider the UML Class

[1]The Object Management Group (OMG) was founded in 1989 by computer companies and was formed to create a component-based software marketplace by introducing of standardised object software.

Diagram within the whole UML notation, in a homogeneous and modular way, which makes proving UML inter-model properties possible, in addition to those properties related to just one model.

The rest of the paper is organized as follows: section 2 discusses some issues that provide the motivation for our framework . Section 3 introduces the main properties of Maude and shows examples that are used to present our proposal in the following section, section 4, which describes a procedure to support the extensibility mechanisms by means of both equational and rewriting logic computation. Finally, section 5 presents some concluding remarks, lessons learned and an outline of the work to be done in the future.

2. MOTIVATION

The UML notation has received a great deal of criticism since its appearance (some of which have been recognized by its authors), due to the absence of a formal definition. Many problems (ambiguity, inconsistency and incompleteness) have been identified in its semantics [1, 7, 24]. The UML static semantics is described by a semi-formal constraint language, the object constraint language (OCL) and the UML dynamic semantics is expressed in informal English. The formalization of a graphic notation can help to identify and remove these problems and also allows us to rigorously verify the system models constructed by that notation. Thus, the important goal is to combine the intuitive appeal of visual notations with the precision of formal specification languages.

Concerning extensibility, each extension mechanism implementation could show different semantics, resulting in different or ambiguous interpretations. In this matter, formal methods (FM) can also play an important role. The formalization of these mechanisms provides a unique interpretation and helps to identify and remove any ambiguity. This problem was formerly investigated by the authors, putting forward a set of extension mechanisms in this case for the object-oriented specification language OASIS [23] for which an OBJ3 formalization was proposed [25].

Formal languages based on logic, such as Z or VDM, permit us to suitably describe the structure of a system. On the other hand, formal languages based on process algebras, such as CSP or LOTOS, allow us to appropriately represent behavior of a system. However, these languages do not currently include the ability to directly handle extensibility mechanisms or specify metaprogramming applications necessary to formalize the extension of the UML metamodel.

The formal language chosen to implement our proposal is Maude[2] , a mathematically well-founded language, which is based on equational [14] and rewriting [21] logic and in addition is executable. Maude is an extension of OBJ3 with a far greater performance. Maude reaches an agreement with respect to the structure and the behavior of a system and, unlike other formal specification languages, offers an excellent framework to cope with changes in the UML metamodel at modeling time. The rewriting logic reflective framework can be exploited to support the UML metamodel evolution in a natural way.
Recently, the formalization of UML has been drawing increasingly considerable attention from many researchers. The effort of research groups has evolved from traditional graphical notations towards the UML notation. In this sense, one of the most relevant research groups is the pUML[3] group. Research directly related to this paper, but using the formal language Z, has been reported by this group [8, 11]. The primary goal of the pUML group is to define precise semantics for the UML notation and develop mechanisms that allow developers to rigorously analyze the UML models. One of the key differences between our work and the work on formalizing UML in Z by this group is, besides the reflection feature, that Maude specifications are directly executable, thus providing rapid prototyping.

Although there is some research effort directed towards formalizing particular extension mechanisms [2], none of them have addressed the extensibility of the UML within a global framework, as this paper does. The most innovative aspects in the UML formalizing approach presented in this paper are, in our opinion, firstly the fact of formalizing the UML metamodel evolution, and secondly its inclusion into the same formal framework as the other three UML architecture layers, thus providing a unique modular algebraic specification. Moreover, inasmuch as the UML is expected to undergo substantial changes in its semantics in the forthcoming years (three versions have emerged in the last four years), extensibility may play a pivotal role in this endeavor. Our formalizing process is holistic (the whole formal model provides integrated support for most of the UML diagrams) and seamless (you can navigate from the dynamic semantic aspects at the UML object layer towards the model, metamodel and metamodel evolution ones, that is to say, along the UML four-layer metamodeling architecture). The holistic approach is pursued

[2]Maude interpreter has been available since January 1999 and it is a more powerful language than OBJ3. Currently, Maude is freeware and runs under Linux.
[3]The pUML group is made up of international researchers and practitioners who are interested in providing a precise and well-defined semantics for UML.

for unification purposes, that is to say, this integrated formal framework permits the user to detect inconsistencies among different views modeled by a UML user. For example, a UML collaboration diagram is equivalent to a UML sequence diagram, except that the former focuses on the relationship among objects, while the latter emphasizes the time sequence, both in an interaction. The Maude executable specification generated can also be considered as a formal UML virtual machine, therefore providing the possibility for a developer to manipulate and animate the UML models.

3. DEALING WITH RIGOR AND METAPROGRAMMING ASPECTS IN MAUDE

In order to help us understand the next section, some basic notions regarding the formalism used are presented. Maude evolved from OBJ3, so we will briefly describe some of the differences between the underlying formalism in OBJ3 and Maude. The Maude features will be illustrated with some of the definitions and the algebraic specifications used later in Section 4. A detailed description of order-sorted equational logic [14] and rewriting logic [21] can also be found in any book related to the topic (e.g. [13]).

Extensibility by reflection (the ability of a logic to be interpreted in itself) is exploited in Maude so that the basic functionalities of the language Core Maude, are extended by reflection to Full Maude. Next, both aspects of the language will be presented.

3.1. BASIC ASPECTS: CORE MAUDE

Maude [4] is a formal specification language that supports both equational and rewriting logic computation. Maude is an extension of OBJ3 which is based on equational logic and provides parameterized programming, multiple inheritance, and a *large-grain* programming technique, to support the scalability of the specification and appropriately manage the complexity of a system. Maude's equational logic called *membership equational logic* extends OBJ3's equational logic by supporting *membership axioms*, a generalization of sort constraints in which a term is asserted to have a certain sort if a condition is satisfied. Maude's functional modules are theories in membership equational logic. For example, figure 1 shows a functional module, where the abstract syntax of a type in UML is described. A type comprises a name, a list of attributes and a list of operations. The sort *Type*, the constructor *type* and the query operation *typeAttribute* are declared in the functional module *TYPE* (*fmod*

stands for *functional module*, in Maude). This last operation takes one argument, an element of sort Type, and yields its list of attributes. The sorts *TypeName*, *AttributeList* and *OperationList* are declared, respectively, in the modules *TypeName*, *AttributeList* and *OperationList*, which are imported by protecting declarations.

```
(fmod TYPE is sort Type .
  protecting TYPENAME . *** Importing the module TYPENAME,
  protecting ATTRIBUTELIST . *** the module ATTRIBUTELIST
  protecting OPERATIONLIST . *** and the module OPERATIONLIST
  op type : TypeName AttributeList OperationList -> Type
  op typeAttribute : Type -> AttributeList .
  op typeName : Type -> TypeName .
  var TN : TypeName . var AL : AttributeList .
  var OL : OperationList .
  eq typeAttribute (type (TN, AL, OL)) = AL .
  eq typeName (type (TN, AL, OL)) = TN .
 endfm)
```

Figure 1 Algebraic specification of a UML type

The second kind of module in Maude, system module, is based on rewriting logic. For the sake of space, this one is not shown in this paper. Core Maude also supports hierarchies of system modules and unparameterized functional modules by using the importation of modules. The functional module *META-LEVEL* implements the reflection. This property of rewriting logic allows us to represent any finitely presented rewrite theory R by means of a particular rewriting theory, the *universal theory* U. Rewriting logic is also a good logical framework in which other logics can be represented [20], thus endowing Maude with high expressive power.

3.2. ADVANCED ASPECTS: FULL MAUDE

The basic functionality of the language, *Core Maude*, is extended by reflection to Full Maude. Full Maude, which is written in Core Maude, is a metaprogramming application that includes object oriented modules (system modules with some syntactic sugar so as to allow us to specify concurrent object oriented systems), parameterized modules, views and module expressions in the OBJ style.

As an example, an extract of an object-oriented module called *PERSON* is shown in figure 2. The declaration of a class named *Person* with two attributes called *name* of sort *Qid* (quoted identifier) and *age* of sort *MachineInt* (machine integer) is introduced by the key word *class*. The message declarations are introduced by the keyword *msg* (message

changingAge) or *msgs* if multiple messages with the same arity are defined. The labeled rewrite rules, which are introduced by the keyword *rl* (rule), specify in a declarative way the behavior associated with the messages. For instance, the rule labeled *changingAge* specifies the behavior of person objects, which may receive messages to change its age. The *rew* (rewrite) command is used to rewrite a term representing a configuration of a concurrent object system: when a person called *John* with object identifier (OI) 1 receives a message *changingAge (1, 15)*, value 15 becomes its attribute *age*. The subsort relationship *MachineInt* < *Oid* is introduced, indicating that machine integers can be viewed as object identifiers. All object-oriented modules implicitly include the *CONFIGURATION* predefined functional module which defines the basic concepts of concurrent object systems. Some of these definitions are left unspecified such as the sort *Oid*, completed and used in the module *Person*. Other examples can be found in Maude's documentation [4].

```
(omod PERSON is
 protecting QID .
 protecting MACHINE-INT .
 subsort MachineInt < Oid .
 class Person | name : Qid, age : MachineInt .
 msg changingAge : Oid MachineInt -> Msg .
 var OI : Oid .  var QI : Qid .  vars MI, MI2 : MachineInt .
 rl [changingAge] : changingAge (OI, MI2) < OI : Person |
 name : QI, age : MI > => < OI : Person  | name : QI, age : MI2 > .
endom)
(rew < 1 : Person | name : 'John, age : 14 >
     < 2 : Person | name : 'Peter, age : 24 > changingAge (1, 15) .)
Result Configuration : < 1 : Person | age : 15 , name : 'John >
                       < 2 : Person | age : 24 , name : 'Peter >
```

Figure 2 Specification in Full Maude of an object-oriented module named PERSON

As seen above, Maude like other formal languages is helpful in specifying both the structure and the behavior of a system. However, the outstanding and distinguishing feature of Maude, on which our proposal is based, is reflection. This characteristic has been efficiently implemented in Maude through its *META-LEVEL* module, and has a great practical interest in Software Engineering, thus providing the possibility of building both specifications of concrete models and CASE tools supporting notation evolution, with rigor and homogeneity based on a solid scientific foundation.

3.3. TAKING ADVANTAGE OF THE REFLECTIVE PROPERTIES

Rewriting logic is reflective [5]. This means that there is a rewrite theory called *universal theory U* that can represent any rewrite theory R (including U itself) as a term $\overline{R}$, any terms t, t1 in R as terms $\overline{t}$ and $\overline{t1}$, and any pair (R, t) as a term ($\overline{R}$, $\overline{t}$), in such a way that the following equivalence is satisfied:

$$R \vdash t \rightarrow t1 \Leftrightarrow U \vdash (\overline{R}, \overline{t}) \rightarrow (\overline{R}, \overline{t1})$$

Subsequently, an extract of the signature of U is presented in which terms and functional modules are reified as elements of the data types *Term* and *FModule*. Figure 3 shows the operators used to represent a term of any sort and a functional module.

```
fmod META-LEVEL is sort FModule .
subsort Qid < Term .*** to represent variables
subsort Term < TermList .
op { _ } _ : Qid Qid -> Term . *** constants by pairs [constant, sort]
op _ [ _ ] : Qid TermList -> Term . *** operators (operator, subterms)
op _,_ : TermList TermList -> TermList [assoc] . *** list of terms
op error* : -> Term .
op fmod_is_ _ _ _ _ _ _endfm : Qid ImportList SortDecl
SubsortDeclSet OpDeclSet VarDeclSet MembAxSet EquationSet -> FModule .
```

Figure 3 Operators of the universal theory U to represent terms

The declaration of subsorting *Qid < Term* is used for representing variables as quoted identifiers. The first operator, { _ } _, is used to represent constants by pairs (constant and sort of the constant), both in quoted form. The next operator, _ [_], denotes the recursive construction of term out of subterms. The first argument represents the top operator in quoted form, and the second argument represents the list of subterms. Finally, the operator term concatenation is declared. On the other hand, a functional module is metarepresented by means of eight elements: a name (sort Qid), a list of imported modules (sort ImportList), a sort declaration (sort SortDecl), a set of subsort declarations (sort SubsortDeclSet), a set of operator declarations (sort OpDeclSet), a set of variable declarations (sort VarDeclSet), a set of membership axioms (MembAxSet), and a set of equations (EquationSet). The description of these sorts is omitted here, but it can be found in Maude's documentation [4].

In order to illustrate the meta-representation of a term, the module *TYPE* defined in figure 1 is used. A term of sort *TypeName* and its meta-representation (figure 4) are shown as an example.

tn = typeName (type ('Person, attribute ('Male, 'Bool) attribute ('Age, 'Integer,), operation ('Income, parameter ('date, 'Date), 'Integer)))
$\overline{tn}$= 'typeName ['type [{ "Person } 'Qid , '__ [
'attribute [{ "Male } 'Qid , { "Bool } 'Qid] ,
'attribute [{ "Age } 'Qid , {"Integer } 'Qid]] ,
'operation [{ "Income } 'Qid, 'parameter [{ "date } 'Qid, { "Date } 'Qid], { "Integer } 'Qid]]]]

Figure 4 Term of sort TypeName and metaterm of sort Term

Maude draws efficiently on *reflection* which can be exploited in meta-programming applications such as the present work proposes. In such applications rewrites and reductions at both high level (universal theory U) and low level (rewrite theory R) may be necessary. Although all rewrites and reductions may be done at high level, we should be aware that a single step of rewriting at low level involves many rewriting steps at high level. Notice that the terms and the modules including equations, operations and constant symbols at the low level are metarepresented at the high level (see figure 4). The reduction of a term as the one given in said figure (*tn*) at the high level ($\overline{tn}$), leads to describing new equations, operations and constant symbols in the universal theory U and a number of reductions at the high level.

In order to increase the efficiency of metalevel computations, descent functions such as *meta-reduce*, *meta-apply* and *meta-rewrite* are defined in a metalevel theory M, which is an extension of the universal theory U. As an example, a metareduction by using the function *meta-reduce* is shown:

```
Maude> (red meta-reduce (TYPE, up(TYPE, tn)) .)
Result Term : { "Person } 'Qid
```

where the function *meta-reduce* takes two arguments: the first argument is the meta-representation of the module *TYPE* obtained by means of the constant called *TYPE* declared in the importation of the module *META-LEVEL[TYPE]* (high level). The second argument is the meta-representation of a term *tn* of sort *TypeName* (figure 4) included in the module *TYPE* (low level). The meta-representation of the term *tn* in the module *TYPE* is obtained by means of the *up* command.

4. TOWARDS A HOLISTIC FORMALIZATION OF THE UML

Is it actually feasible to define the semantics for all of the UML notation and its architecture? According to the pUML group, the answer is unknown until semantics for all UML is constructed [19]. In addition, we must take into consideration the existing problems, referred to in section 2, in relation with the current version of the notation. In spite of this situation, many efforts are being made in the direction of offering formal models for that, as mentioned in said section.

We know that trying to formalize a (complex) notation in these circumstances is not an easy task, but, in particular, the formal framework that we report in this paper, can cope with the current semantics of UML, and it also serves as a basis for its future versions. This is due to the fact that this framework includes the metametamodel of UML; but there is a limitation: we assume that the UML four-layer metamodeling architecture (explained below) is conserved. Only in the case that its architecture changes, the formal framework should change too.

Subsequently, three approaches to formalize a graphical notation like UML are presented. One of these approaches is chosen and tailored to the UML four-layer metamodeling architecture. Finally, the formal specification supporting our selection is carried out.

4.1. CHOOSING A FEASIBLE FORMALIZING APPROACH

Extension mechanisms provide additional information included in the UML metamodel, that is, metainformation. If formalizing a graphical notation is intended, representing both the metainformation and its evolution should be taken into account. According to the metainformation that can be represented, three approaches are distinguished:

1 Modeling strategy. The formalization is carried out by means of a translational strategy, that is to say, by translating each UML modeling element to some formalism element.

2 Metamodeling strategy. The formalism is used to specify the language metamodel.

3 Meta-metamodeling strategy. The formalism is used to specify the language metamodel and its evolution.

In the first approach a correspondence between each modeling element in the graphical notation and some particular expressiveness, feature or construct in the formal language is established [3, 12]. Each

UML diagram corresponds to a formal specification in the formalism. For example, if an algebraic language such as Larch, OBJ3 or Maude is used, an attribute or an operation of a class can be denoted as an operation symbol in the formal language. Therefore, an algebraic specification corresponds to a domain model (figure 2 shows an example of this approach).

In this approach, a change in the model leads to a change in the formal specification, following a manual or automatic procedure based on the correspondence previously obtained. If a new modeling element such as a stereotype is included in the graphical notation, some formal language mechanisms are used to describe both its syntax and semantics, so enriching the formal-informal correspondence. In order to obtain this goal, analysts must search for suitable features in the formal language that best fit the new element and then modify the formal specification [2]. Therefore, the evolution of the metamodel is not formally supported in the initial specification.

In the second approach, the language metamodel is represented by a formal specification [8, 9, 15]. In an algebraic framework such as Maude, the graphical notation syntax is described by what we call the *syntactic specifications*[4], incorporating the static semantics by means of equations or axioms in the syntactic specifications (figure 1). The graphical notation dynamic semantics (for example, specifications representing objects) are expressed by *semantic specifications*[5] which use the syntactic specifications previously obtained so as to maintain the consistency in the system state. In a similar way, the same role is played by the concepts of descriptor and instance element used by the pUML group [19].

As an example, the metamodeling strategy applied to the UML Class Diagrams is shown in figure 5. A particular problem model is initially depicted by a UML class diagram from the functional requirements previously elicited. Then, this model is transformed and represented in an alternative and equivalent way by means of a formal term of the quotient term algebra of the syntactic theory signatures, that is to say, the interpretation of the signature. In turn, the instance of a model corresponds to a formal term of the quotient term algebra of the semantics theory signatures that can also be represented by a term or by the elements of the problem. *Term1... Termn, Termm* are formal terms of the (syntactic or semantic theories) corresponding term algebras or formal interpreta-

[4] A syntactic theory is a module that represents the syntax and static semantics of the UML model elements.

[5] A semantic theory is a module that represents the dynamic semantics of the UML model elements

tion models; *ErrorMessage* and *canonicalTerm1... canonicalTermm* are formal terms of the quotient term algebra, which is the minimal formal interpretation model. For instance, in the top part of figure 5, a number of different incorrect UML class diagrams are firstly translated to the same number of, corresponding, formal terms of the term algebra. Then, as all of them are erroneous, they are converted (or reduced, in algebraic terminology) to a unique, minimal term with the same semantics, in this case the term *ErrorMessage* from the quotient term algebra. In the same way, as seen in the bottom part of figure 5, objects and relationships among objects can also be dealt with.

However, in the second approach, if a new graphical notation modeling element, such as a stereotype, is included in the UML, as seen in the previous approach, the analysts must search for some formal language mechanisms to express its syntax and semantics and then modify the initial formal specification. Therefore, the evolution of the UML metamodel is not formally supported either.

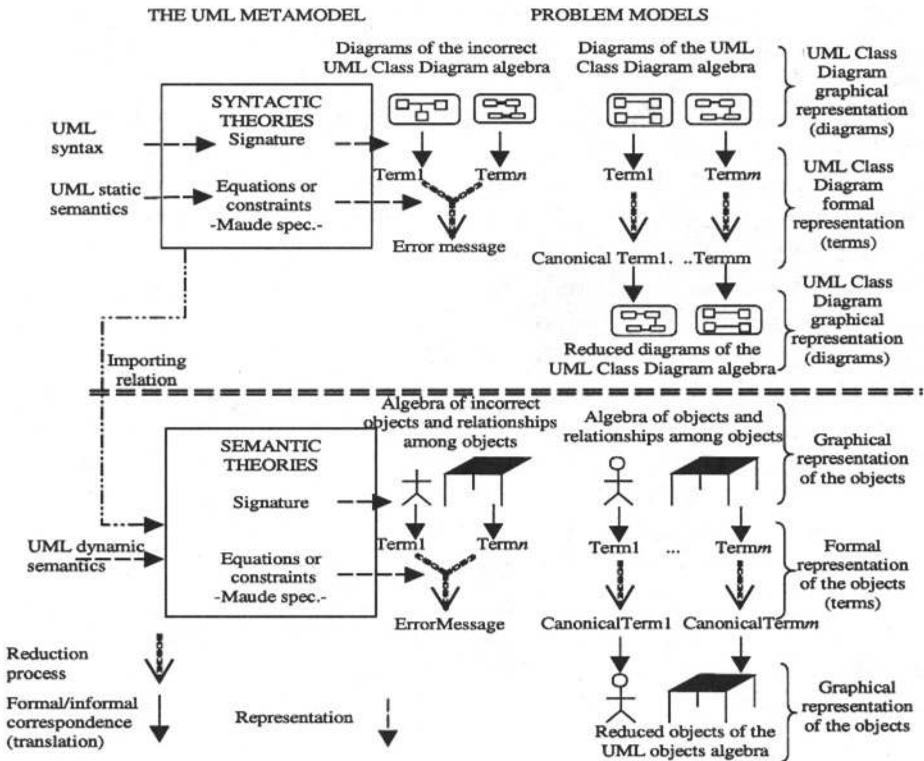

Figure 5 Formalizing the UML metamodel

The third approach deals with the issue of evolution in its two dimensions (modeling and metamodeling). Two levels are proposed to formalize a modeling language. The lower level formally specifies the graphical notation syntax and semantics, that is to say, its metamodel. This level supports the evolution of the model, in such a way that a class, an object or whatever modeling element may be included in the system represented by a term (in the same way as the second approach). The higher level formally supports the evolution of the language metamodel and enables us to establish properties that must be satisfied by this metamodel. In this level the modules defined to represent the language metamodel are reified as formal terms in such a way that elements such as a class, a relationship or a constraint may be included in the metamodel, therefore extending the notation. This is the approach chosen in our research.

4.2. SEAMLESS FORMALIZING THE UML FOUR-LAYER METAMODEL ARCHITECTURE

UML currently provides lightweight extensibility by means of stereotypes, constraints and tagged values. Therefore, a formal language endowed with enough expressiveness to specify new modeling mechanisms at specifying time is required. The key idea in our approach focuses on using reflection to represent the evolution of the metamodel. Although any formal language endowed with reflection could be used, the Maude language shall be employed for the reasons mentioned in previous sections. Table 1 shows the correspondence between the UML conceptual framework and the formal framework that we propose.

Table 1 Representing in Maude the four-layer metamodeling architecture of UML

The UML layer	*Maude formal layer*	*Example*
Meta-metamodel	The module META-LEVEL	MetaClass, MetaOperation
Metamodel	Syntactic and semantic specifications	Class, Operation
Model	Terms (of syntactic specifications)	Person, Name, Age
User objects	Terms (of semantic specifications)	<John, 14>, <Joy, 34>

The UML user objects layer allows a particular domain to be described. This layer lies on terms of the semantic specifications. For

instance, specific objects such as John and his age, 14, can be represented in this layer.

The UML model layer permits diagrams to be defined in terms of the concepts represented in the UML metamodel. Problem domain concepts such as *person*, *name* and *age* can be formally represented by terms of syntactic specifications.

The UML metamodel layer aims at obtaining the conceptual framework to specify domain models and its evolution, that is to say, the syntactic and semantic rules on how to construct correct UML class diagrams belong to the metamodel layer (as an example you can see figure 1). On the other hand, the application of these UML class diagrams to describe concrete systems (for instance the subterm representing a type name of a domain model shown in figure 4) belongs to the model layer. Transformations of diagrams can be supported by means of equations. Thus, semantically equivalent diagrams, which are simpler than the original ones, can be obtained. This layer is represented by the syntactic and semantic specifications. Concepts such as *class*, *attribute*, *association*, *operation* are formalized in the syntactic specifications and *link*, *object* and *value* are formalized in the semantic specifications. The UML metamodel also contains rules, constraints and model usage aspects which can be expressed in Maude.

Finally, the UML metametamodeling layer (the UML highest abstraction level) presents the language for defining the UML metamodel and its evolution. An extension of the module *META-LEVEL* covers the metametamodeling layer. This allows the UML to be customized and adapted to the analyst's modeling requirements or to the other methodologies development process [6] and to develop UML evolving CASE tools. The formalization of this layer also enables us to establish properties that must be satisfied by the UML metamodel. Therefore, the formalizing approach presented here complies with the UML meta-metamodel layer framework

4.3. FORMALIZATION USED

As a first step towards applying the metamodel extension process, the formalization of a subset of UML will be shown. The abstract syntax of an attribute in UML is described. An attribute is represented by means of a name and a type. The sort *Attribute*, the constructor *attribute* and the query operations *typeName* and *attributeName* are defined in the functional module ATTRIBUTE (figure 6, belonging to the syntactic specification layer in table 1). These operations take one argument, an attribute, and yield its name and its type, respectively.

```
(fmod ATTRIBUTE is sort Attribute .
  protecting ATTRIBUTENAME .
  protecting TYPENAME .
  op attribute : AttributeName TypeName -> Attribute .
  op typeName : Attribute -> TypeName .
  op attributeName: Attribute -> AttributeName .
  var NA : AttributeName .    var NT : TypeName .
  eq typeName (attribute (NA, NT)) = NT .
  eq attributeName (attribute (NA, NT)) = NA .
endfm)
```

Figure 6 Algebraic specification of a UML attribute

The next step is to define an extension of the module *META-LEVEL* to deal with the meta-metamodel UML layer (see table 1). The operations *addSorts*, *addOperation*, *replaceOperation*, *addVariable*, *addEquation* and *removeEquation* are defined to allow the modification of functional modules (figure 7).

A functional module is metarepresented by terms of sort *FModule* (figure 3), included in the predefined *META-LEVEL* module. With the module declaration *protecting META-LEVEL[ATTRIBUTE]*, the constant *ATTRIBUTE* of sort *Module* (supersort of Fmodule to metarepresent system modules) is declared, and a new equation making the constant *ATTRIBUTE* equal to the metalevel representation of the user-defined module with name *ATTRIBUTE* previously declared is included (figure 6). This constant will be used to extend the meta-representation of the module *ATTRIBUTE.*

Note that in this section we are focusing on the technical aspects of the underlying formal models. The practical use of this research assumes that a software tool is available, to hide these ugly equations, operations and constants symbols from their users. A prototype of this tool is already working [10], which takes UML diagrams edited by Rational Rose (in XMI format) and produces their equivalent formal representation, and an accompanying process model has also been defined [22].

4.4. EXAMPLE OF CHANGEABILITY EXTENSION

In order to gain insight into our approach, an example illustrating how to extend the UML Class Diagrams is presented. In particular, a new attribute, named changeability, is included in the UML metamodel metaclass *Attribute.* This property specifies whether the value of an attribute may be modified after the object has been created. With this

```
(fmod META-LEVEL-EXTENSION is protecting META-LEVEL[ATTRIBUTE] .
*** Add a set of sorts
  op addSorts : FModule QidSet -> FModule .
*** Add an operation
  op addOperation : FModule Qid QidList Qid AttrSet -> FModule .
*** Replace an operation of a module identified by its name (second argument)
  op replaceOperation : FModule Qid Qid QidList Qid AttrSet -> FModule .
*** Replace an operation from a set of operation declarations
  op repOper : OpDeclSet Qid Qid QidList Qid AttrSet -> OpDeclSet .
*** Add a variable declaration
  op addVariable : FModule Qid Qid -> FModule .
*** Add an equation.
  op addEquation : FModule Term Term -> FModule .
*** Remove an equation
  op removeEquation : FModule Equation -> FModule .
*** Remove an equation from a set of equations
  op remEquation : EquationSet Equation -> EquationSet .
  var QS : QidSet .   var QS1 : QidSet .   var QI : Qid .   var QI1 : Qid .
  var QI2 : Qid .   var QI3 : Qid .   var QI4 : Qid .   var OpName : Qid .
  var IL : ImportList .   var SD : SortDecl .   var SSDS : SubsortDeclSet .
  var ODS : OpDeclSet .   var VDS : VarDeclSet .   var MAS : MembAxSet .
  var EqS : EquationSet .   var EQ : Equation .   var EQ1 : Equation .
  var TE1 : Term .   var TE2 : Term   var AS : AttrSet .   var AS1 : AttrSet .
  var QL : QidList .   var QL1 : QidList
  eq addSorts (fmod QI is IL sorts (QS) . SSDS ODS VDS MAS EqS endfm, QS1)
     = fmod QI is IL sorts(QS ; QS1) . SSDS ODS VDS MAS EqS endfm .
  eq addOperation (fmod QI is IL SD SSDS ODS VDS MAS EqS endfm,
              QI1, QL, QI2, AS)
   = fmod QI is IL SD SSDS ODS  op QI1 : QL -> QI2 [AS] .VDS MAS EqS endfm .
  eq replaceOperation (fmod QI is IL SD SSDS ODS VDS MAS  EqS endfm,
                      OpName, QI1, QL, QI2, AS) = fmod QI is IL SD SSDS
    repOper (ODS ,OpName, QI1, QL, QI2, AS) VDS MAS EqS endfm .
  eq repOper (op QI3 : QL1 -> QI4 [AS1] . ODS, OpName, QI1, QL, QI2, AS)
   = if QI3 == OpName then op QI1 : QL -> QI2 [AS] . ODS
  else op QI3 : QL1 -> QI4 [AS1] . repOper (ODS, OpName, QI1, QL, QI2, AS) fi .
  eq addVariable (fmod QI is IL SD SSDS ODS VDS MAS EqS endfm, QI1, QI2)
     = fmod QI is IL SD SSDS ODS VDS var QI1 : QI2 . MAS EqS endfm .
  eq addEquation (fmod QI is IL SD SSDS ODS VDS MAS EqS endfm, TE1, TE2) =
          fmod QI is IL SD SSDS ODS VDS MAS eq TE1 = TE2 . EqS endfm .
  eq removeEquation (fmod QI is IL SD SSDS ODS VDS MAS EqS endfm, EQ)
     = fmod QI is IL SD SSDS ODS VDS MAS remEquation (EqS, EQ) endfm .
    eq remEquation (EQ EqS, EQ1) = if EQ == EQ1 then EqS
                            else EQ remEquation (EqS, EQ1) fi .
endfm)
```

Figure 7 Extension of the module META-LEVEL

aim in mind, the module *ATTRIBUTE* is modified (figure 8) by means of the operations defined above.

```
moduleAttribute = addEquation (addEquation ( removeEquation ( removeEquation (
 addVariable (replaceOperation (addSorts (addOperation ( addOperation
  (ATTRIBUTE, 'changeable, nil, 'Changeability, none)
 ,'frozen, nil, 'Changeability, none) ,'Changeability )
 ,'attribute, 'attribute ,'AttributeName 'TypeName 'Changeability, 'Attribute, none)
 ,'A, 'Changeability) ,eq 'typeName [ 'attribute [ 'NA , 'NT ] ] = 'NT .)
 ,eq 'attributeName [ 'attribute [ 'NA , 'NT ] ] = 'NA .)
 ,'typeName [ 'attribute [ 'NA , 'NT , 'A ] ],'NT)
 ,'attributeName [ 'attribute [ 'NA , 'NT , 'A ] ], 'NA) .
```

Figure 8 Modification of the module ATTRIBUTE

A new sort, *Changeability*, representing the new attribute of the metaclass *Attribute* is included in the sort declaration. The domain of the operation *attribute* is modified with a new sort, *Changeability*, what leads to the modification of the operations *typeName* and *attributeName* equations. Now, we introduce the range of values for *Changeability* that are represented, respectively, by the constant symbols *frozen* (the attribute value may not be altered after the object is instantiated and its values initialized) and *changeable* (no restriction on modification is imposed). These changes are carried out by means of the reduction of the term *moduleAttribute* which yields the meta-representation of the extension of the module *ATTRIBUTE*. The resultant module is shown in figure 9.

```
(fmod ATTRIBUTE is sorts Attribute Changeability .
 protecting ATTRIBUTENAME .
 protecting TYPENAME .
 op attribute : AttributeName TypeName Changeability -> Attribute .
 op frozen : -> Changeability .
 op changeable : -> Changeability .
 op typeName : Attribute -> TypeName .
 op attributeName: Attribute -> AttributeName .
 var NA : AttributeName .  var NT : TypeName .  var A : Changeability .
 eq typeName (attribute (NA, NT, A)) = NT .
 eq attributeName (attribute (NA, NT, A)) = NA .
endfm)
```

Figure 9 The module ATTRIBUTE (obtained from the reduction of the term moduleAttribute in figure 8)

The module *SYSTEMOBJECTS*, which belongs to the semantic specifications in table 1, specifies the population of objects existing in the system, and includes, among others, operations to add and remove objects, and to modify the values of the object attributes. Therefore, the

```
*** constant representing: modifying an attribute labeled as frozen is not permitted
  op errorFrozenAttribute : -> ObjectList .
*** modify the value of an attribute
  op changeAttributeValue : TypeList ObjectList Oid AttributeName
                            Value -> ObjectList
*** modify the value (Value) associated to an attribute (AttributeName) of an object
*** (Oid) from a list of objects (ObjectList)
  op changeObjectList : AttributeList ObjectList Oid AttributeName
                        Value -> ObjectList .
*** modify the value (Value) of an attribute (AttributeName) from the list of values
*** (ValueList) associated to the attributes of an object (AttributeList)
  op changeValueList : AttributeList AttributeName ValueList Value -> ValueList .
*** check if an attribute from a list of attributes is changeable
  op isAttribChangeable : AttributeList AttributeName -> Bool .
  var OI : Oid .  var OI1 : Oid .  var TN : TypeName .  var NEVL : NEValueList .
  var VL : ValueList .  var TL : TypeList .  var AN : AttributeName .
  var AN1 : AttributeName .  var V : Value .  var V1 : Value .
  var OL : ObjectList .  var ATL : AttributeList .  var C : Changeability .
  eq changeAttributeValue (TL, OL, OI, AN, V) =
 if isAttribChangeable (typeAttribRName (objectTypeNameROid (OI, OL), TL), AN)
  then changeObjectList (typeAttribRName (objectTypeNameROid (OI, OL), TL),
                                                      OL, OI, AN, V)
  else errorFrozenAttribute fi .
  ceq changeObjectList (ATL, object ( OI , TN , NEVL) OL, OI1, AN, V) =
   object (OI, TN, changeValueList (ATL, AN, NEVL, V)) OL  if  OI == OI1 .
  ceq changeObjectList (ATL, object ( OI, TN, NEVL) OL, OI1, AN, V) =
   object (OI, TN, NEVL) changeObjectList (ATL, OL, OI1, AN, V) if OI =/= OI1 .
  ceq changeValueList (attribute (AN1, TN, C) ATL, AN, V VL, V1) =
      V1 VL if AN1 == AN .
  ceq changeValueList (attribute (AN1, TN, C) ATL, AN, V VL, V1) =
      V changeValueList (ATL, AN, VL, V1)  if  AN1 =/= AN .
  ceq isAttribChangeable (attribute (AN1, TN, C) ATL, AN) = true
      if AN1 == AN and C == changeable .
  ceq isAttribChangeable (attribute (AN1, TN, C) ATL, AN) = false
      if AN1 == AN and C == frozen .
  ceq isAttribChangeable (attribute (AN1, TN, C) ATL, AN) =
      isAttribChangeable (ATL, AN) if AN1 =/= AN .
```

Figure 10 Definition of errorFrozenAttribute and changeAttributeValue operations

dynamic semantics of the new feature *Changeability* should be described in the module *SYSTEMOBJECTS*. To attain this goal, the operation *changeAttributeValue* and the constant operation *errorFrozenAttribute* are declared in the module *SYSTEMOBJECTS* (figure 10). Likewise, new equations are included to prevent the modification of an attribute value labeled as frozen. The equations declared for the query operations *typeAttribRName* and *objectTypeNameROid* are not shown. The operation *typeAttribRName* yields the attributes of a type by using a type name, and the operation *objectTypeNameROid* yields the type name of an object from an object identifier. For the sake of space, the modification of the module *SYSTEMOBJECTS* from the extension of the module *META-LEVEL* defined in figure 7 is not given. This new module is obtained in the same way as the modified module *ATTRIBUTE* has been (see figure 8).

4.5. VERIFICATION OF THE EXTENSION

To make the practical interest of metamodel extension clear, we will show how to detect the violation of a UML statement (the meaning of the label *frozen*) concerning the new property introduced, *changeability*. A metareduction by using the meta-representation of the terms *tl* and *ol* is shown in figure 11.

```
tl = type ( 'Company, attribute ('SA, 'Bool, frozen)
 attribute ('NumberEmployee, 'Integer, changeable),nonOperation)
 type ( 'Bank, attribute ('Internet, 'Bool, changeable), nonOperation )
  type ( 'Person, attribute ('Male, 'Bool, frozen) attribute ('Age, 'Integer,
changeable), operation ('Income, parameter ('date, 'Date),'Integer)).
ol = newObject (object ( 8, 'Person, (true 20)), tl,
  newObject (object ( 7, 'Person, (true 22)), tl,
  newObject (object ( 6, 'Bank, (false)), tl,
  newObject (object ( 5, 'Bank, (false)), tl,
  newObject (object ( 4, 'Company, (true 260)), tl,
  newObject (object ( 3, 'Person, (true 26)), tl,
  newObject (object ( 2, 'Company, (false 1000)), tl,
  newObject (object ( 1, 'Person, (false 25)), tl, nonObject)))))))) .
(red meta-reduce (moduleSystemObjects,
                changeAttributeValue(tl, ol, 2,' SA, true) ) .)
Result Term : { 'errorFrozenAttribute } 'ObjectList
```

Figure 11 Term representing the addition of a set of objects. Meta-reduction of a term

The term *tl* of sort *typeList* represents a list of types, *Company, Bank*, and *Person*. *TypeList* is a sort declared by importing the parameterized module *LIST* instantiated with the module *TYPE*. This module specifies

the abstract syntax and semantics of a type in UML (a type comprises a name, a list of attributes and a list of operations). For example, the type *Person* has two attributes, *male* and *age*, and one operation, *income.*

The term *ol* of sort *ObjectList* represents the inclusion of eight objects in the modeled system. For instance, the person represented by the object with object identifier 8 is male and his age is 20. The expression $changeAttributeValue(tl, ol, 2, 'SA, true)$ denotes the meta-representation of the term *changeAttributeValue (tl, ol, 2, 'SA, true).* The reduction of this term, which tries to change the value of the attribute SA belonging to the object with object identifier 2, violates the constraint imposed on the equation declared in figure 10, that is to say, an attribute labeled frozen (like the attribute SA) can not be modified. Therefore, the metaterm { *'errorFrozenAttribute* } *'ObjectList* is obtained as a result of the metareduction.

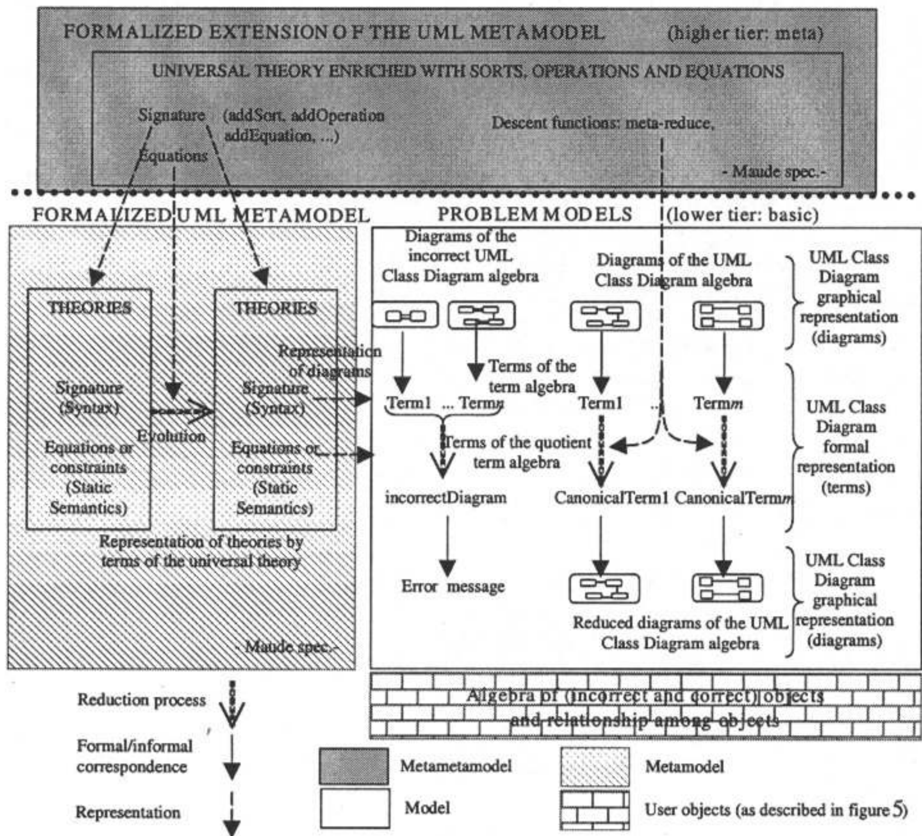

Figure 12 Seamless metametamodel formalization

To conclude this section, the metamodeling relationships for the UML Class Diagrams case, supported by functional modules, are graphically shown in figure 12. The metamodel evolution is specified by an extension of the module *META-LEVEL* that implements the universal theory U. The terms and the modules from the lower level (representing elements of the UML metamodel) are reified as Maude terms. Likewise, descent functions are used to efficiently compute reductions at the metamodel level. As a result, both extension of the metamodel and of the model are supported in a seamless fashion. In the same way, the lowest layer in the UML four-layer metamodeling architecture (the User Object UML layer) can also be integrated into the metametamodel formalization. In this case, the evolution theories are semantic specifications instead of syntactic specifications (see figure 5). For clarity, the internal relationships between the semantic specifications and the metametamodel level are not shown in detail in figure 12.

5. CONCLUSIONS AND FURTHER WORK

This paper reports research leading to a formal framework to support the UML notation extension mechanisms by means of an algebraic formalization. The reflective formal language Maude has been chosen to firstly represent both the graphical notation model and metamodel and secondly, the meta-metamodel by reifying the theories previously obtained. Formalizing the meta-metamodel simplifies the modeling of the UML extensibility.

Due to the size and complexity of the UML and taking into account the lack of a precise semantic definition, its formalization is a difficult task. For this reason, the existence of a formal framework, as the one presented in this paper, may be useful in adding new elements and improving the present definitions in new versions. The final formal models obtained rigorously support the unpredictable and changing nature of the UML, but the capacity of the notation to tailor the UML to needs for a specific application domain of interest is preserved.

Since most practitioners apply only a subset of elements of the modeling languages, the current trend in Software Engineering is to use simpler and easier languages and methods [18]. This research can also be tailored to CML (Core Modeling Language) which consists in a subset of the UML, including the extension mechanisms.

As Maude is executable, the final set of models can also be used as a UML virtual machine, at the specification level, which is a valuable characteristic for practitioners. Maude has lived up to the author's ex-

pectations regarding the advisability of using its reflection feature in metaprogramming applications.

This research has also shown that migrating from OBJ3 to Maude is quite easy. Former experience with OBJ3 greatly helped us to understand the concepts included in Maude and its application to Software Engineering problems. In particular, as Full Maude supports parameterised modules, views, and module expressions in the OBJ style, our previous research related to formalizing UML Statechart Diagrams and Class Diagrams can readily be translated into Maude in order to immediately apply the results of this paper.

The incorporation of a user interface that hides the formal aspects of the language and a defined model process, will result in a powerful tool to edit, validate, verify and execute functional system requirements. In this sense, we have proposed, in conjunction with other researchers, a process model based on a combination of rapid and evolutionary prototyping [22]. In addition we are now in the process of integrating the formal specifications with Rational Rose and other commercial modeling tools via a standard XMI interface. A Java prototype of this environment that combines Maude and Rose is already working. As a continuation of this research, we are currently working on extending the formalization to OCL. A preliminary version of this extended formal specification is already available. By using parameterized programming and reflection, we will integrate the specifications into both the model and metamodel layer to permit the user to specify constraints on a particular model and to extend the UML metamodel in a homogeneous way.

Our future research will address providing formal guidelines for the evolution of the metamodel. A wide range of possibilities have also opened up: exploring the proof of properties about the metamodel such as detecting deficiencies or design faults of the metamodel, and supporting rules that describe its appropriate evolution.

Acknowledgments

The authors gratefully acknowledge the useful comments and criticism of their colleagues in the MENHIR[6] project, especially Isidro Ramos. The authors also acknowledge José Meseguer (SRI International Computer Science Laboratory) and Roel J. Wieringa for their very helpful suggestions to the development of this and other related work. Thanks are also due to the anonymous referees for their constructive comments, and to Juan Piernas and Pilar González for their technical support.

[6]MENHIR is a research project made up of five subprojects, developed by five Spanish universities and granted by the CICYT (Science and Technology Joint Committee)

References

[1] Sinan Si Alhir. Extending the unified modeling language (UML). *http://home.earthlink.net/ salhir/index.html*, December 1999.

[2] J. Araújo, A. Moreira, and P. Sawyer. Specifying persistence, class views and excluding classes for UML. *Proc. 12th Intl. Conference on Software and Systems Applications*, December 1999.

[3] R. H. Bourdeau and B. H. C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Computer Science Laboratory SRI International*, January 1999.

[5] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. *In J. Meseguer, editor, proc. First Intl. Workshop on Rewriting Logic and its Applications*, 4 of Electronic Notes in Theoretical Computer Science. Elsevier, 1996.

[6] I. Díaz and A. Matteo. Objectory process stereotypes. *Journal of Object-Oriented Programming*, pages 29–38, June 1999.

[7] N. Dykman, M. Griss, and R. Kessler. Nine suggestions for improving UML extensibility. *Proc. UML'99 conference*, 1999.

[8] A. S. Evans and A.N.Clark. Foundations of the unified modeling language. *In 2nd Northern Formal Methods Workshop, Ilkley, electronic Workshops in Computing. Springer-Verlag*, 1998.

[9] J. L. Fernández and A. Toval. A formal syntax and static semantics of UML statecharts. *Technical Report, Department of Informatics: Languages and systems, University of Murcia (Spain)*, 1999.

[10] J. L. Fernández and A. Toval. Formally modeling and executing the UML class diagram. *Proc. V Workshop MENHIR 2000), Faculty of Informatics, University of Granada (Spain)*, March 2000.

[11] R. B. France. A problem-oriented analysis of basic UML static requirements modeling concepts. *Proc. OOPSLA'99*, pages 57–69, November 1999.

[12] R. B. France, J.M. Bruel, and M. M. Larrond-Petrie. An integrated object-oriented and formal modeling environment. *Journal of Object-Oriented Programming*, pages 25–34, November/December 1997.

[13] J. A. Goguen and G. Malcom. *Algebraic Semantics of Imperative Programs.* MIT Press, 1996.

[14] J. A. Goguen and J. Meseguer. Order-sorted algebra I. *Technical Report, SRI International, Stanford University*, 1988.

[15] B. A. Grima and A. Toval. An algebraic formalization of the objectcharts notation. *Proc. GULP-PRODE'94*, 1994.

[16] Object Management Group. UML notation guide version 1.3. *http://www.rational.com/uml/*, June 1999.

[17] Object Management Group. UML semantics version 1.3. *http://www.rational.com/uml/*, June 1999.

[18] Ari Jaaksi. A method for your first object-oriented project. *Journal of Object-Oriented Programming*, pages 17–25, January 1998.

[19] S. Kent, A. Evans, and B. Rumpe. Uml semantics FAQ. *in conjunction with ECOOP, http://www.cs.york.ac.uk/puml*, 1999.

[20] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. *In J. Meseguer, editor, proc. of First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of Electronic Notes in Theoretical Computer Science. Elsevier, 1996.

[21] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[22] B. Moros, J. Nicolás, J. G. Molina, and A. Toval. Combining formal specifications with design by contract. *Journal of Object-Oriented Programming*, 12(9):16–21, February 2000.

[23] O. Pastor, F. Hayes, and S. Bear. OASIS: An object-oriented specification language. *In proc. CAISE'92 Conference, Spriger-Verlag*, 1992.

[24] G. Reggio and R. J. Wieringa. Thirty one problems in the semantics of UML 1.3 dynamics. *Workshop Rigorous Modeling and Analysis of the UML Challenges and Limitations, OOPSLA'99*, November 1999.

[25] A. Toval, I. Ramos, and O. Pastor. Prototyping object oriented specifications in an algebraic environment. *Lecture Notes in Computer Science, Springer-Verlag*, 856:310–320, 1994.