OASIS: An Object-oriented Specification Language

Oscar Pastor Lopez[1]
Fiona Hayes[2]
Stephen Bear[2]

[1] Departamento de Sistemas Informaticos y Computacion (DSIC)
Universidad Politecnica de Valencia
Camino de Vera S/N
Apartado 22012
46071 Valencia.- Spain
phone: +34 6 3877350
fax: +34 6 3877359
email: plo@dsic.upv.es

[2] Hewlett Packard Labs.- Bristol
Filton Road, Sotke Gifford
Bristol UK
BS12 6QZ
email: fmh@hplb.hpl.hp.com
email: sb@hplb.hpl.hp.com

Abstract

This paper introduces Oasis, a language for specifying object-oriented information systems using a deductive (temporal) approach ([3]). Oasis extends first versions of OBLOG ([17]) and MOL([12]), a trace based specification languages, with:

1. *triggered relationships* which enable specification of active objects
2. supporting rapid prototyping by generating the First Order Theory formally equivalent to a specification.
3. introducing class operators within an algebraic formal environment to deal with object reification.

1 Introduction

Object-oriented approaches are increasingly popular as a useful paradigm covering the classical software development steps of analysis, design and programming. Several notations and methods are provided for object-oriented analysis and design[4],[2],[12],[15] in addition to a well-known collection of object-oriented programming languages such as C++[18] and Smalltalk-80 [7].

The object-oriented specification language MOL[12] defines the object-oriented paradigm by combining general system theory adapted to information systems[8], abstract object types of OBLOG[16] and clausal or equational logics. The operational semantics of the underlying logics provides an environment in which specifications of passive Information Systems can be executed by generating the Logic Program that represents the First Order Theory equivalent to a Specification ([12]). However, MOL cannot deal with the specification of active objects.

This paper presents Oasis, a language for the specification of open and active information systems with the following most relevant properties:

- extends MOL by defining *triggered relationships*, a new kind of active relationship between objects.
- uses a declarative approach for specifying Information Systems.
- defines class operators in an formal algebraic environment to deal with object reification
- allows rapid prototyping by generating and manipulating the First Order Theory equivalent to an Oasis Specification, as shown in [14].

In this paper, we focus on the set of notations that Oasis provides for developing specifications of object-oriented information systems. It is intended as the basis of an automated object-oriented software production environment.

2 The Object-oriented Model

Our objective is to develop an environment for specifying **open active information systems.**

First we define our terms. According to IFIP WG 8.1 definitions[8], a system is a collection of elements called the **system domain** which has at least one systematic property in relation to the environment that is not possessed by any of the elements. A **system view** is the set of elements of the system domain, the domain of the environment and the relationships between these elements which are necessary to explain the system.

The task of the analyst in object-oriented system specification is to specify a system view of the object system in order to produce the **conceptual model specification**. This approach is useful as object-oriented concepts can closely model real-world phenomena. Using object-oriented approaches, the semantic gap between what the system is and how it is to be represented is narrowed.

An object is a self contained operational unit, which has properties called **attributes**. It has a state that is the state of the set of its attributes. Each object has a name that identifies it during its existence.

Change of state is a fundamental concept. Objects may change state. The creation and destruction of objects change the state of the system. Individual objects may also change their state i.e. the value of their attributes. A **process** causes changes of state in an object. It is limited in time. The elementary parts of a process responsible for a single unified change of state are called **events**. An event is an abstraction of a change of state in the object system. It is discrete, has no duration and occurs at a certain point in time.

An object may be active or passive. We say that an object is **active** if is seen as involved as necessary for a change of state to take place. A **passive** object is one that cannot be seen as active within a relevant period of time.

A **trigger** is a relationship between an event and one or more other events that on a certain level of abstraction expresses the cause for the proper agents

to carry out the execution of such other events. Potentially, every condition on events and/or attributes may serve as a trigger.

Behaviour is the dynamic manner of an object. The object exists during a temporal period, its life, that has a limited duration. Events happen at a certain point of time.

Objects belong to **types**. A type is a set of objects composed by the universe of objects of a class. A **class** is a set of properties which characterises the structure and behaviour that a collection of objects share.

In this environment, the linguistic operator 'instance of' works on a class and yields a member of the corresponding type as the result. The operator 'population of' yields the type for a class. We may have classes and subclasses denoting types and subtypes. Subtype populations are subsets of the supertype populations. Subclasses are defined by adding more properties from the superclasses. We define generalisation/specialisation as relations between classes and subclasses at the same abstraction level.

In the following section, we describe the language (**OASIS**), intended for the executable specification of open and active information systems.

3 Object-oriented Representation of Information Systems

3.1 Introduction

An object has two aspects; structure and behaviour. The structural aspect defines the object composition. A simple object's structure is given by the corresponding typed attributes. In complex classes built using class operators, the structure is defined by the composition of component class structures. The behavioural aspect alludes to the object life represented by the set of events and admissible traces. An observation function relates both aspects. The structure of an object is expressed as a function of its behaviour.

3.2 Class definition

An object class may be formalised as a 4-tuple (X,A,T,ob) where:

- X is the event set, including private and shared events. *Private events* are declared in only one class and participate in the traces of objects of only this class. *Shared events* must be declared in more than one class. They will form part of the traces of objects of all the classes in which they are declared.

  Events have parameters. In particular private events have always as the first parameter a surrogate value which identifies the object to which the event belongs. The parameters of a shared event must include a surrogate value for each object which shares the event.

- A is the typed attributes set, including *constant attributes* (those whose values do not change during the object life) and *variable attributes* (those that change depending on event occurrence)

  Every object instance has to have its own and unique surrogate identifying it during its life. This surrogate will be assigned once the object is created, by means of the corresponding object class creation event occurrence. The surrogate or key is defined as a combination of constant attributes.

  Each attribute has a parameter representing the object owner class. The parameter is a value of the surrogate data space. Attributes are typed: every attribute takes its actual value from its type.

- T$\subset X^*$ is the lifecycle set, equivalent to the admissible event sequences. They will represent the possible object instances.

  Every object trace will be made up of events of the event set X of its corresponding class. The precise specification of the correct traces for a class characterises behaviour.

  Each trace representing an object instance will be composed by an initial creation event, assigning to the object its surrogate and constant attributes, an adequate sequence of object class event occurrences and finally, a (optional) deletion event finishing its existence. All the events composed in an object trace will have as one argument the corresponding object key surrogate. Moreover, each event of the trace has the same key surrogate.

  The statement of what we mean by adequate for traces will describe, as said before, the object behaviour.

- $ob : T \rightarrow obs(A)$ is the observation function. Using it we will obtain for a given trace representing an object its corresponding set of pairs attribute-value obs(A) giving us the object attribute values as a function of the event sequences.

Classes are built over domains. Domains contain a set of values and a set of operations on these values. The domain type is the carrier set of the corresponding abstract data type. Domains denote data subspecification for classes and are used for object surrogates and attribute types. Their instances always exist, they are neither created nor destroyed and they do not have a changing state. Examples include the class Nat and Bool with obvious types {0,1,2...}, {true,false}.

If a class has events and attributes, and it is not built using class operators, it is an elementary class. Complex classes are built by combining elementary classes using the given class operators.

A conceptual schema for an information system can be incrementally built by combining classes. It is defined as the resultant class given by parallel composition of all the objects in the system. Its corresponding 4-tuple definition will have as A the set of all attributes of all defined classes, as X the set of all

events, as T the interleaving of the object traces representing all the system life and as ob the observation in a given instant of the state of the object society.

An object is represented by a set of observable attributes over a life made up of its event trace. We can describe its structure in terms of its set of events and attributes, and its behaviour by its trace. The trace can be expressed in process algebra[5],[10], using petri nets or by specifying preconditions of events.

The link between structure and behaviour is given by the **observation function**[1] ob which defines the set 'obs' of object attributes-value pairs in the observed state. An observation indicates values for some of the attributes. An attribute has an unique value, so it may appear at most once in an observation. If an attribute value is undefined, it does not appear. The empty observation thus expresses that all attributes are undefined. An empty trace e expresses that the object remains nonexistent. The observation of a nonexistent object is always empty.

For each attribute a_i, we assume a data type $type(a_i)$ which determines the values a_i can have. Since object universes are always represented by domain values (due to its corresponding data type key surrogate space), the case of object-valued attributes is included.

The observation function can be implemented in two ways. In a query-oriented system, the occurrence of an event triggers a change of state of the IS. The event occurrence triggers changes in a forward inference style. In an event-oriented system, event occurrences are stored and the observation function is evaluated by backward inference.

4 An Example: Alarm Clock System

Consider an alarm clock system composed by three object classes: an **alarm**, a **window** and a **clock** as our example. An alarm can open and close a window. When the alarm rings it opens a window. When the alarm is not ringing the window is closed (iconised). The alarm checks the clock for the actual time. When the alarm time is reached, the alarm causes the bell window to be opened. If the alarm is not stopped, the bell window is closed after a fixed duration.

Every change of the system state is due to an event occurrence. For example, in order for an alarm to move from quiet to ringing an **openwindow** event must be activated.

An alarm is initially off and can be set. Once set by means of a **set** event it may be cancelled with a **cancel** event occurrence. In the ringing state the window is closed after a fixed duration which causes a **closewindow** event occurrence. A user can stop the alarm by explicitly activating a **stop** event. An alarm must be in a quiet state before an alarm setting can be cancelled. Each

[1] We haven't studied the extension of our model to observation relations. This is an interesting issue to explore.

event occurrence has an associated precondition that must be satisfied in order to activate it.

Each clock instance is related to a set of alarm instances, giving them the actual time continously. This time is represented by a **time** variable attribute that gives us the 'now'. The passing of time is represented by **time_unit** event occurrences.

Window instances will be identified by a bell identifier value. They may contain text written by repeated **type** events. Their **openwindow** and **closewindow** events shared with alarm instances will respectively open and iconise the window.

Next, we present the provided interaction mechanisms between objects. We will use and develop this example in the rest of the paper.

5 Interaction between Objects

There are two interaction mechanisms between objects:

1. **event sharing**: Shared events are those belonging to more than one class event set. They participate in the life cycles of the classes sharing them. When a shared event happens, it is added to all the traces of the relevant class instances sharing it.

   In our example we will have two events **openwindow** and **closewindow** shared between the **alarm** and **window** classes. Each occurrence of them will appear in the traces of objects of both the alarm and window.

   Shared events will have as first arguments the identifiers of all the objects that are sharing the given event.

2. **triggered relations**: Objects in our object society can have active relationships between them. A typical case is when an event occurrence is the cause of others event occurrences. We state these *triggered events* by declaring the so-called triggered relationships, that will show us which events occur in an automatic way when another event (the trigger) is activated.

   Any event belonging to a class can trigger events of any other class if some optional stated preconditions holds. For example, in the alarm clock system, a **closewindow** shared event occurrence is triggered when the alarm is stopped by a **stop** event occurrence, As **closewindow** is a shared event between the alarm and window classes, triggering it adds it to both the relevant alarm and window traces.

   Triggering relationships are expressed by

   $$e1/e2[\text{if } pc2]$$

   meaning that an event e1 occurrence will trigger an event e2 if the precondition pc2 (if present) is satisfied.

A similar notion of triggers is used (in a Data Base environment) in OZ+ ([19]), an Object-Oriented Database System that introduces the concept of self-triggering rules as parameterless rules that execute whenever all their statements are executable.

Also in a Data Base context, Ode ([6]) declares triggers in a class specification, by defining conditions and its related actions. In terms of Oasis, conditions are event's occurrences and the subsequent action is the triggered event.

In both cases, a main difference lies in the declarative and deductive specification style used by Oasis contrasting the dynamic approach used in OZ+ and Ode.

These two mechanisms of interaction between objects will give us the key for expressing the traces defining the active behaviour of a system.

6 Representation of the Observation Function

The last component of our object class definition is the observation function ob. Given a trace $t \in T, ob(t)$ will map the attribute names of the object represented by t into attribute values of their types.

The observation function defines the values of constant and variable attributes. The constant attributes take their values when the creation event happens. An object's variable attributes values depend on the events of the object.

The observation function can be represented using first-order Horn clause logic. In this case the semantic observation function will be represented in the logic as a set of functions[2](one for each attribute). The resulting language is called Relational-OASIS. The observation function can be defined using equational logic for Functional-OASIS. The two approachs can also be combined. These language versions will differ only in the variable attribute definition. The executability of the Oasis language is derived from the representation of the observation function.

7 Types of classes

The specification of a society of interacting objects is based on three main constructs; domains, elementary classes and complex classes.

Each construct is defined in this section and is illustrated in the Oasis language. The alarm clock system presented in section 4 and in [1] illustrates the ideas. Other examples are presented in [11].

[2]modelled by relations

7.1 Domains

Domains denote the data subspecification and are used as object surrogates and attributes classes. Our object society will be built taking them as the basic data types upon which elementary classes are declared. They give us the set of unchanging 'platonic' entities that will be used for object identification (via object surrogates), and attribute types in our class definition.

The domains used in the object society will be declared at the beginning of the specification. The syntactic form is as below;

domains nat,bool,time,string

7.2 Elementary classes

Elementary classes are primitive and built only from the data domains. Each one has

1. a set of constant and variable attributes. A subset of the constant attributes define the surrogate.
2. a set of private and shared events,
3. a set of traces, describing the object behaviour
4. and an observation function, defining every object state as a function of its relevant event occurrences.

The syntactic elementary class representation will follow the template:

```
class name
  attributes
    constant
          ...
    variables
          ...
  events
    private
          ...
    shared
          ...
  preconditions
    [event:pc]
  triggering
    event1/event2 [if condition]
end class.
```

The two first components of our formal class definition are given by the attribute and event declarations. The adequate set of traces is expressed by means of the preconditions paragraph which associates to each event the precondition that must hold for an event occurrence. The observation function is

represented in the variable attribute definition, defining with a deductive style every variable attribute in terms of their relevant events.

The triggering paragraph will allow us to establish active relationships between objects.

For example, the clock specification is

```
elementary class clock
   attributes
      constant
        code: string key
      variable
      ** the 'now', giving us the actual time value **
        time?(clock):time
          clauses: c:clock, t:time, sa:alarmsets
          time?(c)=t :- time_unit(c,sa,t).
   events
      private  c:clock, t:time
         newclock(c,t).
         delclock(c,t).
         setclock(c,t).
      shared c:clock, sa:alarmsets, t:time
         time_unit(c,sa,t).
   preconditions
         newclock(c,t):- not clock(c,t).
         delclock(c,t):- clock(c,t).
         time_unit(c,sa,t):- clock(c,t).
   triggering c:clock, sa:alarmsets, a:alarm, t:time
         time_unit(c,sa,t)/time_of_alarm_clock(a,t) if a in sa.
end class clock
```

The clock class has one constant attribute **code** used as key surrogate, and one variable attribute **time?**. The **time?** definition in terms of its relevant event **time_unit** constitutes the observation function representation. Three events are declared: **newclock** and **delclock** are the private creation and deletion events. **time_unit** is a shared event between clock and a complex class alarmsets, which groups individual instances of alarm class[3]. Each **time_unit** occurrence is shared between an instance of class clock and an instance of the alarmsets complex class (representing a set of individual alarm instances).

The triggering paragraph states the active clock class behaviour. Each **time_unit** occurrence will activate a **time_of_alarmclock** occurrence in every alarm clock belonging to the alarmsets grouping object denoted by sa.

The event **time_of_alarmclock** is an alarm class event.

[3]Complex classes construction is explained later. In particular, we will formally define the grouping composition.

7.3 Complex classes

Complex classes are those defined by class operators. They provide a constructive way for specifying an information system. Complex classes are built by composing other classes using one of four operators;

- aggregation/projection
- generalisation/specialisation,
- grouping
- and parallel composition.

We now state how each of the class operators is defined in terms of our 4-tuple class representation.

7.3.1 Aggregration and Projection

The aggregation class operator combines component classes. The resultant complex class has a constant attribute corresponding to the cartesian product of the component class surrogates. It also has its own set of attributes and events. Aggregation is used to abstract the shared behaviour of components. The approach is similar to aggregation as presented in static General Semantic Models (in particular, the Extended Entity-Relationship Model).

Given two classes C1=$\{X_1,A_1,T_1,ob_1\}$ and C2=$\{X_2,A_2,T_2,ob_2\}$ the definition of the aggregated class C=$\{X,A,T,ob\}$ is as follows:

- the set X will be composed of:
  1. the set of shared events between the component classes, declared as private events in the complex class and identified by taking the intersection of the component class event sets
  2. its own set of declared private and shared events
- the set A of attributes declared in the complex class: one constant attribute of the aggregate class will be the cartesian product of the component class keys.
- the set T of traces is built over the class events, but with the following constraint. Given C an aggregated class and C1 one of its component classes, we have that

  $\forall t \in T, \exists t_1 \in T_1$ such that the projection of t_1 on those of its shared event that are private in C, is equal to the projection of t on these events. And also, the key of the object represented by t_1 is just the projection of the key of the object represented by t on its C1 component.

  and the corresponding converse condition:

  $\forall t_1 \in T_1, \exists t \in T$ such that the projection of t on its private events that are shared in C1, is equal to the projection of t_1 on these events. And the

projection of the key of the object represented by t on its C1 component is just the key of the object represented by t_1

This two conditions state that you cannot have a situation in which an event e of class C that is also a shared event between classes C_1 and C_2 occurs without satisfying the preconditions stated for e in everyone of these classes. It also states that if the precondition for a shared event is satisfied in one component class, it must be satisfied in all components that share it in the aggregate.

- the observation function is given as usual. There is no relation between the observation function of the aggregate class and the observation function of the component classes as they have no common variable attributes.

An aggregated class C of two classes C1,C2 is declared as C=C1*C2. We can break it down by means of the projection operator, resp. C[1], C[2]. It will give us the class surrogates of the components (C1 and C2 class surrogates respectively).

In this example, the alarm and window classes can be aggregated into a complex class icon representing their shared behaviour. Icon is defined by

ICON=ALARM*WINDOW

Each icon class instance is an aggregation of the alarm instance activating an **openwindow** event and the window instance being opened. Once again, we will have a constant attribute composed by the aggregation of its component class keys. The icon class may have an independent set of attributes (such as the time of the window opening etc.). The **openwindow** and **closewindow** shared events between alarm and window classes are private events of icon.

7.3.2 Generalisation and Specialisation

A generalised class is one built over a set of classes by abstracting their common features. Its corresponding inverse operator, the specialisation, allows us to define specialised classes from a parent class, by adding new events and attributes, or by redefining any of the inherited from the parent class.

The generalisation/specialisation operators are used to represent inheritance in Oasis.

An specialised complex class from another 'parent' class is intended to have the following interpretation:

- the set of events will be composed of:
  1. the set of events of the parent class, now owned by the specialised class
  2. newly (private or shared) defined events.
- the set of attributes contains all the parent attributes. New attributes may be added. The key of the specialised class is composed of at least the parent key constant attributes.

- the set T of traces, built as usual but with one constraint. When an event is triggered in the parent it must also be triggered in the specialisation. This means that for P a parent class with a child class C, and t_p and t_c two traces of their respective set of traces having the same key surrogate value, if an event $e \in T_p$ occurs and is relevant to the trace t_p, then it will be also relevant for the child trace t_c.
- the observation function, represented as in elementary classes. No special restriction is required on the relationships between the involved observation functions, to allow for the free redefinition of any inherited variable attribute definition.

The 4-tuple definition of a generalised class can be obtained in a similar way. The generalisation C of two classes C1 and C2 is defined by the following 4-tuple:

- $A = A_1 \cap A_2$.
  The generalised class key is the non empty intersection of C1 and C2 key constant attributes.
- $X = X_1 \cap X_2$ respecting the change in surrogate keys.
- T built as usual over X, with the same constraint stated when dealing when specialisation: an occurrence of any event $\in X$ will be a member of both the generalised and the component class traces.
- $ob = ob1$ restricted to the attributes $\in A$ (the intersection attributes set). We are assuming that we have the same observation function definition for the common attributes.

Syntactically, a generalised class C is declared by C=C1+C2. The inverse operation (specialisation) is allowed by means of the 'using' clause. So, we define: *class C1 using C* defines a specialised class C1.

For example, a specialised round_window class can be defined by inheriting events and attributes from window.

class **ROUND_WINDOW** using WINDOW

Or inversely, assuming defined two classes **round_window** and **squared_window** with the same surrogate keys, a generalised class window is defined by

WINDOW=ROUND_WINDOW+SQUARED_WINDOW

The surrogates of the generalisation are the common key attributes of the component classes. In this simple example, if we assume that the key attribute of round_window and squared_window is a bellid, the WINDOW generalised class would have as key the bellid constant attribute.

7.3.3 Grouping and Ownership

Complex classes can be defined using the grouping operator as in 'collection' classes. The complex classes instances comprise a collection of instances of the grouped class.

The component instances of a grouping class can be given an ordered structure, as *lists, queues or stacks* or an unordered structure such as *sets* or *bags.*

The complex grouping class has two special features;

- It has a variable attribute *members* which is a generic type such as set, list, queue, stack of X where X is the surrogate type of the component class. Oasis provides syntactic constructs for defining a grouped class as desired. The definition of this *members* attribute is given as a function of the corresponding events of element addition and deletion, commented below.
- they will always have two specific events:
  1. insertion of new components to the grouped class instance. This insertion event is equivalent to the classical *push* for stacks or *insert* for the rest.
  2. deletion of existent components. Deletion corresponds to *pop* for stacks and *delete* for the others.

  This insertion and deletion events will change the *members* attribute value, including or removing a component from the grouped class instance. They are implicitly declared in the complex grouping class.

The corresponding 4-tuple definition has:

- the set X of events declared in the new complex class, plus the two of insertion and deletion of components in grouping instances.
- the set A of attributes declared in the complex class, plus a variable attribute *members* as stated earlier.
- the set T of traces, built as usual.
- the observation function, represented as usual.

Grouping classes can be defined using the following explanatory keywords; setof, bagof, listof, queueof and stackof. A grouping class may also be defined by a clause 'group by'+condition, where condition defines the criteria for the grouping operation. The condition has the general form

$$attribute\ [\ OP\ value]$$

, where OP is a comparison operator of the attribute type. The use of the comparison operators $(<, >)$ will be allowed only if we have a partially ordered attribute type.

Component classes may be grouped by attribute. In this case, the complex class defines instances as collections of the component class with the same value

for the attribute. We can build more complex grouping conditions using the classical logical operators *and, or, not.*

As an example, alarms may be grouped by by the maximum duration they may ring, the **finish** attribute. A complex grouping class ringing_alarm is defined as

class **RINGING_ALARM**= SETOF(ALARM) grouped by finish.

Ringing_alarm is a class with potential instances setof(alarm) with finish=50, ..., setof(alarm) with finish=60 and so on if we assume finish is a natural number.

7.3.4 Parallel Composition

Finally, the parallel composition class operator allows us to define a whole Conceptual Schema as a composition of previously defined classes.

For $C1=\{X_1,A_1,T_1,ob_1\}$ and $C2=\{X_2,A_2,T_2,ob_2\}$ two classes, we define the parallel composed class $C=\{X,A,T,ob\}$ ($C=C1||C2$). An instance of such a class will denote an element of the set of all the possible subsets of the union of the surrogate spaces of its component classes. So, if C_1 has a key k_1 of type t_1, and C_2 has a key k_2 of type t_2, an instance of C can be made up of o_1 of type t_1, or $\{o_1{:}t_1,o_2{:}t_2\}$, or $\{o_1{:}t_1,o_1{\prime}{:}t_1,o_2{\prime}{:}t_2\}$, etc.

The 4-tuple definition for C will have:

- a set X of events, the union of event sets of the component classes , all of them viewed as private events, plus its own creation and deletion events. The new and destroy events of the component classes are interpreted as private events of the composed class. They have a new parameter which represents the surrogate of the parallel composed class.

  For example, if we build a parallel composed class **alarmsystem** from **alarm**, **window**, **clock** and a grouping class **alarmsets**, the alarm component class **newalarm** event, creator of alarm instances, is regarded as a private event **newalarm**(s:alarmsystem,a:alarm) of the alarmsystem.

- a set A of attributes, the union of those (constant and variables) of the component classes. The key attributes of the component classes are constant attributes of the composed class. Each attribute is tagged with the new complex class surrogate.

  For example, in our Alarm Clock System, the **bellid** string constant attribute of the **window** component class is a constant attribute **bellid** of the alarmsystem and is tagged by the alarmsystem surrogate.

  bellid(w:window,t:string) $\rightarrow$ **bellid**(s:alarmsystem,w:window,t:string)

- a set T of traces, composed by interleavings components traces, respecting the synchronisation constraints for shared events and triggered relationships.

A further requirement for a parallel composed class is that events of the composed class are activated if and only if their preconditions are satisfied in each relevant component class.

- an observation function which is the 'sum' of the components' observation functions.

The syntax for the parallel operator is ||. Our alarm clock system conceptual schema alarmclock is defined by the parallel composition of the elementary classes alarm, clock and window and the grouping class alarmsets.

ALARMCLOCK=ALARM||WINDOW||CLOCK||ALARMSETS.

An alarmclock trace will be composed of any correct sequence of events of the component classes as defined in [1].

8 Conclusion

This paper presents an object-oriented specification language Oasis which extends notions from OBLOG[17] and MOL[12] to enable the specification of active systems.

Oasis provides the basis for a complete object-oriented software production environment. The operational semantics is based on clausal or equational logic which support the validation of specifications by software prototyping. Specifications can be animated using logic programming or by algebraic rewriting techniques. The development of an environment consisting of graphical tools, validation and code generation tools is now in progress.

References

[1] S.Bear,P.Allen,D.Coleman,F.Hayes. 'Graphical Specification of Object Oriented Systems'. OOPSLA 90.

[2] G.Booch 'Object Oriented Design with applications' Benjamin/Cummings 1990

[3] Bubenko,J.A.:Olive,A. *Dynamic or Temporal Modelling? An Illustrative Comparison* SYSLAB Working Paper 117,Nov.1986

[4] Coad,P.,Yourdon, E. 'Object Oriented Analysis' Englewood Cliffs Prentice-Hall 1990

[5] C.A.R. Hoare. 'Communicating Sequential Processes, Prentice-Hall International, 1985.

[6] Gehani,N.:Jagadish,H.V. *Ode as an Active Database:Constraints and Triggers* Proceedings of the 17th International Conference on Very Large Data Bases,VLDB 1991, Barcelona.

[7] A.Goldberg, D.Robson 'Smalltalk:The language and its implementation' Addison Wesley 1983

[8] P.Lindgreen ed. 'A framework of Information System Concepts'.FRISCO Interim Report. IFIP WG8.TG.90.

[9] J.W.Lloyd 'Foundations of Logic Programming' Springer-Verlag 1987

[10] R.Milner. 'A Calculus of Communicating Systems' Lecture Notes in Computer Science, vol 92,Springer-Verlag, 1980

[11] O.Pastor 'OASIS:Open and Active Specification of Information Systems' Internal Technical Memo. HP-Labs.Bristol.

[12] I.Ramos. 'Logics and OO-Data Bases:a declarative approach.' DEXA 90

[13] I.Ramos et al. 'A Conceptual Scheme Specification for Rapid Prototyping' XII IASTED Conference on Applied Informatics. Insbruck 90.

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

[15] Rumbaugh,J.:Blaha,M.:Premerlani,W.:Eddy,F.:Lorensen,W. *Object-Oriented Modelling and Design* Prentice Hall 1991.

[16] A.Sernadas et al. 'Abstract Object Types: a temporal perspective' Colloquium on Temporal Logic and Specification.

[17] Sernadas,A.:Sernadas,C:Ehrich,H.D. *Object Oriented Specification of Databases: An Algebraic Approach.* Proc. 13th Int.Conf. on Very Large Data Bases VLDB'87,Brighton,1987. Morgan-Kaufmann, Palo Alto, 1987, pp. 107-116.

[18] B.Stroustrup 'The C++ Programming Language' Addison-Wesley 1987

[19] Weiser,S.P.:Lochovsky,F. *OZ+:An Object Oriented Database System* Object Oriented Concepts, Databases and Applications, ACM-Press 1989