On the Use of Algebras as Semantic Domain of Object Societies

I. Ramos, O. Pastor, J.H. Canós

Departament de Sistemes Informàtics i Computació. Universitat Politècnica de València

Camí de Vera, s/n , E-46071 València (Spain). e-mail {iramos,plo,joseh}@dsic.upv.es

July 15, 1992

Abstract

One of the main current research topics is to find a semantic domain for the Object-Oriented (OO) model. The interpretation of OO concepts has to reflect in a precise way the properties of the model, keeping at the same time good formal properties. One needs to be able in such a domain to cope with both structural and dynamic aspects. It is possible to use different sorts of logics (clausal, equational) to interpret the static aspects of objects; some extensions using temporal aspects (temporal logic, equational dynamic logic, etc.) have been used to interpret the dynamic ones, but in these cases we lose an efficient operational semantics allowing the *animation* of object societies. In software engineering, the need of such a semantics is crucial to give a meaning to the validation and verification of software via rapid prototyping.

In this paper, we use *term algebras* as the semantic domain for the object societies. Structural (*static*) aspects are thus handled using the traditional algebraic specification approach; to deal with dynamic aspects, a second-order extension of the equational logic is given. These second order aspects are managed in a first order style using syntactic machinery: operators and equational genericity. Doing things in this way we have a well defined and efficient operational semantics allowing the animation of OO-specifications.

The ideas presented in this paper are in an incipient state; an example following them is completely presented in this paper. A first implementation has been done using the functional language and interpreter *Axis* as a reification of term algebras. The operational semantics is conditional term rewriting.

1 Introduction

Many attempts have been done in the last years to find a semantic domain for the Object-Oriented (OO) model ([ES 91,Wie 91a]). The interpretation of concepts such as *object*, *class*, *inheritance*, etc., has to reflect in a precise way the properties of the model, and at the same time to keep good formal properties. In such a domain, one needs to be able to cope with both structural and dynamic aspects. It is possible to use different sorts of logics (clausal, equational) to interpret the static aspects of objects; some extensions using temporal aspects (temporal logic ([Gab 87]), equational dynamic logic ([Wie 91a]), etc.) have been used to interpret the dynamic ones, but in these cases we lose an efficient operational semantics allowing the animation of object societies. In software engineering, the need of such a semantics is crucial to give a meaning to the validation of software via rapid prototyping ([BCG 83]).

In this paper, we use *term algebras* as the semantic domain: a class will be interpreted as a term of a given sort, an object as a ground term, etc. At the specification level, the functional expressiveness will cope with the classical OO concepts such as encapsulation, reification, and others, and allows an easy way to take into account object interaction and inheritance (the well-known abstraction operations: generalization, specialization, aggregation, part-of, and others will be interpreted as operators over terms, and their semantics given in an axiomatic way). To deal with dynamic aspects, a second-order extension of the equational logic is given. These second order aspects are managed in a first order style using syntactic machinery: operators and equational genericity. We will focus on the aplication of these ideas to OASIS([PHB 92]), an open and active information systems specification language developed at the DSIC.

This paper is organized as follows. The next section makes a brief description of the OO model underlying the OASIS language, as well as the basic langage constructs. Section 3 introduces the use of term algebras to establish the semantics of the object societies, and section 4 explains how the reification of the OASIS object model is carried out. Appendices A to C show the complete source and object codes for the example discussed along the paper and a simplified algebraic specification of the OASIS language.

2 The OASIS framework

The OO paradigm is attracting a lot of attention in different fields of Computer Science. In Conceptual Modelling, object-orientation allows us to model an Information System (IS) and its environment in a uniform way using the object as the single design unit. In the OO model, concepts are close to real world phenomena; thus, the *semantic gap* (i.e., the difference between *what* the system is and *how* it is represented) is narrower than those of non OO approaches. Many languages have been developed for OO Conceptual Modelling([JHS 91,

Wie 91b]): they look at the world as a collection of interacting objects. OASIS follows these ideas.

An **object** will be for us an observable process; by that we mean that an object's evolution is a linear process, starting on the object's creation and finishing (if it happens) on its destruction. Its properties can be observed at any moment during its *life*. **Attributes** are *valued* properties and their values depend on time. There are also *universal* properties or object **laws** ([Wan 89]) that always must hold.

We call **class** a collection of objects sharing the same properties. In OO programming, the notion of type is closely related to the notion of class. In our view, a class consists of

- a type (in the sense of [ES 91]),
- a set of **object identifiers** (OIDs), and
- a mapping from the OID set to the population.

So, an **instance** of a class is an object of its type. An object encapsulate both structural and behavioural aspects. Let us take a brief view of them within the OASIS framework.

The basis for the formalization of structural aspects is the abstract data types (ADT) framework. ADTs give us a potentially infinite naming mechanism for objects in which equality can be tested via equational reasoning.

Attributes are valued object properties, whose type is an ADT. Their values at a given moment of the object's life define its **state**. Changes of the object's state are driven by occurrences of **events**. Every object has a life (which is a process execution) that we will represent as a *trace* or sequence of events. So, observations will consist of reading attribute values. Those attributes whose value does not change during the object's existence are called *constant* attributes; otherwise, they are named *variable* attributes. A constant attribute will be selected as the OID or **key**.

For each object, there are two special events: `new` creates the object, and must be the first event in the object's trace; `destroy` is the cause of the object's destruction. So, if we call E the set of events associated to an object O, any trace O will follow the pattern `new`$\bullet\ e_1 \bullet \cdots \bullet e_n \bullet$ `destroy`, being $e_i \in E,\ i = 1, \cdots, n$.

Obviously, we will use the notion of class as the basic specification construct. Structural mechanisms are used in order to build our conceptual model in a constructive way; a collection of operators gives us a set of relationships between classes. *Aggregation* induces a part-of relationship; classes may also be embedded in a *specialization* hierarchy; specialization usually implies reuse of specification code by allowing a class to inherit properties from its superclass. The *grouping* operator allows us to build complex classes whose instances are

made of a collection of instances of the grouped class. Last but not least, the *parallel composition* operator defines the whole object society as a composition of previously defined classes. So, class operators provide a constructive way to specify the whole Information System.

Objects behaviour can be modelled by means of Petri nets, process algebra, etc. In OASIS, arbitrary interleaving has been used as an implementation for parallelism; in this way, users are able to choose the next action to be taken in a menu-driven process.

There are two dynamic relationships between objects: **event sharing** and **triggering relationships**. A shared event will participate in the lives of the objects sharing it. In the other hand, conditions may serve as triggers of events from other objects, introducing activity in the system.

Appendix A shows the OASIS specification of the elementary class `person` in the context of the well known employment agency case study. A class specification starts describing the structure of the objects belonging to that class. We have constant and variable attributes, being the constant attribute `person-code` the key; abstract data types — in particular, the ADT *string* — give us a naming mechanism for objects, which is potentially infinite and which equality can be tested, so it is a good object identification mechanism.

The observation function allows us to specify the values of the attributes as a function of the objects' behaviour. The class `person` has two variable attributes, `is-cand` and `is-employee`. A person is an employee while working in a company:

```
is_employee(person,time):bool
        formulas
        is_employee(P,T)=false :- new_person(P,T).
        is_employee(P,T)=true :- hire(C,P,T).
        is_employee(P,T)=false :- fire(C,P,T).
        end_formulas
```

As we can see in the above example, the observation function is defined axiomatically using a logical expressiveness. The example corresponds to the clausal+equational version of the language (L-OASIS). We use a trace language to represent the objects lives, including a trace as the last argument of every event in the system being specified. The other variable attribute, `is-cand`, is defined in a similar way (see page 15).

The allowed set of traces is defined by means of preconditions. In our example, a person might apply only once prior to be hired, and only can be destroyed if he/she is not an employee:

```
preconditions
     apply(P,T) :- is_cand(P,T)=false,is_employee(P,T)=false.
     destroy_person(P,T) :- is_employee(P,T)=false.
```

The active behaviour is defined in the `triggering` paragraph. A person must apply after created:

```
triggering
   self :: apply(P,T) :- new_person(P,T).
```

Both preconditions and triggering sentences are phrases of a dynamic clausal+equational logic ([Wie 91a]).

3 Term algebras as the semantic domain for object societies

The problem of the Information Systems (IS) specification could be stated from our point of view as *how to define complex functions starting on the Abstract Data Types (ADT) world* (the data subspecification) using the facilities that functional languages usually offer. We will start this section introducing the well-known basic concepts of the algebraic specification framework; after them, we will give an overview of our approach, that is fully developed in the section 4.

3.1 Preliminaries

As we have said, we start in the ADT framework; to introduce the basic concepts we will follow the notation from [GTW 78], starting form the notion of indexed set.

Definition 1 *Let S be a set. A S-*indexed set *A is a collection of sets A_s for all index s in S. We will use $A = \{A_s\}_{s \in S}$ to denote S-indexed sets.*

Definition 2 Signature.
We call signature *Σ a pair (S, Ω), where S is a collection of* sort *names and Ω a $S^* \times S$-indexed family of operation symbols. $\Omega = \{\Omega_{w,s}\}_{w \in S^*, s \in S}$. If $\sigma \in \Omega_{w,s}$ then w is the* arity *of the symbol σ and s is its* sort.

Definition 3 Σ-Algebra.
Given a signature Σ, a Σ-algebra *$\mathcal{A}$ is composed of a non-empty domain A, called the* universe, *that contains a different* carrier *A_s for each sort $s \in S$ ($A = \{A_s\}_{s \in S}$), and a family F of sets of functions such that for each operation symbol $\sigma \in \Omega_{w,s}$, the corresponding function $\sigma_A \in F$ is defined as $\sigma_A : A \rightarrow A_s$, where $A = A_{s1} \times A_{s2} \times \ldots \times A_{sn}$. if $w = s1 \ldots sn$.*

We will always assume that universes are non empty, i.e., they will contain at least a constant. When S is unary we name the algebra as *one-sorted*. Otherwise we call it *many-sorted*. From now on we will consider the signature Σ as fixed and we will use only the word algebra instead of Σ-algebra.

Definition 4 Homomorphism.
An homomorphism φ from an algebra $\mathcal{A}$ to another algebra $\mathcal{B}$ is a family of applications $(\varphi_s)_{s \in S}$

$$\varphi_s : A_s \rightarrow B_s$$

that preserves the operations in Σ, in the sense that

$$\varphi_s(f_{\mathcal{A}}(a_1, \ldots, a_n)) = f_{\mathcal{B}}(\varphi_{s1}(a_1), \ldots, \varphi_{sn}(a_n))$$

for all $f \in \Omega_{w,s}$, $w = s1 \ldots sn$ and $a_i \in A_{si}$.

An *isomorphism* is a bijective homomorphism.

Definition 5 Free algebra.
An algebra $\mathcal{A}$ in a class of algebras $\mathcal{C}$ is free *over a set of variables X if X is a subset of $\mathcal{A}$ and for any algebra $\mathcal{B} \in \mathcal{C}$ and for any assignment $\theta : X \rightarrow \mathcal{B}$ there exists a unique homomorphism $\varphi : \mathcal{A} \rightarrow \mathcal{B}$ such that φ and θ agree over X.*

If the free algebra exists, it is unique up to an isomorphism.

Definition 6 Initial Algebra.
An algebra $\mathcal{A}$ in a class $\mathcal{C}$ of algebras is initial *if, for any algebra $\mathcal{B} \in \mathcal{C}$ there exists a unique homomorphism $\varphi : \mathcal{A} \rightarrow \mathcal{B}$.*

Proposition 1 *If $\mathcal{A}$ is initial in a class $\mathcal{C}$ of algebras then an algebra $\mathcal{B}$ is initial in $\mathcal{C}$ iff $\mathcal{A}$ and $\mathcal{B}$ are isomorphic.*

To some extent, that means that (abstract) data are representation-independent.

Definition 7 Ground term
Given a signature $\Sigma = (S, \Omega)$, the set of ground terms *of sort s, T_s, is defined inductively as follows:*

1. $\Omega_{\lambda,s} \subset T_s, \forall s \in S$.
2. $\sigma(t_1, \ldots, t_n) \in T_s, \forall \sigma \in \Omega_{s1 \ldots sn,s}$ *y* $\forall t_i \in T_{si}$ *with* $i = 1, \ldots, n$.

Definition 8 *The* ground term algebra *$\mathcal{T}$ is the algebra having as carrier the ground terms of the language, and where the function symbols are interpreted as*

$$\begin{array}{l} \forall \sigma \in \Omega_{s1 \ldots sn,s} \\ \forall t_i \in T_{si} \\ \sigma_{\mathcal{T}} : T_{s1} \times \ldots \times T_{sn} \rightarrow T_s \\ \sigma_{\mathcal{T}}(t_1, \ldots, t_n) = \sigma(t_1, \ldots, t_n) \end{array}$$

i.e., each function symbol is interpreted as itself.

Definition 9 Term Algebra.
Let $X = \{X_s\}_{s \in S}$ be a set of variable symbols such that X_s are variables of sort s. The Term Algebra $T(X)$ *is constructed by adding to the signature of T the elements of X_s as constants of sort s. Among all the algebras, the (absolutely) free algebra over X is (isomorphic to) $T(X)$.*

Definition 10 Equation
Given $t_1, t_2 \in T(X)$, an equation *between t_1 and t_2 is denoted by $(\forall x)t_1 = t_2$ or, assuming that variables are universally quantified, $t_1 = t_2$.*

Definition 11 Assignment
Given an algebra $\mathcal{A}$ and a set of variables X an assignment *θ from X to $\mathcal{A}$ is a mapping $\theta : X \rightarrow \mathcal{A}$ that assigns values of $\mathcal{A}$ to the variables of X. Thus, there exists a unique homomorphism from $T(X)$ to $\mathcal{A}$, denoted by $\theta^\sharp$, that extends θ.*

Definition 12 Model
Let E be a set of equations. An algebra $\mathcal{A}$ is a model *of E if and only if for each equation $t = s$ in E and for each value assignment to the variables in t and s, the meanings of t and s are identical. We call $Mod(E)$ the class of all the models of E; each one of them is an algebra.*

Definition 13 *An* Abstract Data Type *(ADT) is the class of isomorphic initial algebras for the ADT's signature.*

3.2 From ADTs to Objects

Traditionally, the ADT approach deals with values rather than objects. This is why some authors think that it is not an appropiate framework to formalize an Object-Oriented model ([FM 91]). Nevertheless, in this paper we develop a clear and powerful formal algebraic environment to characterize our specification language.

The main idea is to look at an OASIS specification as a term with variables $t(x)$ of the term language with variables $T_\Sigma(x)$ generated by a signature Σ. Any specification language will be characterized by its signature. Once it is fixed, the specifications written on that language will be terms $t(x) \in T_\Sigma(x)$. So, if we fix the signature of the OASIS language, we have a constructive way to write specifications. Furthermore, using a functional programming language interpreter we will have implementations of them[1].

Recent work on conditional rewriting logic — a logic of actions whose models are concurrent systems — ([Mes 90]) gives us the chance to deal with concurrence. Rewriting logic is implicit in term rewriting systems, but has passed for

[1] Previous work on OASIS environments ([RPC 91,CP 91a,CP 91b,PCA 92]) have been based on the construction of translators from an OASIS specification to the formal first order (clausal or equational) theory corresponding to it. Such a process of translation is not necessary in the presented environment, because a declarative and operational semantics can be directly associated to the specification language

the most part unnoticed due to our overwhelming tendency to associate term rewriting with equational logic. *Maude* ([Mes 90]) is a programming language whose modules are rewriting logic theories with well-defined declarative and operational semantics. It provides a simple unification of concurrent programming with functional and object-oriented programming, and supports high-level, declarative programming of concurrent systems. In this context, a significative research work is in progress to deal with concurrence in a natural way within our object-oriented specification environment by characterizing the rewriting logic theory that could be associated to a OASIS specification.

4 Object-Oriented Concepts Reification

Once we have defined the object-oriented concepts and chosen the semantic domain, let us reificate the elements of the former in terms of the latter. We will require to the reification mapping the following four properties:

1. To be a mapping preserving the structure of the operations (by *operations* in our model we mean concepts such as: instance-of, population-of, aggregation, grouping, etc.). Let us call *rep* such a mapping, *op* such an operation and let o and o' be two object-oriented concepts. Then

   for *op* unary we have $rep(op(o)) = op(rep(o))$, and
   for a binary *op* $rep(o\ op\ o') = rep(o)\ rep(op)\ rep(o')$,

   assuming an infix concrete syntax. The same must apply for n-ary operators.

   An alternate view to these equations is to say that certain diagrams commute. Following a cathegorial approach, we will have objects (o, o') and morphisms (rep, op) that constitute a cathegory with certain properties. But let us follow in the algebraic approach.

2. The semantic domain and the mapping *rep* have to capture properly the object oriented properties without introducing nor deleting other ones.

3. The process of reification (resp. de-reification) needs to be easy and straigthforward.

4. As we are interested on reasoning about the object oriented model (validating, verifying, etc.) in an automated way, the semantic domain must be easily computerizable.

Let's see now how the chosen semantic domain verifies these properties by showing how the different object-oriented concepts are reificated.

For a functional language such as F-OASIS we will define:

- A *Basic Language BL* that will be the term language of the signature $\Sigma_{BL} = (S_{BL}, \Omega_{BL})$. For the OASIS language we have:

  - S_{BL} = { `concep-sch, events, v-attrib-list, events, ...` }; we introduce a sort for each construct in the language.
  - Ω_{BL} will include four cathegories of operators:
    1. trivial constructors (TCR),
    2. trivial consultors (TCS),
    3. non-trivial constructors (NTCR), and
    4. non-trivial consultors (NTCS).

  The TCR will be used for building the specification equivalent terms (except for the second-order terms needed for defining the axioms that implement the observation function). The TCS will give us the elementary parts that constitute a class definition: events, preconditions, attributes (both constant and variable), triggering relationships and other *projections* of the classes. A set of equations will be introduced defining the TCS in terms of the TCR. In this way we have a presentation to give account for the architecture of the objects society. This is a first-order presentation.

- The extension of the language, Δ, that will be the term language associated to the signature $\Sigma_\Delta = (S_\Delta, \Omega_\Delta)$, where S_Δ = (`Instance, State`) and Ω_Δ = (`*ev, *ac, *av, obs`). These are the non-trivial constructors (`*ev`) and consultors (`*ac, *av, obs`). They are non-trivial in the sense that they are second-order concepts that will be used to construct the second-order axioms that model the implementation of the observation function, triggering relationships and other behavioural aspects of the object-oriented model.

  The signature Σ_Δ together with the axioms constitute a second-order presentation modelling events, lives, triggering relationships (actors) and in general all the behavioural properties.

Definition 14 *A class C will be represented by a term of the language generated by the signature* $\Sigma = (S, \Omega)$, *where* S = {`class-id, attrib-id, event-id, var-id, ...`} *(see page 16), and* Ω = { `class, attributes, constant, variable, ...` } *(id.).*

The reification of an elementary class will be a term with variables. The constructor `class` is the first operator of the term and it accepts as arguments a class identifier, the attributes, the events, the event preconditions and the triggering relationships for that class term, being all of them also terms. This term will belong to the sort `class-id` used in the domain. See as an example the term representing the class `person` in appendix C.

Definition 15 *An object that is an instance of a class C will be reificated as the term* `instance-of(attsk, attsc, C)`. *where* `attsk` *and* `attsc` *will be resp. the terms of the key and constant attributes that substitute the attribute classes (taken as variables) of the term representing the class C. The sort of this term will be* `instance`.

The attributes will be two new terms representing constant and variable attributes. The constant attributes term will be composed of three subterms: the key attribute, the constant attribute list and the static constraints. The variable attribute term will be a list where each element will contain the axiomatic definition —given in the chosen logic— of the observation function for the corresponding attribute:

Definition 16 *The constant attributes of a class will be reificated as the terms with functor* `key/constant-attributes` *and domain and codomain as shown in the signature (see next definition).*

Definition 17 *The Object Identifier (OId) concept will be represented in our model by the term representing the constant attribute prefixed by the unary operator* `key`. *Example:* `key("name"@string)`.

Definition 18 *The naming mechanism consists of an ADT that is given to the user —*`string` *in the previous example— with the OIds properties: infinite cardinality and built-in inequality. The user will choose at object creation time an element of the ADT —different to those already in use— as the key attribute, being a parameter of the* `instance-of` *operator (see definition 15).*

Definition 19 *The object-oriented concept* event *will be represented by the generic constructor of the signature* Σ `*ev`. *that accepts as first argument the event name.*

The different event names are elements of the carrier denoted by the sort `event-id`, and they are known only when the user writes a concrete class term (a specification). This is why we need to include the generic operator at the signature definition time. In the equations where the event participates (e.g., those defining the observation function and/or the preconditions) the variable used as first argument for the `*ev` operator will be universally quantified over the `event-id` carrier. We have so a second-order logic.

The same thing happens with the `class-id` and `attrib-id` sorts. And the same solution will be taken: to introduce generic operators (consultors for the attributes, `*ac`, `*av`) that introduce second-order characteristics in our reification. This second-order will be dealt with through first-order mechanisms using the syntactic machinery shown below.

Definition 20 *The life concept will be represented by a ground term built using* `*ev` *generic operators. For example:*

```
*ev("apply",1, *ev("new-person",1,"John Smith",instance-of(person, ...)))
```

The initial state of an object's life will be represented by the term of the sort `instance instance-of (...)`. That is to say, that term represents the object —i.e., some class' instance— in its initial state. Other states of the same instance will be represented by `*ev(...)`, like the term shown above. The final state —if that situation arrives— will be represented by `destroy(...)`. An implicit axiom exists in the reification:

```
destroy(*ev( ...instance-of(...) ...) = nil
```

for each defined class (element of the sort `class-id`).

Definition 21 *The concept of* state *of an object will have an equivalent double representation:*

1. *As the object's life (see definition 20). In this case we are in a update-oriented or backward strategy.*
2. *As the value of the observation function. Then we have a query-oriented or forward strategy.*

People from the Information Systems community would talk respectively about deductive and dynamic conceptual models!

The observation function will have the following syntax:

obs : `instance` —— `state`

being the elements of the sort `state` tuples composed of the attribute values in a given state.

Definition 22 *The* change of state *concept will be represented by the events.*

Definition 23 *An observation point will be represented by the evaluation of the obs function at a point.*

Once defined how a class is represented by a term, class operators (aggregation, part-of, generalization, specialization, ...) will be trivially represented as ordinary operators in the term algebra. Their syntax and semantics will be given in the usual way, following a pure algebraic style! This allows the users to define their very specific class operators. They only needs to specify their syntax and semantics. So, we have an *extensible specification language* that users can customize according their convenience. Obviously, the most usual operators will be predefined.

In this way we cope with the structural aspects of the object-oriented model in an elegant, well known and computerizable manner. The dynamic aspects are coped with using the second-order facilities already explained.

The price we have payed in our example is the *collapse* of the type system to a homogeneous one[2]. But in our opinion these are non-general constraints and depend only on our very concrete implementation. The consequences are that the answers to queries are given in a non-normal form. But at the stage of our work this is for the moment irrelevant.

5 Conclusions and future work

We have shown how term algebras constitute an appropiate framework to deal with the object orientation from a formal perspective. That seems to be very valuable due to the executability of the constructs used to reificate the object-oriented concepts, against other approaches perhaps more promising but hard to be provided with the necessary executability. Second-order aspects that arise when dealing with dynamic aspects are handled using a pure syntactic machinery, allowing us to stay on a first-order environment.

A first implementation has been made using *Axis* ([CDG 88]) as the algebraic specification language. Other languages such as *Obj* ([Gog 82]) will be used in the future in order to improve both the expressiveness and performances of the system.

Future work will include the definition of the most important class operators as well as the introduction of the appropiate mechanisms that make easy to incorporate new, user-defined class operators making the language truly extensible. On the other hand, the use of concurrent rewiting systems would improve the system performances and would keep the implementation closer to what the objects world is.

Acknowledgments Authors whish to thank José Cuevas and Jaume Devesa for their implementations as well as their valuable contribution to the first drafts of the paper.

References

[BCG 83] Balzer, R., Cheatman, T.E. and Green, C., *Software Technology in the 1990's: Using a New Paradigm*, IEEE Computer, Nov. 1983.

[BIM 91] *Prolog by BIM Release 3.0 Reference Manual*. ISS, Belgium, 1991.

[2] Due to the lack of genericity, coercions and retracts in the functional language employed; other languages allow the use of such facilities that would allow avoid this problem

[CP 91a] Canós,J.H. and Pastor,O. *Object Oriented and Functional Specification of Information Systems* in Proc. of DEXA-91,Berlin,1991.

[CP 91b] Canós,J.H. and Pastor,O. *Adding Logic Variables to a Functional and Object Oriented Specification Language*, in Proc. of IASTED Conference on Applied Informatics, Zurich, 1991.

[CDG 88] Coleman, D., Dollin, C., Gallimore, R., Arnold, P. and Rush, T., *An Introduction to the Aris Specification Language*, Technical Report, Hewlett-Packard Labs., Bristol, UK, 1988.

[Dav 88] Davison, A., *Blackboard Systems in Polka*, Dept. of Computing, Imperial College, London, 1988.

[ES 91] Ehrich,H.D. and Sernadas,A *Fundamental Objects Concepts and Constructions*, Proc. of the Second International IS-CORE Workshop. Imperial College,London-1991, pp.1-24.

[FM 91] Fiadeiro,J. and Maibaum,T. *Towards Object Calculi*, Proc. of the Second International IS-CORE Workshop. Imperial College,London-1991, pp.129-178.

[Gab 87] Gabbay, D. M., *Modal and temporal logic programming*, in A. Galton, editor. *Temporal Logics and Their Applications*. chapter 6, pages 197-237. Academic Press, London, December 1987.

[GTW 78] Goguen,J., Thatcher,J. and Wagner,E. *An Initial Algebraic Approach to the Specification, Correctness, and Implementation of Abstract Data Types*, in R.Yeh (ed) Current Trend in Programming Methodology, Vol.4, Prentice-Hall 1978,80-149

[Gog 82] Goguen,J., *Rapid Prototyping in the OBJ executable specification language* ACM Software Engineering Not. 1982,7 (5), pp.75-84.

[Mes 90] Meseguer, J., *A Logical Theory of Concurrent Objects*, in Proc. of the ECOOP-OOPSLA '90, Oct. 1990.

[PCA 92] Pastor, O., Cuevas, J. and Alpuente, M., *Functional, Relational and Object-Oriented Specification of Information Systems*, in Proc. of the IASTED Conference on Applied Informatics, Innsbruck (Austria), Feb. 1992.

[JHS 91] Jungclaus,R., Hartmann,T., Saake,G. and Sernadas,C. *Introduction to TROLL- A Language for Object Oriented Specification of Information Systems* Second International IS-CORE Workshop. London-1991.

[PHB 92] Pastor, O., Hayes. F. and Bear, S., *OASIS: An Object-Oriented Specification Language*, in Proceedings of the CAISE-92 Conference, Springer-Verlag, 1992.

[RCO 89] Ramos, I., Canós, J.H., Forradellas, R. and Oliver, J., *A Conceptual Schema Specification System for Rapid Prototyping*, Proc. of the XI IASTED , Feb. 1990.

[RAM 90] Ramos, I., *Logics and OO-Databases: A Declarative Approach*, Proc. of the DEXA-90, Springer-Verlag, 1990.

[RPC 91] Ramos,I., ,Pastor,O. and Casado,V. *OO and Active Formal Information System Specification* In Proc, of DEXA-91, Springer-Verlag, Berlin, 1991

[SSE 89] Sernadas, A. , Sernadas, C. and Ehrich, H.-D, *Object-Oriented Language Features for Information Systems Specification*, INESC 89.

[Wan 89] Wand, Y., *A proposal for a formal model of Objects*, in Kim and Lochovsky (eds). *Object-Oriented Concepts, Databases and Applications*, ACM Press, Addison Wesley, 1989.

[Wie 91a] Wieringa, R.J., *A Formalization of objects using equational dynamic logic*, Proc, of the DOOD'91 Conference, Springer-Verlag, 1991.

[Wie 91b] Wieringa,R. *A Conceptual Model Specification Language (CMSL version 2)*, Technical Report,Dep.of Mathematics and Computer Science,Vrije Universiteit, Amsterdam,Abril 1991

A Example: the OASIS specification of the class Person

```
class person
constant_attributes
    key person_code:string;
    person_name:string.

variable_attributes

    is_cand(person,time):bool
        formulas P:person; C:company; T:time.
        is_cand(P,T)=false :- new_person(P,T).
        is_cand(P,T)=true :- apply(P,T).
        is_cand(P,T)=false :- hire(C,P,T).
        is_cand(P,T)=true :- fire(C,P,T).
        end_formulas

    is_employee(person,time):bool
        formulas
        is_employee(P,T)=false :- new_person(P,T).
        is_employee(P,T)=true :- hire(C,P,T).
        is_employee(P,T)=false :- fire(C,P,T).
        end_formulas

private_events
    new new_person(P,T);
    destroy destroy_person(P,T);
    apply(P,T).

shared_events
    hire(C,P,T);
    fire(C,P,T).

preconditions
    # person can apply only once
     apply(P,T) :- is_cand(P,T)=false,is_employee(P,T)=false.
     destroy_person(P,T) :- is_employee(P,T)=false.

triggering
self ::- apply(P,T) :- new_person(P,T).

end_class
```

B The algebraic specification of the OASIS language

```
%% CLASSES

SPEC cl
SORTS class_id
OPS nat: -> class_id
    bool: -> class_id
    string: -> class_id
    _:id -> class_id
ENDSPEC

%% ATTRIBUTES

SPEC at
SORTS attrib_id
OPS _:id -> attrib_id
ENDSPEC

%% EVENTS

SPEC ev
SORTS event_id
OPS _:id -> event_id
ENDSPEC

%% VARIABLES

SPEC var
SORTS var_id
OPS _:id -> var_id
ENDSPEC

%% WELL-FORMED FORMULAE

SPEC fbf
USING var+
      at+
      ev+
      cl

SORTS atom
      formula
      term
      vector
OPS

nil_f : -> formula
_&_ : atom formula -> formula
_|_ : atom formula -> formula
not : formula -> formula

_==_ : term term -> atom
```

```
_<<_ : term term -> atom
_>>_ : term term -> atom
_=<_ : term term -> atom
_>=_ : term term -> atom
_<>_ : term term -> atom

r~ec : class~id var~id -> atom

nil~t : -> term
_ : var~id -> term
_ : nat -> term
_ : bool -> term
_ : vector -> term

_++_ : term term -> term
_--_ : term term -> term
_**_ : term term -> term
_//_ : term term -> term

r~atc : attrib~id var~id term -> term
r~atv : attrib~id var~id term -> term
r~ev : event~id var~id var~id var~id -> term

nil~vc : -> vector
_!_ : term vector -> vector

ENDSPEC

%%   CONCEPTUAL SCHEMA

SPEC ec

USING cl+
      at+
      ev+
      var+
      fbf

SORTS

      concep~sch    events                  v~attrib~list
      e~class~list    private                  v~attrib~def
      e~class          priv~event~list        v~attrib~decl
      attributes        priv~event              v~attrib~eq~list
      c~attrib          shared                   v~attrib~eq
      k~attrib          sh~event~list
      c~attrib~list    sh~event                prec
      c~attrib~def    v~attrib                   prec~formula
      nok~attrib       constraints             prec~formula~list
      trig                 trig~relation            trig~relation~list
      trig~address   trig~formula

 OPS

%%   TRIVIAL   CONSTRUCTORS
```

```
conceptual~schema : class~id e~class~list -> concep~sch
nil~e : -> e~class~list
_;_ : e~class e~class~list -> e~class~list

class : class~id attributes events prec trig -> e~class

attributes : c~attrib v~attrib -> attributes

constant : k~attrib nok~attrib constraints -> c~attrib
key : c~attrib~def -> k~attrib
nokey : c~attrib~list -> nok~attrib
constraints : formula -> constraints
nil~c : -> c~attrib~list
_;_ : c~attrib~def c~attrib~list -> c~attrib~list
_@_ : attrib~id class~id -> c~attrib~def

variable : v~attrib~list -> v~attrib
nil~v : -> v~attrib~list
_;_ : v~attrib~def v~attrib~list -> v~attrib~list
_equations_ : v~attrib~decl v~attrib~eq~list -> v~attrib~def
_@_ : attrib~id class~id -> v~attrib~decl
nil~eq : -> v~attrib~eq~list
_;_ : v~attrib~eq v~attrib~eq~list -> v~attrib~eq~list
_==_ : term term -> v~attrib~eq

events : private shared -> events

private : priv~event~list -> private
nil~p : -> priv~event~list
_;_ : priv~event priv~event~list -> priv~event~list
new : event~id class~id -> priv~event
destroy : event~id class~id -> priv~event
normal : event~id class~id -> priv~event
update : event~id class~id class~id -> priv~event

shared : sh~event~list -> shared
nil~s : -> sh~event~list
_;_ : sh~event sh~event~list -> sh~event~list
sh : event~id class~id class~id -> sh~event

preconditions : prec~formula~list -> prec
nil~pc : -> prec~formula~list
_;_ : prec~formula prec~formula~list -> prec~formula~list
_if_ : atom formula -> prec~formula

triggering : trig~relation~list -> trig
nil~tr : -> trig~relation~list
_;_ : trig~relation trig~relation~list -> trig~relation~list
_::_ : trig~address trig~formula -> trig~relation
self : -> trig~address
object : -> trig~address
class : -> trig~address
_if_ : atom formula -> trig~formula
```

```
*ev : event~id term term term -> term

%% TRIVIAL CONSULTORS

atk : e~class -> c~attrib~def
atc : e~class -> c~attrib~list
atv : e~class -> v~attrib~list
constr : e~class -> formula
evp : e~class -> priv~event~list
evsh : e~class -> sh~event~list
ev~new : e~class -> event~id
term~atc : attrib~id term c~attrib~list -> term
term~eq : attrib~id event~id v~attrib~list -> term

%% NON-TRIVIAL CONSULTORS

*ac : attrib~id term e~class -> term
*av : attrib~id term e~class -> term

FORALL
        cid,cid1 : class~id
        ct : constraints
        cal : c~attrib~list
        ka : k~attrib
        noka : nok~attrib
        val : v~attrib~list
        ca : c~attrib
        va : v~attrib
        pel : priv~event~list
        p : private
        sel : sh~event~list
        s : shared
        a : attributes
        e : events
        f : formula
        pc : prec
        tr : trig
        aid,aid1 : attrib~id
        eid,eid1 : event~id
        ec : e~class
        cad : c~attrib~def
        vaeql : v~attrib~eq~list
        t,t1,t2,t3,t4 : term
        v1,v2,v3 : var~id
        vc : vector

AXIOMS for atk:
        atk(class(cid,attributes(constant(key(cad),noka,ct),va),e,pc,tr))=cad

AXIOMS for atc:
        atc(class(cid,attributes(constant(ka,nokey(cal),ct),va),e,pc,tr))=cal

AXIOMS for constr:
        constr(class(cid,attributes(constant(ka,noka,constraints(f)),va),e,pc,tr))=f
```

```
AXIOMS for atv:
        atv(class(cid,attributes(ca,variable(val)),e,pc,tr))=val

AXIOMS for evp:
        evp(class(cid,a,events(private(pel),s),pc,tr))=pel

AXIOMS for evsh:
        evsh(class(cid,a,events(p,shared(sel)),pc,tr))=sel

AXIOMS for ev~new:
        ev~new(class(cid,a,events(private(new(eid,cid);pel),s),pc,tr))=eid

AXIOMS for term~eq:
        %% no variable attributes
        term~eq(aid,eid,nil~v) = nil~t

        %% looking for the appropiate attribute
        term~eq(aid,eid,aid1@cid1 equations vaeql ;val) = term~eq(aid,eid,val)
                IF not aid==aid1

        %% no equations for the variable attribute
        term~eq(aid,eid,aid@cid1 equations nil~eq;val) = nil~t

        %% looking for the appropiate equation
        term~eq(aid,eid,aid@cid1
                equations r~atv(aid,v1,r~ev(eid1,v1,v2,v3))==t3;vaeql;val) =
        term~eq(aid,eid,aid@cid1 equations vaeql;val)
                IF not eid==eid1

        %% this is the equation!
        term~eq(aid,eid,aid@cid1
                equations r~atv(aid,v1,r~ev(eid,v1,v2,v3))==
                                r~atv(aid,v1,v3)++v2 ; vaeql;val) = "+"

        term~eq(aid,eid,aid@cid1
                equations r~atv(aid,v1,r~ev(eid,v1,v2,v3))==
                                r~atv(aid,v1,v3)--v2 ; vaeql;val) = "-"

        term~eq(aid,eid,aid@cid1
                equations r~atv(aid,v1,r~ev(eid,v1,v2,v3))==
                                r~atv(aid,v1,v3) ; vaeql;val) = "="

        term~eq(aid,eid,aid@cid1
                equations r~atv(aid,v1,r~ev(eid,v1,v2,v3))==v2 ; vaeql;val)
                                                                = "~"

        term~eq(aid,eid,aid@cid1
                equations r~atv(aid,v1,r~ev(eid,v1,v2,v3))==t1 ; vaeql;val) = t1
AXIOMS for term~atc:
        %% no constant attributes
        term~atc(aid,nil~t,nil~c) = nil~t
        term~atc(aid,nil~vc,nil~c) = nil~t

        %% looking for the appropiate constant attribute
        term~atc(aid,t!vc,aid1@cid;cal) = term~atc(aid,vc,cal)
```

```
            IF not aid==aid1

        %% this is the constant attribute!
        term~atc(aid,t!vc,aid@cid;cal) = t


AXIOMS for *av:
        %% no trace
        *av(aid,nil~t,ec) = nil~t

        %% looking at the previous state
        *av(aid,*ev(eid,t,t1,t2),ec) = *av(aid,t2,ec) ++ t1
                IF term~eq(aid,eid,atv(ec)) == "+"

        *av(aid,*ev(eid,t,t1,t2),ec) = *av(aid,t2,ec) -- t1
                IF term~eq(aid,eid,atv(ec)) == "-"

        *av(aid,*ev(eid,t,t1,t2),ec) = *av(aid,t2,ec)
                IF term~eq(aid,eid,atv(ec)) == "="

        %% frame rule
        *av(aid,*ev(eid,t,t1,t2),ec) = *av(aid,t2,ec)
                IF term~eq(aid,eid,atv(ec)) == nil~t

        %% variable
        *av(aid,*ev(eid,t,t1,t2),ec) = t1
                IF term~eq(aid,eid,atv(ec)) == "~"

        %% constant
        *av(aid,*ev(eid,t,t1,t2),ec) = term~eq(aid,eid,atv(ec))
                IF (not term~eq(aid,eid,atv(ec)) == nil~t)
                and (not term~eq(aid,eid,atv(ec)) == "+")
                and (not term~eq(aid,eid,atv(ec)) == "-")
                and (not term~eq(aid,eid,atv(ec)) == "=")
                and (not term~eq(aid,eid,atv(ec)) == "~")

AXIOMS for *ac:
        %% no trace
        *ac(aid,nil~t,ec) = nil~t

        %%   new
        *ac(aid,*ev(eid,t,t1,t2),ec) = term~atc(aid,t1,atc(ec))
                IF ev~new(ec) == eid

        %% frame rule
        *ac(aid,*ev(eid,t,t1,t2),ec) = *ac(aid,t2,ec)
                IF not ev~new(ec) == eid


ENDSPEC
```

C A class as a term with variables

The following is the representation of the class **person** of the appendix A as a term of the language shown in appendix B. We have abbreviated it for the sake of shortness.

```
class (
        "person",
        attributes(
                constant
                    key("person_code"@string),
                    nokey("person_name"@string;nil~c),
                    constraints(nil~f)
                   ),
                variable(
                    "is_cand"@bool equations
                        r~atv("is_cand","p", r~ev("new_person"."p","t")==false;
                        (...) ;
                        nil~eq;
                        (...)
                        nil~v
                        )
                   ),
       events(
          private(
                  new("new_person","person");
                  destroy("destroy_person","person");
                  update("apply","person");
                  nil~p
                  ),
          shared(
                 sh("hire","p","c");
                 sh("fire","p","c");
                 nil~s
                 )
            ),
     preconditions(...),
     triggering(...)
     )
```