Animation of System Specifications Using Concurrent Logic Programming

P.Letelier, P.Sánchez, I.Ramos
Department of Information Systems and Computation
Valencia University of Technology

Camino de Vera, s/n, 46071 Valencia - Spain
☎ 3877350; Fax: 3877357
email {letelier | ppalma | opastor | iramos} @dsic.upv.es

ABSTRACT

OASIS is a Language for the Specification of Object Oriented Conceptual Models. In OASIS the conceptual schema of an information system is represented as a society of concurrent objects interacting with one another. Animation of such models in order to validate the Specification of Information Systems is an interesting topic. We have developed a Basic Execution Model for OASIS Specifications. The Model includes features such as the representation of attributes, events, derivations, preconditions, evaluations and triggers. On the other hand, Concurrent Logic Programming is a suitable paradigm for distributed computation because it allows a natural representation of concurrence. Using Concurrent Logic Programming we are able to animate OASIS Specifications according to the OASIS Execution Model. In this work we show how the object-oriented (OO) concepts of an OASIS Specification are represented directly using Concurrent Logic Programming concepts. We give an example of a bank account codified in Parlog language and we show parts of the program that animates its corresponding OASIS specification. A CASE tool supporting the OASIS Model has been implemented and our aim is to add a module for animation of specifications.

1. Introduction

OASIS [8][9] is a Formal Language for the Conceptual Specification of Information Systems. Nowadays there is a growing interest in formal approaches to model information systems. CMSL [18] and TROLL [5] are languages near to OASIS that address the system specification in a similar way. OO-Method [11] is the companion methodology for the OASIS approach.

It is important to animate the system specification by means of a prototype automatically generated, so that the analyst can validate the requirements. Experiments have been carried out using Petri Nets [13] and Concurrent Logic Programming (CLP) [6] as semantics domains for OASIS Specifications. These efforts have led to the establishment of a basic Execution Model [7]. This model can be used in order to animate OASIS Specifications implemented over concurrent programming environments.

The aim of this paper is to present the mapping between OASIS concepts and clauses in CLP. Experiments have been carried out with Parlog [1] and Kl1 [2] as Concurrent Programming Languages, which have got similar features. We will concentrate on Parlog to show some examples because the code is simplier.

2. Basic Concepts of OASIS

OASIS is a language for Open and Active System Information Specification. In the object model of OASIS, an ***object*** is an observable process encapsulating structure and behaviour. An object is an operational unit that has static and dynamic features. Each object has an internal and unique identifier (*Oid*) that allows identifying it during its whole existence. Object behaviour is characterised by the actions received, those which the object is able to carry out, and by the actions sent, those which it is able to request. Actions are received and sent by interaction mechanisms between objects.

The object properties are represented by ***attributes***. Thus, the ***state*** of the object in a given moment is determined by the value of its attributes. An ***event*** is the abstraction of a change of state. The state can change after the occurrence of an event that modifies the values of some attributes. An event does not have duration and it occurs in an instant of time.

Each object possesses a fixed set of events that could affect it during its existence. The state of the object in a given instant will depend on the events occurred in its life until the moment during which is observed. In this way an object is seen as an observable process.

The attributes of the object are classified as follows: ***constant*** if its value does not change during the life of the object, ***variable*** if its value can change as a consequence of the occurrence of events. Furthermore, if the value of an attribute is obtained from the value of other attributes, the attribute is said ***derived***.

Two kinds of events can be distinguished: ***private*** and ***shared***. Private events participate in the life of objects of only one class, they appear only in the signature of one class or in several classes belonging to the same hierarchy of specialisation. Shared events are part of the life of objects of more than one class. These events appear in the signatures of all the classes that share them.

Agents (also called ***clients***) activate the events and they have this responsibility. We will name ***action*** the event requested by an agent object to some ***server***

object(s). An event will only be relevant in the life of an object if the state of the object is suitable for that. ***Preconditions*** are formulas and they should be satisfied in order to allow the occurrence of an event.

The objects are not isolated. In OASIS the interaction between objects is modelled in two forms: by ***events sharing*** and by ***triggers***. Triggers introduce activity in the society of objects, allowing an object to act as an agent whenever some conditions are satisfied in its state.

The values of attributes must follow some rules. These rules are named ***integrity constraints***. They are defined by formulas in some kind of logic. They should be satisfied in all the states in the life of the object (static constraints) or among certain sequences of states (dynamic constraints).

A group of events that operates as a unit of execution of greater granularity is called ***transactions***. They follow the two basic principles for transactions: all or nothing, and not observation of intermediate states.

It is possible to construct ***processes*** using events and transactions as atomic actions, they allow specifying the correct sequences of actions in the life of the objects.

A ***template*** is a set of common properties that are shared by several objects. A ***class*** is defined as the group of objects that share the same template. An individual object is called a ***class instance***.

The Society of Objects of our object-oriented system is built from primitive classes, elementary classes and complex classes. Objects that have only one state constitute ***the primitive classes*** or domains and always exist. In general, they undertake the domain of the abstract data types. The ***elementary classes*** are built from the primitive classes. In the template, attributes, events, transactions, preconditions, integrity constraints, triggers and process description are defined and they characterize the objects of the class.

The ***complex classes*** are defined from the elementary ones by the application of operators between classes. The operators that allow the construction of complex classes are ***aggregation***, ***association***, ***specialisation*** and ***parallel composition***. These operators are orthogonal, that is, they are independent and they can be combined.

OASIS is a formal specification language that allows the definition of conceptual schemas according to the object model that has been presented. Here is an example of an account in a bank system.

```
class Account
identification
   by_id:(id).
constant_attributes
   id:nat;
   name:string.
variable_attributes
   balance:int;
   number:nat (0).
derived_attributes
   good_balance:bool.
private_events var N:nat.
   open new;
   close destroy;
   deposit(N);
   withdraw(N);
   pay_commission.
valuation
   [deposit(N)] balance=balance+N and number=number+1;
   [withdraw(N)] balance=balance-N and number=number+1;
   [self:pay_commission] balance=balance-100 and number=0.
derivation
   good_balance={balance>100000}.
preconditions
   withdraw(N) if balance>=N;
   close if balance=0.
triggers
   self::pay_commission if number>10 and good_balance=false.
processes
   ACCOUNT  =   open.ACCOUNT0;
   ACCOUNT0 =   close +
                deposit(N).ACCOUNT0 +
                withdraw(N).ACCOUNT0 +
                pay_commission.ACCOUNT0.
end_class
```

This concrete syntax is formalized in dynamic logic [10] and it has been used in [2].

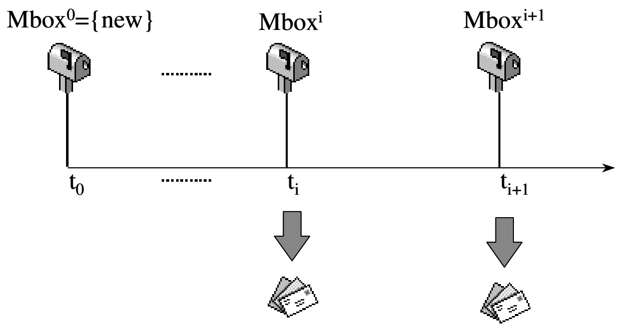

Figure 1: Object life cycle

The life cycle of an object is shown in Figure 1, according to the previous concepts. Each object has a mailbox. The change of state does not have duration and occurs at a precise instant of time, when the actions received in the mailbox are executed. To **attend** an action means to extract it from the *Mbox*. The attended actions can be **admitted** or **rejected** depending on whether they satisfy or not their preconditions. Thus, $Accept^i$ will be the set of admitted actions at the instant t_i , then $Accept^i \subseteq Mbox^i$.

Considering $Exec^i$ as the set of actions executed when the state is $State^i$, then $Exec^i \subseteq Accept^i$. Therefore, an object will change from the state $State^i$ to the $State^{i+1}$ by the **execution** of one or more actions included in $Exec^i$. Intra-object concurrence is allowed. The constraint is that the set of executed actions is not in conflict [7].

Requested actions received by an object are called ***service actions***. In the instant t_i , if an object must send an action to a determined server(s), there will be an associated action in the *Mbox* to remind it about this obligation. Those reminding actions are called ***trigger actions***. It is to say, for each action that the object must trigger whenever it reaches the state $State^i$ it will exist a trigger action in the mailbox at the instant t_i. Thus, the server perspective of an object is integrated with its client perspective in a homogeneous form of operation. This means, the object must serve service actions as well as trigger actions.

Service actions in $Mbox^i$ are represented by $Mbox^i_{services}$. Trigger actions in $Mbox^i$ are represented by

$Mbox^{i}_{triggers}$. Therefore, it is always true that:

$$Mbox^{\iota} = Mbox^{\iota}_{triggers} \cup Mbox^{\iota}_{services}$$

At each t_i $Mbox^{i}_{triggers} \subseteq Exec^{i}$, because all the trigger actions have to be executed. Thus, $Exec^{i}$ can additionally contain a subset of accepted service actions that are in conflict neither with one another nor with the trigger actions. The contents of the mailbox must be updated in each t_i:

(a) Trigger actions or service actions at t_{i-1} will have been extracted.
(b) All the new actions received until t_i must be included.
(c) The actions accepted and not executed in previous states (because they were in conflict) must be stored again.

All these ideas have been considered in order to define a basic execution model including the main features of OASIS specifications. This model allows animating an OASIS specification accurately.

3. CLP & OASIS

The concurrent logic languages rise as an attempt to improve the efficiency of logic languages by means of the exploitation of stream AND parallelism [11]. Besides, they are high level programming languages and are very convenient for parallel and distributed systems.

A concurrent logic program is a set of Horn Clauses with Guards.

$$H \leftarrow G_1,\ldots,G_n : B_1,\ldots,B_m \qquad n,m \geq 0$$

A goal has the following form:

$$\leftarrow M_1,\ldots,M_k \qquad k > 0$$

The M_i are evaluated in parallel (AND parallelism), using the clauses of the program for their reduction.

For each M_i the clauses that can reduce it (OR parallelism) are examined in parallel, selecting only one and backtracking is not allowed.

The criterion of selection is that the M_i unifies with the head of the clause, and the conjunction of guards $G_1,\ldots,G_n$ (evaluated also in AND parallelism) will be satisfied, if they exist. If more than one clause could be candidate, then a sequential search can be established (in textual order from top to bottom) separating those clauses by ";"

The integration of Concurrent Logic Programming and the OO model have generated many researches and corresponding proposals. We are interested in modelling objects as perpetual processes. This approach is inspired by the works by Shapiro and Takeuchi [12], in which an object is implemented as a tail-recursive process that passes the update state components of the object as arguments in the recursive call. The identity of the object is the name of an input stream argument of the process into which all messages for the object are merged. The works in this direction are principally based on making OO extensions to concurrent logic languages. Within this approach some proposals are: Polka [4] and L2||O2 [10] based on Parlog, and A'UM [17] based on GHC [16].

The main motivation in the proposals of integration between CLP and the OO model has been to provide OO qualities for a concurrent logic language in order to take advantage of the benefits of OO methods for systems development. Thus, the result has been a Logic and OO Programming Language. However, in the research frame of OASIS, our aim is to use Concurrent Logic Programming as a tool for:

a) Experimenting with OO concepts defined in OASIS. As it will be shown, many OO concepts have a direct and natural representation in Concurrent Logic Programming.
b) Providing a declarative and operational semantics for OASIS from the perspective of Concurrent Logic Programming and according to the equivalences settled down.
c) Providing a concurrent implementation support that allows the animation of OASIS specifications

4. Objects and Classes in CLP

Now we will sketch the essential features that allow to consider an OASIS specification as a CLP program. A more complete description can be found in [6].

In CLP a society of objects can be seen (at run time) as a goal to solve, where each object is a subgoal. Each object is evaluated using AND parallelism (inter object concurrence).

$$\leftarrow object_1(In_1,Attrib_1),\ldots,object_k(In_k,Attrib_k) \qquad k > 0$$

In_i is a channel for input messages, mixture of the messages received from itself and from other objects. **$Attrib_i$** are the attributes of the object.

In general, creation (and destruction) of objects at run time is required. Subgoals that have the capacity of generating in their reduction a new instance of an object should exist. These subgoals correspond in natural form to the notion of class. Thus, classes are implemented as other objects of the society in the same way as in [3]. A class has an attribute called ***Population***, which is a list of pairs (Mbox,Key) for each class instance. Mbox is the identifier of the input channel of an instance, Key is a set of constant attributes that allow referring to the object.

Example: The class accounts in our bank system (at run time) is represented in CLP as a goal. Classes are goals at the beginning of the execution. Thus the class account appears as the following subgoal:

$$\leftarrow \ldots,account(Account,Population),\ldots$$

Classes will carry out the following operations:

- To create instances in the class, verifying the non-existence and updating the variable Population.
- To answer messages asking for instance existence in the class.
- To distribute messages to instances in the class.

4.1 Object Behaviour

Each object attempts to reduce itself using the clauses that represent its object specification. Each clause recognizes one action and is able to reduce the object to a set of subgoals. Here are an object goal and clauses with which this object goal could be reduced. Considering $p,q,s,t \geq 0$ and $r > 0$:

$$\leftarrow \ldots,\ object_i(In_i, Atrib_i),\ \ldots$$

$$object_{i1}([mensaje_{i1}|In_i], Atrib_i) \leftarrow$$
$$G_{i11},\ldots,G_{i1p} :$$
$$B_{i11},\ldots,B_{i1t}.$$
$$\ldots$$
$$object_{ir}([mensaje_{ir}|In_i], Atrib_i) \leftarrow$$
$$G_{ir1},\ldots,G_{irq} :$$
$$B_{ir1},\ldots,B_{irs}.$$

4.1.1 Object Creation

The classes are in the initial goal. We obtain the instances by means of reduction of its class when the action new occurs. The creation of an object implies that the following subgoals are generated:

```
create_channel(Oid,Mbox), object(Mbox,Attributes)
```

A variable ***Mbox*** associated to the term Oid is created. This logical variable can obtain a value by means of partial instantiation. That behaviour simulates a channel receiving actions. Whoever knows the value of Oid could write in this variable. ***Attributes*** is a list of pairs *(Attribute,value)* for each attribute of the object.

Example: the clause to which the class account reduces when the action new occurs (action with the open event). In the body of that clause there is the creation of a new object (goal) account.

```
account([action(Client,Server,open(Id,Name,Balance,Number)))
        |Rest],Population) <-
    create_channel(Oid,Mbox),
    derivate(Good_balance,Balance),
    o_account(Mbox,[(id,Id),(name,Name),
            (balance,Balance), (number,Number),
            (good_balance,Good_balance)]),
    account(Rest,[(Oid,Id)|Population]).
```

derivate(Good_balance,Balance) is a predicate that implements the derivation of attribute Good_balance based on the value of the attribute balance. When the event open(100,john,0,0,false) is received the goal account becomes:

```
account([open(100,john,0,0,false)|Rest],[Population])
```

This goal is reduced in the following two subgoals:

```
account(Rest,[(Mbox,100)|Population]),
o_account(Mbox,[(id,100),(name,john),(balance,0)
                (number,0),(good_balance,false)]).
```

That is, a new object account has been created.

4.1.2 Change of State

We say that an object is implemented as a perpetual process, because among the subgoals in which an object is reduced the same object appears. This produces the effect of continuity in the object life.

Whenever the object goal is thrown as subgoal in the reduction, some of its attributes may be modified. Thus a change of state due to the occurrence of the associate event is represented.

The effect "to execute action" is obtained in the reduction when the new input channel is used as the original one without considering the last executed action.

The formulas that define the change of state when the event is executed are subgoals that assign new values to attributes of the object.

Example: The execution of the action associated with the event deposit(10000) sent to the recently created object is represented as the reduction of the goal:

```
o_account(Mbox,[(id,100),(name,john),
         (balance,0),(number,0), (good_balance,false)])
```

This goal is reduced using the following clause:

```
o_account([action(Client,Server,deposit(N))|Rest],
        [(id,Id),(name,Name),(balance,Balance),
        (number,Number),(good_balance,Good_Balance)]) <-
    NBalance is Balance + N,
    NNumber is Number + 1,
    derivate(NGood_balance,NBalance),
    o_account(Rest,[(id,Id),(name,Name),
        (balance,NBalance),(number,NNumber),
        (good_balance,NGood_balance)]).
```

Thus the goal is reduced to

```
o_account(Rest,[(id,100),(name,john),(balance,10000),(number,1),
         (good_balance,false)])
```

That is, the object has changed its state.

4.1.3 Object Destruction

The destruction of an object is obtained by reduction that is carried out by selecting a clause whose body does not contain the same object as subgoal. Hence, the execution of the object ends.

Example: when the object executes the action related to the close event, the following clause is selected:

```
o_account([close|Rest],[(mbox,Mbox)|Att]) <-
    close_channel(Mbox).
```

4.1.4 Preconditions

Event preconditions are incorporated directly as guards. For each precondition for an event (in addition to the clause that recognizes the event) there is one clause that recognizes the "non-action", that is to say, the refusal of the action whose event has a precondition unsatisfied.

Example: Preconditions are implemented as guards in clauses that execute the non-action. Here is the sequence (clauses separated by ";") of clauses that verify these preconditions when the event withdraw(20000) is considered.

```
o_account([action(Client,Server,deposit(N))|Rest],
        [(id,Id),(name,Name),(balance,Balance),
        (number,Number),(good_balance,Good_Balance)]) <-
   not(Balance>=N):
      o_account(Rest,[(id,Id),(name,Name),(balance,Balance),
         (number,Number),(good_balance,Good_Balance)]);

o_account([action(Client,Server,deposit(N))|Rest],
        [(id,Id),(name,Name),(balance,Balance),
        (number,Number),(good_balance,Good_Balance)]) <-
      NBalance is Balance - N,
      NNumber is Number + 1,
      derivate(NGood_balance,NBalance),
      o_account(Rest,[(id,Id),(name,Name),(balance,NBalance),
         (number,NNumber),(good_balance,NGood_balance)]).
```

Only the second clause produces a change of state. The second clause however is only used if the first has not satisfied its guard.

4.2 Inter-Object Communication

The communication mechanism offered by CLP is to share variable among subgoals, interpreted as a communication channel among objects. In order to allow that many objects could communicate with a determined object, the channel concept implemented in Parlog was used with the primitives create_channel/2, write_channel/2 and close_channel/1. It is the same as to instantiate a variable by mixing an indetermined number of inputs. Hence, many senders could send actions to an object. There is a unique channel for each object and it is named Mbox.

4.2.1 Actions

Each object is suspended, waiting for actions in its input channel. The trigger of an action from an object to an other is produced by the goal:

```
write_channel(Channel,Action)
```

This sentence will appear in the body of a selected clause in the reduction of an object (subgoal).Channel is the identifier of the shared variable that constitutes the input channel of actions for the client object.

Until now, we have not made a precise distinction between event and action. Actions have the following structure:

```
action(Client,Server,Event)
```

Client is a term representing the client object, Server is a term representing the server object and Event is the event itself.

Actions must first arrive in the class in order to determine the channel of the server object. This channel identifier will be registered in the state variable Population.

4.2.2 Sending Actions

In [6], examples of implementation of several kinds of communication among processes (1:1, 1:N, N:M) are introduced. Here we will focus on the synchronous or asynchronous nature of the communication.

In CLP, the effect of sending an action from a client toward a server is produced when the client object shares a variable with the client. That variable is the input channel of the server. The sentence used is write_channel/2 and it is one of the subgoals that could appear in the body of a clause in the client object. We are interested in two kinds of communication:

a) **Asynchronous Communication**: the selected clause reduces the object to a sentence write_channel/2 and the object is reduced as a subgoal without waiting for the result of the communication.
b) **Synchronous Communication**: the selected clause reduces the object to a sentence write_channel/2, the action carries a not ground variable (incomplete messages [1]), and the object is forced to wait using the subgoal wait/1. When the server executes the action it assigns a value to the variable and this allows the client to continue its execution.

4.2.3 Triggers and Shared Events

In OASIS the communication among objects is established by means of Triggers and Shared Events.

A) Triggers: they represent asynchronous communication. Hence, triggers are represented by executing write_channel/2 without waiting for answer. Triggers are generated due to a change of state. Thus, in the body of each clause that represents a change of state related with a condition in the trigger, a goal that evaluates the condition and could shoot the trigger will be added.

Example: be the following goal representing an object account reaching a new state:

```
o_account(Mbox,[(id,100),(name,john),
   (balance,0),(number,11),(good_balance,false)])
```

In this case we are sure that the object mailbox contains the action: action(this,this,trigger(this,self,pay_commision) because the object must send such action. The clause that makes the reduction is:

```
o_account([action(this,self,pay_commision)|Rest],
        [(id,Id),(name,Name),(balance,Balance),
        (number,Number),(good_balance,Good_Balance)]) <-
      NBalance is Balance - 100,
      NNumber is 0,
      derivate(NGood_balance,NBalance),
      write_channel(Mbox,action(this,self,pay_commision),
      o_account(Rest,[(id,Id),(name,Name),(balance,NBalance),
         (number,NNumber),(good_balance,NGood_balance)]).
```

B) Shared Events: the agent of a shared event has to make a broadcast to all the objects that share that event. The fact that the action is executed in all servers must be controled. A dialogue among the servers is established and the protocol that they follow is similar to Two Phase

Commit protocol used in distributed databases. We have establish an algorithm to control this type of events [12].

5. Conclusions and Future Work

We have shown how the main features of our object-oriented Conceptual Modelling Language OASIS can be naturally and directly represented in a CLP program.

Using the execution model of OASIS as a guide we can guarantee an accurate animation of the OASIS specification.

We are now working to obtain CLP code automatically from an OASIS specification using the established correspondences. Our final aim is to integrate this work inside a CASE tool for system modelling supporting the OASIS model.

References

[1] Conlon T., *Programming in PARLOG*, Addisson-Wesley, 1989.

[2] Chikayama, T., *KLIC User s Manual. (version 2.001),* Institute for New Generation Computer Tecnology, Tokyo JAPAN ,March 1995.

[3] Clark K., Wang T., *D_Parlog++: Object Oriented Programming with Distributed Active Classes*, in “Formal Methods an Object Terchnology”, S.J.Goldsack and S.J.H.Kent (Eds), pages 63-85, Springer Verlag 1996.

[4] Davison A., *Polka: A Parlog Object-Oriented Language*, Tesis Doctoral, Depto. Computer Science, Imperial Collegue Londres, 1989.

[5] Hartmann T., Saake G., Jungclaus R., Hartel P., Kusch J., *Revised Version of Modelling Language TROLL*, Informatik-Bericht 94-03, TU Braun-schweig, 1994.

[6] Letelier P., Sánchez P., Ramos I., *Basic Concepts of OASIS Specifications Implemented Using Concurrent Logic Programming*, Technical Report, DSIC, Universitat Politècnica de València, 1996.(in spanish)

[7] Letelier P., Sánchez P., Ramos I., *A Basic Execution Model for OASIS Specifications in a Concurrent Environment*, Technical Report, DSIC, Universitat Politècnica de València, 1997. (in spanish)

[8] Pastor O., *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el modelo orientado a Objetos*, Tesis doctoral dirigida por Isidro Ramos, DSIC, Universitat Politècnica de València, 1992.

[9] Pastor O., Ramos I.,*OASIS versión 2 (2.2) : A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*, SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.

[10] Pimentel E., *L2||O2: Un Lenguaje Lógico Concurrente Orientado a Objetos*, Tesis Doctoral, Facultad de Informática, Universidad de Málaga, Marzo 1993.

[11] Pelechano V., Pastor, O., *Case OO-Method : Un entorno de producción automática de software orientado a objetos*, Proc.of CIL-95, Barcelona,1995.

[12] Sánchez P., Letelier P., Ramos I., *An Algorithm for to Inplementate Shared Events of OASIS*, Technical Report, DSIC, Universitat Politècnica de València, 1996. (in spanish)

[13] Sánchez P., Ramos I., *Object Petri Nets: Implementation Model for OASIS Specifications*, Technical Report, DSIC, Universitat Politècnica de València, 1996. (in spanish)

[14] Shapiro E., *The Family of Concurrent Logic Programming Languages*, in ACM Computing Surveys. Vol.21, ACM, pp.413-510, 1989.

[15] Shapiro E., Takeuchi A., *Object Oriented Programming in Concurrent Prolog*, en New Generation Computing, Vol.1, pp 25-48,1983.

[16] Ueda K., *Guarded Horn Clauses*, Technical Report, TR-103, ICOT, 1985.

[17] Yoshida K., Chikayama T., *A'UM: A String Based Concurrent Object-Oriented Language*, en Proc. of the Int.Conf. on FGCS, ICOT, pp.638-649, 1988.

[18] Wieringa R.J., *A Conceptual Model Specification Language (CMSL Version 2)*, Technical Report IR-248, Vrije Universiteit, Amsterdam, 1991.

[19] Wieringa R.J., *A Formalization of Objects using Equational Dynamic Logic*, 2nd Conference on Deductive and OO Databases, pages 431-452, Springer 1991, Lecture Notes in Computer Science 566.