# Object Interaction in Object-Oriented Deductive Conceptual Models

Carme Quer
Antoni Olivé

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona
Pau Gargallo 5
08028 Barcelona - Catalonia
{ cquer | olive }@lsi.upc.es

**Abstract.** We present the main components of an object-oriented deductive approach to conceptual modelling of information systems. This approach does not model object interaction explicitly. However interaction among objects can be derived by means of a formal procedure that we outline.
Based on our results, we discuss whether explicit object interaction is a desirable feature of conceptual models.

## 1 Introduction

The purpose of this paper is threefold:

a) To present the main components of an object-oriented deductive approach to conceptual modelling of information systems.

b) To outline a formal procedure for the derivation of object interactions in an object-oriented deductive conceptual model.

c) To show that most difficulties in the modelling of the dynamic aspect with current object-oriented methods arise because they try to model explicitly the interaction among objects, which is not necessary from a conceptual point of view.

Deductive conceptual models (DCMs) model an Information Base (IB) [JaR84] by means of predicates. Predicates may be *base*, corresponding to external events, *derived*, corresponding to entities, attributes and generated events, or *constraints*, corresponding to the integrity constraints that the IB must satisfy at any time. Deduction rules define derived facts in terms of base and/or other derived facts [Oli89,MSS92]. However, DCMs are flat, in the sense that they lack a structuring mechanism of the (usually large) set of predicates.

We provide here a combination of the deductive and object-oriented approaches, by which we group the IB predicates using the concept of object. We do not intend here to present specific language features, but the CIAM language [GKB82] could be a materialization of our approach. The object-oriented deductive conceptual models (ODCMs) thus obtained still have all advantages over the operational models [Oli86], while gaining in modularization.

ODCMs do not model object interaction explicitly. Any deduction rule may refer to any event, object or attribute. However, it is interesting to derive the object interactions implied by a given ODCM. This may be useful as a validation tool and as a preliminary step towards an ulterior object-oriented design. To this end, we have developed a formal procedure that produces the object interaction patterns that may result from an ODCM.

This procedure shows that object interaction embodies some design decisions and that several distinct object interaction patterns are possible. This might be the reason why the specification of the interaction among objects is a difficult issue in object-oriented methods, as can be seen from the great diversity of approaches taken by current methods [dCP92]. Our findings suggest that explicit object interaction should perhaps not be defined at the conceptual level.

The paper is structured in 6 sections. Next section presents the ODCMs, including an example that will be used throughout the paper. Section 3 briefly reviews the concepts of internal event and internal events model. Section 4 describes a procedure that uses this model to derive object interactions in an ODCM. Section 5 then discusses our approach with respect to the conventional object-oriented approaches. Finally, Section 6 gives the conclusions and points out future work.

## 2 Object-Oriented Deductive Conceptual Models

In this section we characterise the main features of an ODCM in a first-order logic framework. We try to concentrate on the main issues and, therefore, we refrain from explaining unnecessary details. We will use the ODCM example given in Figures 1 and 2 throughout the paper.

An ODCM consists of a set E of external event classes and a set O of object classes.

### 2.1 External Event Classes

External events correspond to the inputs received by the Information System from the environment. The definition of an external event class consists of:

- The *attributes* of the events. We distinguish between attributes provided by the environment and derived attributes, i.e. those attributes whose value must be computed when the event occurs. We assume that each event has an implicit time attribute that gives, in a suitable time unit (such as seconds), the instant when the event occurs. For each derived attribute, we define one or more deduction rules.

- The *constraints* that the events must satisfy to be valid. For each constraint we define a deduction rule. These constraints must be verified when the events occur.

*Deduction rules* have the form: $p(X) \leftarrow \Phi$, where $X$ is a vector of variables, $\Phi$ a first order formula, and where all free variables are universally quantified. Note that we do not restrict deduction rules to Horn clauses (with negation) at this level.

Additional details will be explained using the example of Figure 1. We assume that each event has a unique identifier, which is provided automatically by the system. A fact such as sale(z,t) is true if a sale event with identifier z occurred at time t. A fact such as qty(z:q) is true, where z is the identifier of a sale event, if the qty attribute of z has value q. Note the difference in notation: sale(z,t) denotes the existence of a sale event z at time t, while qty(z:q) denotes q as the value of the qty attribute of the sale event z.

Instances of new_prod event class have two attributes: prodno (with domain integer), which is provided by the environment, and prod (with domain the set of identifiers of objects of class PRODUCT), which is derived. The deduction rule states how the prod attribute is defined: in this case, the product identifier is just the prodno value. To save space, we have omitted similar deduction rules in the rest of Figure 1.

There is a constraint associated with event class new_prod. The name of the constraint is prod_exists, which is a predicate name. A fact prod_exists(p,t) is true, meaning that, the constraint is violated, if there exists an event new_prod(z,t), such that its prod attribute has value p, and p is a product at time t-1. Note that, as we explain below, product(p,t-1) is true if p is an object of class PRODUCT at time t-1. It is assumed that the external events that violate a constraint (a new_prod in this example) will be rejected. We also omit similar constraint rules in the rest of Figure 1.

```
external events

event new_prod (prodno: integer; derived prod: PRODUCT).
          prod(Z:P) ← prodno(Z:P).
          constraints
          prod_exists(P,T) ← new_prod(Z,T), prod(Z:P), product(P,T-1).

event new_cust (custno: integer; type: (w,r ); derived cust: CUSTOMER).

event new_store (stono: integer; derived sto: STORE).

event set_cost (prodno: integer; cost: integer; derived prod: PRODUCT).
          constraints
          prod_not_exists(P,T) ← set_cost(Z,T), prod(Z:P), not product(P,T-1).

event replenishment (prodno: integer; stono: integer; qty: integer;
                     derived prod: PRODUCT; ps : PROD_STO).

event sale (prodno: integer; stono: integer; custno: integer; qty: integer;
                     derived prod: PRODUCT; ps: PROD_STO;
                               cust: CUSTOMER; amount: integer).
          amount(Z:A) ← prod(Z:P), prodcost(P,T-1:Co),qty(Z:Q), A = Q*Co.

event payment(custno: integer; amount: integer; derived cust: CUSTOMER).
```

**Figure 1.** External events in the ODCM example

The deduction rule of attribute amount of sale events, defines that the amount of a sale z is *a* (amount(z:*a*)) if the prod attribute of z is *p* (prod(z:*p*)), the prodcost attribute of PRODUCT *p* at time t-1 is *co* (prodcost(p,t-1:*co*)), the qty attribute of z is *q* (qty(z:*q*)) and $a = q*co$.

Note that literals in deduction rules may be events (like new_prod(Z,T)), event attributes (like prod(Z:P)), objects (like product(P,T-1)), object attributes (like prodcost(P,T-1:Co)), and evaluable literals (like A = Q *Co).

### 2.2 Object Classes

The definition of an object class C consists of:

- The *existence rule*, which is a deduction rule that states the conditions that an object must satisfy to be a member of the class at a given time.

- The *attributes* of the objects. For each attribute, we define its name, domain and one or more deduction rules, which define the value of the attribute at a given time.

- The (static and dynamic) *constraints* of the objects. For each constraint, we define its name and a deduction rule.

- The events generated by the objects. For each *generated event*, we define its name, the event attributes and one or more deduction rules, defining when the event is generated and the value of its attributes.

We again give additional details using the example of Figure 2. We assume that each object has a unique identifier, which is usually given by the external events. A fact such as product(p,t) is true if p is an identifier of an object of class PRODUCT at time t. A fact such as prodcost(p,t:co) is true, where p is an identifier of a product, if the prodcost attribute of p has value co at time t.

The existence rule of class PRODUCT defines that a product p belongs to this class at time t if an event new_prod occurred at a time t1 ($t1 \leq t$), such that its prod attribute has value p. Existence rules for STORE and CUSTOMER are quite similar. Class PROD_STO corresponds to the aggregation of a PRODUCT and a STORE. In the example, it is assumed that all products are present in all stores: this is what defines the existence rule for PROD_STO. Note the use of evaluable predicate id(P,S,Ps) which relates the identifiers of PROD_STO to the product and store identifiers. If, for example, the identifier of a product_store is formed by the concatenation of the product and store identifiers then id(p1,s1,p1s1) could be a fact of this predicate. Note, however, that we do not need to further specify this predicate at the conceptual level. The existence rules for WHOLESALER and RETAILER define these classes as subsets of CUSTOMER.

Instances of the PRODUCT class have three attributes. To save space, we only give the domain (integer) of the first (prodnumber). The deduction rules give the definitions of these attributes. For instance, the qoh_value of a product p at time t is defined as the sum of the values at time t of this product in all stores. We assume that deduction rules are defined such that attributes are single-valued.

**class** PRODUCT

product(P,T) ← new_prod(Z,T1),T1 ≤ T,prod(Z:P).

**attributes**

prodnumber: integer.
prodnumber(P,T:Pn) ← new_prod(Z,T1),T1 ≤ T,prod(Z:P),prodno(Z:Pn).
prodcost(P,T:Co) ← set_cost(Z,T1),prod(Z:P),cost(Z:Co), T1 ≤ T,
¬∃W,T2 (set_cost(W,T2),prod(W:P), T2 ≤ T, T2 > T1).
qoh_value(P,T:Qv) ← sum(V,[value(Ps,T:V), id(P,S,Ps)],Qv).

**class** STORE

store(S,T) ← new_store(Z,T1),T1 ≤ T,sto(Z:S).

**attributes**

stonumber(S,T:Sn) ← new_store(Z,T1),T1 ≤ T,sto(Z:S),stono(Z:Sn).
count_belowzero(S,T:Bz) ← count(Z,[belowzero(Z,T1), T1 ≤ T,sto(Z:S)],Bz).

**class** PROD_STO

prod_sto(Ps,T) ← product(P,T),store(S,T),id(P,S,Ps).

**attributes**

sto_qoh(Ps,T:Q) ← sum(S,[sale(Z,T1),T1 ≤ T,ps(Z:Ps),qty(Z:S)],Qs),
sum(R,[replenishment(Z,T1),T1 ≤ T,ps(Z:Ps),qty(Z:R)],Qr),
Q=Qr-Qs.
value(Ps,T:V) ← sto_qoh(Ps,T:Q), id(P,S,Ps), prodcost(P,T:Co), V = Q*Co.

**events**

belowzero(prod:P, sto:S,time:T) ← sale(Z,T),ps(Z:Ps),qty(Z:Q1),
sto_qoh(Ps,T-1:Q), Q1 > Q, id(P,S,Ps).

**class** CUSTOMER

customer(C,T) ← new_cust(Z,T1),T1 ≤ T,cust(Z:C).

**attributes**

custnumber(C,T:Nc) ← new_cust(Z,T1),T1 ≤ T,cust(Z:C),custno(Z:Nc).
custtype(C,T:Ty) ← new_cust(Z,T1),T1 ≤ T,cust(Z:C),type(Z:Ty).
balance(C,T:B) ← sum(S,[sale(Z,T1),T1 ≤ T,cust(Z:C),amount(Z:S)],Sl),
sum(P,[payment(Y,T1),T1 ≤ T,cust(Y:C), amount (Y:P)],Py), B=Py-Sl.
units_sold(C,T:U) ← sum(Q,[sale(Z,T1),T1 ≤ T,cust(Z:C),prod(Z:P),qty(Z:Q),
prodcost(P,T1:Co),Co>1000],U).

**constraints**

negbalance(C,T) ← balance(C,T:B),B < 0.

**class** WHOLESALER **is a** CUSTOMER

wholesaler(C,T) ← customer(C,T),custtype(C,T: w ).

**class** RETAILER **is a** CUSTOMER

retailer(C,T) ← customer(C,T),custtype(C,T: r ).

**Figure 2.** Object classes in the ODCM example

Instances of the WHOLESALER and RETAILER classes inherit the attributes from CUSTOMER class.

There is a constraint associated with object class CUSTOMER. The name of the constraint is negbalance, which is a predicate name. A fact negbalance(c,t) is true, that is, the constraint is violated, if the balance of customer c is negative at time t. It is assumed that the external events (a sale in this example) that induce a violation of a constraint will be rejected.

There is a generated event associated with object class PROD_STO. The name of the event is belowzero, which is a predicate name, with three attributes. A fact belowzero(p,s,t) is true, that is, the event is generated, if the qty of a sale is greater than the sto_qoh at time t-1. Like external events, generated events have also an internal identifier, which is given automatically by the system when they are generated. Note that in class STORE an attribute is defined (count_belowzero) that gives the count of such events generated up to a given time for a given store.

## 3 Internal Events and Internal Events Model

In this section, we briefly review the key concepts of internal events and internal events model. See [Oli89,San90] for further details. In the next section we will use the internal events model as a basis for determining the object interactions involved in an ODCM.

### 3.1 Internal Events

There are three kinds of internal events:

1) insertion, which capture the notion of inserting a new object into a class or giving an initial value to an object attribute.

2) deletion, which correspond to deleting an object from a class or removing an attribute value of an object.

3) modification, which capture the notion of changing an attribute value.

For each object class obj we define an insertion and a deletion internal event predicate as follows:

$$\forall X,T \ [\iota obj(X,T) \leftrightarrow obj(X,T) \wedge \neg\, obj(X,T\text{-}1)] \quad (1)$$

$$\forall X,T \ [\delta obj(X,T) \leftrightarrow obj(X,T\text{-}1) \wedge \neg\, obj(X,T)] \quad (2)$$

Thus, for example, $\iota$product(x,t) is true if object x is a member of class PRODUCT at time t, and it was not a member of that class at time t-1.

For each attribute att we define an insertion, a deletion and a modification internal event as follows:

$$\forall X,T \ [\iota att(X,T{:}V) \leftrightarrow att(X,T{:}V) \wedge \neg\exists V1\ att(X,T\text{-}1{:}V1)] \quad (3)$$

$$\forall X,T \ [\delta att(X,T{:}V) \leftrightarrow att(X,T\text{-}1{:}V) \wedge \neg\exists V1\ att(X,T{:}V1)] \quad (4)$$

$$\forall X,T \ [\mu att(X,T{:}V,V1) \leftrightarrow att(X,T\text{-}1{:}V) \wedge att(X,T{:}V1) \wedge V \neq V1] \quad (5)$$

Thus, for example, μsto_qoh(ps,t:qb,qa) is true if the value for the attribute sto_qoh of product_store ps was qb at time t-1 and is qa at time t.

For notational convenience, we also associate an insertion internal event predicate to each external, constraint or generated event. Thus, ιsale(z,t) will be the internal event associated to sale(z,t), and ιprod_exists(p,t) will be the internal event associated to constraint prod_exists of event new_prod.

From the above definitions, we have the following transition axioms:

$$\forall X,T\ [obj(X,T) \leftrightarrow (obj(X,T\text{-}1) \wedge \neg \delta obj(X,T)) \vee \iota obj(X,T)] \quad (6)$$

$$\forall X,T\ [att(X,T{:}V) \leftrightarrow (att(X,T\text{-}1{:}V) \wedge \neg \delta att(X,T{:}V) \wedge \neg \mu att(X,T{:}V,V1)) \vee \iota att(X,T{:}V) \vee \mu att(X,T{:}V1,V)] \quad (7)$$

which define object existence or attribute values at time t in terms of object existence or attribute values at time t-1, and the internal events induced during a transition from t-1 to t.

### 3.2 Internal Events Model

From the definitions (1) - (5) given above, and using the transition axioms (6) and (7), we get the internal events model corresponding to a given ODCM. The internal events model consists of a set of rules. Each rule defines the condition for the occurrence of an internal event. We refer again to [Oli89,San90] for details on the procedure that derives automatically the internal events model.

Figure 3 shows part of the internal events model corresponding to the ODCM example. In the following, we comment the internal events rules related with object class PROD_STO. The first two rules define the conditions for the insertion of a product_store ps in its class, which are:

1) An internal event ιproduct(p,t) is induced in a transition, a store s exists at time t and the identifier of the new object is ps, or
2) An internal event ιstore(s,t) is induced in a transition, a product p exists at time t and the identifier of the new object is ps.

The next five rules define insertions and modifications of attributes sto_qoh and value. Note, from the ODCM definition, that such attributes values are never deleted, since object instances of the classes in our example are never deleted.

The last rule defines the condition for the generation of internal event ιbelowzero: when an ιsale occurs at time T with a qty greater than sto_qoh at time T-1.

**(in class** STORE)

rs.1 ιstore(S,T) ← ιnew_store(Z,T),sto(Z:S).

**(attributes)**

rs.2 ιstonumber(S,T:Sn) ← ιnew_store(Z,T),sto(Z:S),stono(Z:Sn).
rs.3 ιcount_belowzero(S,T:0) ← ιstore(S,T).
rs.4 μcount_belowzero(S,T:Cb,Ca) ← count_belowzero(S,T-1:Cb),
count(Z,[ιbelowzero(Z,T),sto(Z:S)],C), Ca = Cb+C.

**(in class** PROD_STO)

rps.1 ιprod_sto(Ps,T) ← ιproduct(P,T),store(S,T),id(P,S,Ps).
rps.2 ιprod_sto(Ps,T) ← product(P,T), ιstore(S,T),id(P,S,Ps).

**(attributes)**

rps.3 ιsto_qoh(Ps,T:0) ← ιprod_sto(Ps,T).
rps.4 μsto_qoh(Ps,T:Qb,Qa) ← sum(S,[ιsale(Z,T),ps(Z:Ps),qty(Z:S)],Qs),
sum(R,[ιreplenishment(Z,T),ps(Z:Ps),qty(Z:R)],Qr),
sto_qoh(Ps,T-1:Qb), Qa=Qb+Qr-Qs, Qa ≠ Qb.

rps.5 ιvalue(Ps,T:0) ← ιprod_sto(Ps,T).
rps.6 μvalue(Ps,T:Vb,Va) ← μsto_qoh(Ps,T:Qb,Q), id(P,S,Ps),
prodcost(P,T:Co), value(Ps,T-1:Vb),Va = Q*Co.
rps.7 μvalue(Ps,T:Vb,Va) ← sto_qoh(Ps,T:Q), id(P,S,Ps),
μprodcost(P,T:Cob,Coa), value(Ps,T-1:Vb), Va = Q*Coa.

**(generated events)**

rps.8 ιbelowzero(prod:P, sto:S,time:T) ← ιsale(Z,T),ps(Z:Ps),qty(Z:Q1),
sto_qoh(Ps,T-1:Q), Q1 > Q ,id(P,S,Ps).

**Figure 3.** Part of the internal events model of the ODCM example

## 4 Object Interaction in an ODCM

In this section, we outline a formal procedure that determines the object interactions implied by a given ODCM.

### 4.1 Target Events

We define as *target events* of an object class those internal events that must be produced by the object class or some of its instances in order to fulfil their task. More specifically, the target events are:

1) The insertion and deletion events of the objects. An object class must be able to create object instances and, thus, producing the corresponding insertion events. To each creation of an object instance corresponds an insertion event. An object instance must be able to remove itself from its class, producing the corresponding deletion event.

2) The insertion, deletion and modification of object attribute values. An object instance must be able to initialise, remove or change the value of its attributes producing the corresponding insertion, deletion or modification events.

3) The insertion event of constraints. An object instance or class must be able to produce such events. If they are indeed produced in a transition the current transaction must be rejected.

4) The insertion events of generated events. An object instance or class must be able to produce such events when the corresponding conditions are satisfied.

The target events can be easily determined from the internal events model: they correspond to the predicates in the head of some rule. In the example of figure 3, the target events of object class PROD_STO are: ιprod_sto, ιsto_qoh, μsto_qoh, ιvalue, μvalue and ιbelowzero.

### 4.2 Relevant Events

We define *relevant events* for an object class as those events that must be communicated to the object class or to some of its instances in order for them to be able to produce the target events. Relevant events are those that appear in the body of some rule of the internal events model.

In the example of figure 3, relevant events for object class PROD_STO are: ιproduct, ιstore, ιprod_sto, ιsale, ιreplenishment, μsto_qoh and μprodcost. Note that the same internal event may be both target and relevant for a class. In the example, μsto_qoh is both a target event (necessary for maintaining the value of attribute sto_qoh) and a relevant event (must be communicated in order to produce μvalue).

### 4.3 Induction of Events

Rules in the internal events model establish a relationship between relevant and target events: target events appear in the head of the rule, and relevant events appear in its body. We call *induces* to this relationship. Thus, we say induces(*a*,*b*) if *a* is a relevant event for *b*. Occurrence of internal event *a* may induce the occurrence of internal event *b*. Note that this is a potential induction, since actual induction depends on other conditions.

Thus, if we consider the rule:

ιbelowzero(prod:P, sto:S,time:T) ← ιsale(Z,T),ps(Z:Ps),qty(Z:Q1),
sto_qoh(Ps,T-1:Q),Q1 > Q, id(P,S,Ps).

we find induces(ιsale, ιbelowzero). However, occurrence of a particular ιsale may actually induce or not an ιbelowzero internal event.

The set of "induces" facts that corresponds to figure 3 are:

| (in class PROD_STO) | (in class STORE) |
|---|---|
| induces(ιproduct,ιprod_sto) | induces(ιnew_store,ιstore) |
| induces(ιstore,ιprod_sto) | induces(ιnew_store,ιstonumber) |
| induces(ιprod_sto,ιsto_qoh) | induces(ιstore,ιcount_belowzero) |
| induces(ιsale,μsto_qoh) | induces(ιbelowzero,μcount_belowzero) |
| induces(ιreplenisment,μsto_qoh) | |
| induces(ιprod_sto,ιvalue) | |
| induces(μsto_qoh, μvalue) | |
| induces(μprodcost, μvalue) | |
| induces(ιsale, ιbelowzero) | |

Knowing that a particular event $e$ has been induced, the rules:

induced(X) ← induces(Y,X), induced(Y).
induced(*e*).

compute the events induced by $e$. In the example above, if we know that event ιnew_store has been induced, the rules give the events induced by it: ιnew_store, ιstore, ιstonumber, ιcount_belowzero, ιprod_sto, ιsto_qoh and ιvalue.

### 4.4 Object Interaction Model

Several object interaction models can be built for a particular system. We discuss some of them here, but others are possible.

In the first model, a central component, that we call the Events Manager, is responsible of all interactions. Its functions are:

1) To receive all external events produced at a given time. We allow the possibility of receiving several simultaneous events from the environment.

2) To send to event classes the external events just received, in order to check that they satisfy the constraints and to compute the derived attributes defined at those event classes.

3) To send to object classes relevant events for the production of new object instances, and to receive the insertion events that may be produced by them.

4) To send to object instances relevant events for the destruction of object instances or updating object attributes, and to receive the events that may be produced by them.

5) To send to object classes or instances relevant events for constraints checking or the generation of events, and to receive the events that may be produced by them.

In this model, a *method* must be defined for each relevant event of an object and event class. The functions of these methods are:

1) To receive the relevant event.

2) To produce the target events directly induced by the received event, thus creating or destroying object instances, updating object attributes, checking integrity constraints or generating events.

3) To send back to the Events Manager the set of target events produced.

We assume that other methods exist for accessing and returning attribute values. We will not consider them in our object interaction analysis, since they do not pose significant problems.

As an illustration of this model, we give below (and show in figure 4) an *object interaction pattern* that would evolve in the example of figure 3, when the external event tnew_store occurs:

1) The event tnew_store is received by the Events Manager.

2) The Events Manager sends the event tnew_store to a method of event class new_store. This method would check if the event satisfies the constraint store_exists (similar to prod_exists) and would computed attribute sto. For the sake of simplicity, in this paper we will not describe any further this kind of interaction between the Events Manager and the event classes.

3) The Events Manager sends the event tnew_store to method MS1 of class STORE. The method adds a new instance to class STORE and initialises attribute stonumber to 0, and returns events tstore and tstonumber.

4) The Events Manager sends the received event tstore to method MS2 of class STORE. The method initialises attribute count_belowzero to 0, and returns tcount_belowzero.

5) The Events Manager sends the event tstore to method MPS1 of class PROD_STO. The method adds a new instance to class PROD_STO for each existing product instance, and returns the events tprod_sto thus produced.

6) The Events Manager sends each tprod_sto event received to method MPS2 of class PROD_STO. The method initialises attributes sto_qoh and value to 0, and returns the events tsto_qoh and tvalue.

It is interesting to note that the above object interaction pattern is just one of the many valid patterns. Other patterns would produce the same overall effect.

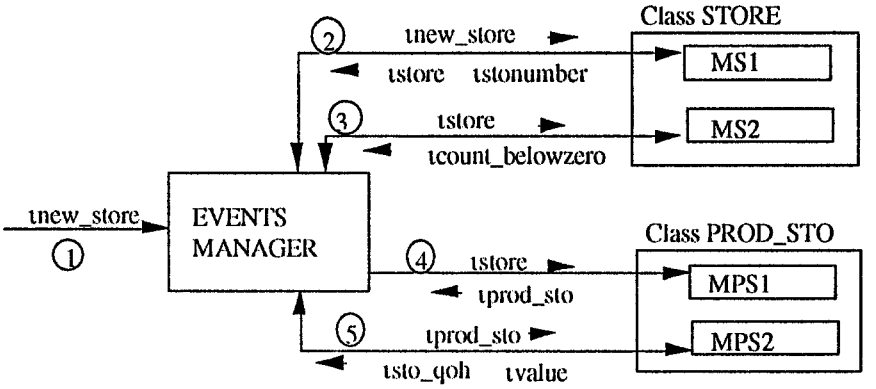



**Figure 4.** Example of object interaction pattern

In an alternative object interaction model, methods would produce all target events induced in a class, either directly or indirectly, by the received event. This model would reduce the need for object interaction, but then the methods become larger and less robust to changes (for instance, addition of a new object attribute might require the modification of existing methods). In some way, this is used by Oblog and TROLL ([SFS+89,SRG+91, JSH+91]).

Another object interaction model would eliminate the need of a central Events Manager, by allowing direct interaction between objects. However, methods then become still larger and even less robust to changes. Addition of new object classes, for instance, might require changing existing methods to reflect the need for interaction with the new objects. This model is used by most methods (see, for instance [Bai89,CoY91,Wie91,Prak92]).

In what follows, we outline a procedure for determining object interaction patterns according to the first model, but we could adapt it to other models (see [Que92]).

### 4.5 Valid Object Interaction Patterns

In our model, we call *object interaction pattern* (shortly, interaction) a sequence of messages sent by the Events Manager to methods in object classes, in response to one or more external events received from the environment.

We describe the sending of a message to a method of an object class by means of a predicate send(seqnumber,message,class,method), where the first argument is a sequence number. In this way, the interaction described above, for the external event tnew_store, would be defined by the sequence:

send(1,tnew_store,class_STORE,method_MS1)
send(2,tstore,class_STORE,method_MS2)
send(3,tstore,class_PROD_STO,method_MPS1)
send(4,tprod_sto,class_PROD_STO,method_MPS2)

The sequence number will also be used to distinguish between states in an interaction. The final state of an interaction will be the greatest sequence number (4 in the example above).

We can also define the simultaneous sending of one or more events to different methods. In this case, we would just assign the same sequence number to two or more "send" facts.

Before describing how such object interaction patterns can be generated, we consider the issue of validity of a given interaction. To this end, we need to introduce first the concept of "computed".

Rules in the internal events model state the events that must be sent to a class to compute a given target event. In general, however, a target event can have more than one rule (for instance, tprod_sto has two rules: rps.1, rps.2).

A target event e of class c defined by a rule r with relevant event $e_1$, is *partially computed* at some state S (denoted by pcomputed(S,e,r)) if:

a) $e_1$ has not been induced, or
b) $e_1$ has been induced and it has been sent at some state S1 to the corresponding method m (send(S1,$e_1$,c,m)), with S1 ≤ S.

This definition can be extended easily with more than one relevant event (see example below).

A target event e is *computed* at some state S (denoted computed(S,e)) if it has been partially computed with respect to all its rules:

computed(S,e) ← pcomputed(S,e,$r_1$),.....,pcomputed(S,e,$r_n$).

The following rules define the "computed" and "pcomputed" predicates for the target event tprod_sto of class PROD_STO:

pcomputed(S,tprod_sto,rps.1) ← induced(tproduct),
send(S1,tproduct,class_PROD_STO,method_MPS3), S1≤S.
pcomputed(S,tprod_sto,rps.1) ← not induced(tproduct).
pcomputed(S,tprod_sto,rps.2) ← induced(tstore),
send(S1, tstore, class_PROD_STO,method_MPS1), S1≤S.
pcomputed(S,tprod_sto,rps.2) ← not induced(tstore).

computed(S,tprod_sto) ← pcomputed(S,tprod_sto,rps.1), pcomputed(S,tprod_sto,rps.2).

We also define the "computed" concept for the existence of object instances. We say that computed(s,object_class) is true if the insertion and deletion internal events corresponding to the creation and destruction of the object instances have been computed at state s. Formally, if oc is an object class:

computed(S,oc) ← induced(ιoc), induced(δoc), computed(S,ιoc),computed(S,δoc).
computed(S,oc) ← induced(ιoc), not induced(δoc), computed(S,ιoc).
computed(S,oc) ← induced(δoc), not induced(ιoc), computed(S,δoc).
computed(S,oc) ← not induced(ιoc), not induced(δoc).

Note that the last rule defines that the object instances of oc are computed at any state of a transition if the ιoc and δoc have not been induced, since in such case the extension of oc is not changed.

The following rules define the "computed" and "pcomputed" predicates for the target event μsto_qoh of class PROD_STO:

pcomputed(S,μsto_qoh,rps.4) ← induced(ιsale),induced(ιreplenishment),
send(S1, ιsale, class_PROD_STO,method_MPS4), S1≤S,
send(S2, ιreplenishment, class_PROD_STO,method_MPS5), S2≤S.
pcomputed(S,μsto_qoh,rps.4) ← induced(ιsale), not induced(ιreplenishment),
send(S1, ιsale, class_PROD_STO,method_MPS4), S1≤S.
pcomputed(S,μsto_qoh,rps.4) ← not induced(ιsale), induced(ιreplenishment),
send(S1,ιreplenishment,class_PROD_STO,method_MPS5), S1≤S.
pcomputed(S,μsto_qoh,rps.4) ← not induced(ιsale), not induced(ιreplenishment).

computed(S,μsto_qoh) ← pcomputed(S,μsto_qoh,rps.4).

We also define the "computed" predicate for attributes as follows:

computed(S,att) ← induced(ιatt), induced(μatt), induced(δatt),
computed(S,ιatt),computed(S,μatt),computed(S,δatt).
computed(S,att) ← induced(ιatt), induced(μatt), not induced(δatt),
computed(S,ιatt),computed(S,μatt).
computed(S,att) ← induced(ιatt), not induced(μatt), induced(δatt),
computed(S,ιatt),computed(S,δatt).
computed(S,att) ← not induced(ιatt), induced(μatt), induced(δatt),
computed(S,μatt),computed(S,δatt).
computed(S,att) ← induced(ιatt),not induced(μatt), not induced(δatt),
computed(S,ιatt).
computed(S,att) ← not induced(ιatt), induced(μatt), not induced(δatt),
computed(S,μatt).
computed(S,att) ← not induced(ιatt), not induced(μatt), induced(δatt),
computed(S,δatt).
computed(S,att) ← not induced(ιatt), not induced(μatt), not induced(δatt).

In principle we could define the validity of an object interaction pattern I for a set E of external events, as:

$$\forall X \text{ (induced(X) } \rightarrow \text{ computed(fs,X))}$$

where fs is the final state, meaning that I is *valid* for E if all induced target events are computed by I. However, this does not take into account the *interaction constraints* that I must satisfy. These constraints, that can also be obtained from the internal events model, state the conditions that must hold for sending a message to a method.

For example, the following rules define the interaction constraints associated to the relevant events of target event tprod_sto of class PROD_STO:

ic(S) ← send(S1,tproduct, class_PROD_STO,method_MPS3), S1≤S,
S2 = S1 -1, not computed(S2,store).
ic(S) ← send(S1, tstore, class_PROD_STO,method_MPS1), S1≤S,
S2 = S1 -1, not computed(S2,product).

First constraint is violated if tproduct is sent to method MPS3 of class PROD_STO at some state S1, and store object instances had not been computed in the previous state. This is so because MPS3 requires to know which stores exist before creating new product_store instances. Recall that we allow the creation of both products and stores in the same transition (that is, we allow that one or more tnew_prod and tnew_store external events occur simultaneously). Second constraint is similar.

Having defined the "induced" and "computed" concepts, and the constraints that an interaction must to fulfil, we say that an object interaction pattern I is valid for a set of external events E if all events induced by E are computed by I and no one ic is violated.

$$\forall X \text{ (induced(X) } \rightarrow \text{ computed(fs,X),not ic(fs))}$$

### 4.6 Generation of Object Interaction Patterns

Generation of object interaction patterns consists in obtaining the messages to be sent by the Events Manager to methods in object classes, in response to the external events received at a given time by the system. The interaction generated is different depending on the external events received.

We outline in this section our procedure for the automatic generation of interaction, which is based on plan generation techniques.

Let E be a set of external events, that the Events Manager can receive at the same point of time. If we know E in advance the interaction can be generated at development-time. On the contrary, it must be generated by the Events Manager at execution-time.

The procedure uses the computed, pcomputed and ic predicates and their definition, presented in last section. Recall that these rules can be derived automatically from the internal events model.

Let us denote by U the set of events induced by E.

Let C be the set of target events already computed at some stage of our procedure. At the beginning, C consists of the elements of set E. Let I be the set of messages to be sent by the Events Manager. Initially, $I = \emptyset$. Our procedure consists of 6 steps:

1) Select an element *elem* from set C.

2) Select a set R of rules having a send(S,*elem*,class,method) literal in their body with the same class and method.

3) Extract from set R the subset Q of relevant rules. A rule in R is relevant if:

- for all its literals induced(*elem1*) appearing in its body, *elem1* is in U.
- for all its literals not induced(*elem1*) appearing in its body,*elem1* is not in U.

4) Check that all rules in set Q are applicable. All rules in Q are applicable if there is not any constraint violated by $I \cup$ send(S,*elem*,class,method).

5) If all rules in Q are applicable, then:

- Add the send(S,*elem*,class,method) literal to I. Assign to the sequence number of this newly added "send" literal the current state. Add 1 to the current state.
- Add to C the *elem2* appearing as pcomputed(S,*elem2*,R) in the heads of each rule Q, only if *elem2* is totally computed at the current state.

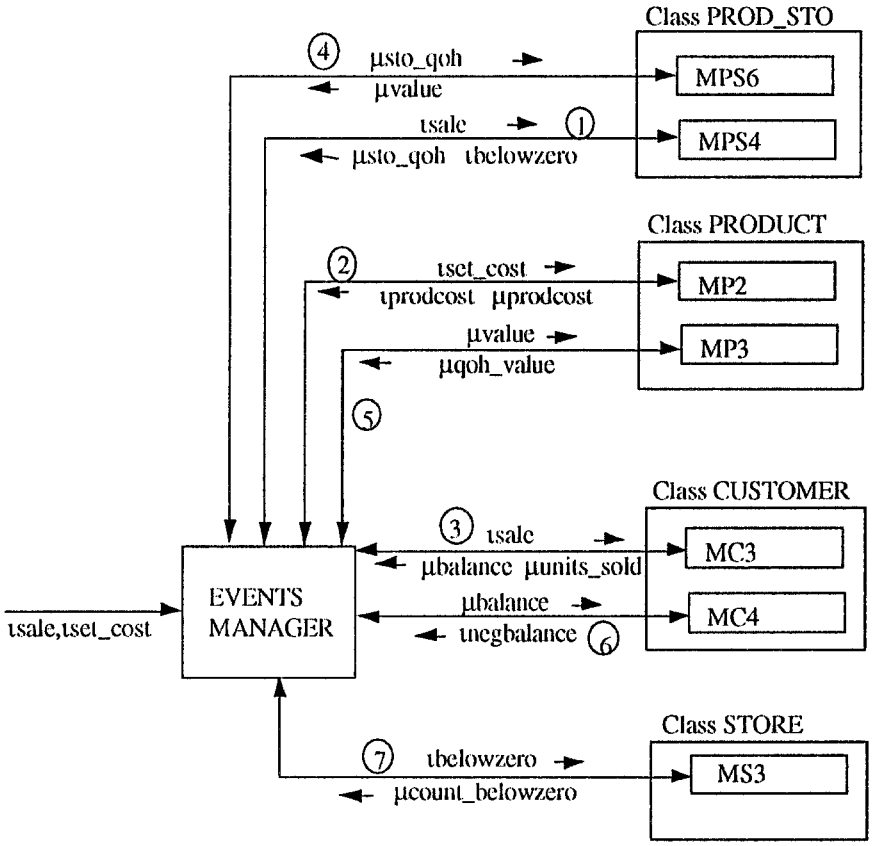
6) The procedure finishes as soon as U is a subset of C.

We illustrate the procedure by means of an example. Assume the set of external events received by the Events Manager is E={ιsale,ιset_cost}.

One solution of the generation procedure of object interaction patterns is:

I = { send(1,ιsale,PROD_STO,method_MPS4),
send(2, ιset_cost,PRODUCT,method_MP2),
send(3, ιsale,CUSTOMER,method_MC3),
send(4, μsto_qoh,PROD_STO,method_MPS6),
send(5, μvalue,PRODUCT,method_MP3),
send(6, μbalance,CUSTOMER,method_MC4),
send(7, ιbelowzero,STORE,method_MS3)}

Fig. 5 shows graphically and explain the messages sent by the Events Manager to the objects.




1) The Events Manager sends the event ιsale to method MPS4 of class PROD_STO. The method modifies the sto_qoh value, and produce the generated event ιbelowzero, that is returned with the event μsto_qoh.

2) The Events Manager sends the event ιset_cost to method MP2 of class PRODUCT. The method initialises or modify the value of attribute prodcost, and returns the events ιprodcost, μprodcost.

3) The Events Manager sends the event ιsale to method MC3 of class CUSTOMER. The method modifies the values of attributes balance and units_sold, and returns the events μbalance, μunits_sold.

4) The Events Manager sends the event μsto_qoh to method MPS6 of class PROD_STO. The method modifies the value of attribute value, and returns the event μvalue.

5) The Events Manager sends the event μvalue to method MP3 of class PRODUCT. The method modifies the value of attribute qoh_value, and returns the event μqoh_value.

6) The Events Manager sends the event μbalance to method MC4 of class CUSTOMER. The method checks the negbalance constraint, and returns ιnegbalance if violated.

7) The Events Manager sends the event ιbelowzero to method MS3 of class STORE. The method modifies the value of attribute count_belowzero, and returns the event μcount_belowzero.

**Figure 5.** Object Interaction pattern example

We now explain in detail how the solution is obtained. The events induced by E are:

U= {ιsale,ιset_cost,ιbelowzero,ιprod_cost,μsto_qoh,μunits_sold,
μbalance,μvalue,ιnegbalance,μprodcost,μcount_belowzero,μqoh_value}

The set of computed target events, and the set of send literals, at the beginning of the process are:

$C_0$ = {ιsale,ιset_cost}
$I_0$ = {}

1) We select an element of $C_0$, *elem* = ιsale.

2) We select the set R from rules associated to target events of class PROD_STO:

pcomputed(S,μsto_qoh,rps.4) ← induced(ιsale),not induced(ιreplenishment),
send(S1, ιsale, class_PROD_STO,method_MPS4), S1≤S. (CR2)
pcomputed(S,μsto_qoh,rps.4) ← induced(ιsale), induced(ιreplenishment),
send(S1, ιsale, class_PROD_STO,method_MPS4), S1≤S, S2≤S.
send(S2, ιreplenishment, class_PROD_STO,method_MPS5). (CR4)
pcomputed(S,ιbelowzero,rps.8) ← induced(ιsale)
send(S1,ιsale,class_PROD_STO,method_MPS4), S1≤S. (CR5)

3) We extract from R the subset of relevant rules

Q = { CR2,CR5 }

4) We check if rules in Q are applicable.They are applicable because there is not any ic violated by $I_0 \cup$ send(S, ιsale, class_PROD_STO,method_MPS4).

5) We add send(S, ιsale, class_PROD_STO,method_MPS4) literal to $I_0$, assigning the current state to the sequence number of this literal:

$I_1$ = {send(1,ιsale,PROD_STO,method_MPS4)}

We add to $C_0$ the events that will be totally computed once the ιsale message has been sent to method MPS4 of class PROD_STO. We know which are these events by the following computed rules:

computed(S,μsto_qoh) ← pcomputed(S,μsto_qoh,rps.4)
computed(S,ιbelowzero) ← pcomputed(S,ιbelowzero,rps.8)

So, the new set of computed target events is:

$C_1$ = {ιsale,ιset_cost,μsto_qoh,ιbelowzero}

6) Given that U is not a subset of C yet, then it is necessary to repeat the process.

We outline the next iterations in the following. Eight iterations are needed to arrive to the object interaction pattern example:

$C_1$ = {ι sale,ι set_cost,μsto_qoh,ι belowzero}
$I_1$ = {send(1,ι sale,PROD_STO,method_MPS4)}
|
*Iter. 2* | *elem* = μsto_qoh, rules of class PROD_STO ==> not applicable
|
$C_2 = C_1$
$I_2 = I_1$
|
*Iter. 3* | *elem* = ι set_cost rules of class PRODUCT
|
$C_3$ = {ι sale,ι set_cost,μsto_qoh,ι belowzero,ι prodcost,μprodcost}
$I_3 = I_2 \cup$ {send(2, ι set_cost,PRODUCT,method_MP2)}
|
*Iter. 4* | *elem* = ι sale rules of class CUSTOMER
|
$C_4$ = {ι sale,ι set_cost,μsto_qoh,ι belowzero,ι prodcost,μprodcost,μbalance,
μunits_sold}
$I_4 = I_3 \cup$ {send(3, ι sale,CUSTOMER,method_MC3)}
|
*Iter. 5* | *elem* = μsto_qoh rules of class PROD_STO
|
$C_5$ = {ι sale,ι set_cost,μsto_qoh,ι belowzero,ι prodcost,μprodcost,μbalance,
μunits_sold,μvalue}
$I_5 = I_4 \cup$ {send(4, μsto_qoh,PROD_STO,method_MPS3)}
|
*It.6,7,8* ....
|
$C_8$ = {ι sale,ι set_cost,μsto_qoh,ι belowzero,ι prodcost,μprodcost,μbalance,
μunits_sold,μvalue,μqoh_value,ι negbalance,μcount_belowzero}
$I_8$ = { send(1,ι sale,PROD_COST,method_MPS4),
send(2, ι set_cost,PRODUCT,method_MP2),
send(3, ι sale,CUSTOMER,method_MC3),
send(4, μsto_qoh,PROD_STO,method_MPS6),
send(5,μvalue,PRODUCT,method_MP3),
send(6, μbalance,CUSTOMER,method_MC4),
send(7,ι belowzero,STORE,method_MS3)}

At the end of the 8th iteration,we have in C all the target events that can be induced (U is a subset of C), and the procedure finishes.

In all iterations the element selected from C to be sent to a method of a class did not violated any interaction constraint, except in *Iter.2* where the purpose was to modify the value of the qoh of products in a store, when the cost of the products was not already modified (having a set_cost event in set E). In that case the following ic were violated by the selected event μsto_qoh:

$$ic(S) \leftarrow send(S1, \mu sto\_qoh, class\_PROD\_STO, method\_MPS6),\ S1 \leq S,$$
$$S2 = S1 - 1,\ not\ computed(S2, prodcost).$$

There is more than one valid object interaction pattern for the same set of external events. We obtain different interactions depending on how the elements in set C are selected in step 1 of the procedure.

For example, another valid solution given by our procedure, would be:

$$I = \{\ send(1, \iota set\_cost, PRODUCT, method\_MP2),$$
$$send(2, \iota sale, CUSTOMER, method\_MC3),$$
$$send(3, \iota sale, PROD\_STO, method\_MPS4),$$
$$send(4, \mu balance, CUSTOMER, method\_MC4),$$
$$send(5, \iota belowzero, STORE, method\_MS3)$$
$$send(6, \mu sto\_qoh, PROD\_STO, method\_MPS6),$$
$$send(7, \mu value, PRODUCT, method\_MP3)\}$$

## 5 Discussion

The results from the previous section show that a procedure exists that determines the object interactions implied by a given ODCM. We can then pose the question whether explicit object interaction, as present in most current object-oriented methods, is a desirable feature for conceptual models of information systems. We believe that a definite answer to that question is not possible yet. However, we hope that the following comments might provide some elements for that answer (see also [HeE90,EFG+91,dCP92]).

Our procedure shows that object interaction relies on some object interaction model. We have seen in section 4.4 that several models are possible, and that object interactions are quite different from a model to another. On the other hand, changing from a model to another is a difficult task. Such a task might be necessary if the object interaction model used at the conceptual level is different from the one used at the design level.

Our procedure also shows that several, equivalent object interaction patterns may exist for the same input external events. Explicit object interaction modelling forces to choose arbitrarily one of them at the conceptual level.

It can also be shown that object-oriented models using direct inter-object communication are very sensitive to changes. Additions of new object classes or attributes may imply changes in existing methods. In our example, assume the case when a single-instance object class COMPANY is added to the model, with attribute global_qoh_value. Then, methods at the product_store or product level should be changed in the way that they now communicate to that new object the changes on the qoh_values.

Some object-oriented methods allow defining derived attributes, that is, attributes whose value may be computed from the values of other attributes of the same or different object. In these cases, the procedure for evaluating the attributes may have an impact on object interactions. If, at the design level, the value of the attributes is computed whenever they are requested, then no object interaction is necessary: the object method evaluates the defining rule and returns the result. However, if the value for that attribute is made explicit in the object local state, then the need arises for new interaction: the object must know changes on the values of the terms in the defining rule.

## 6 Conclusions

We have described the main components of an object-oriented, deductive approach (ODCM) to conceptual modelling of information systems. Basic concepts of this approach are event and object classes. Event classes have attributes and constraints. Object classes have attributes, constraints and generated events. Rules specified at the class level relate object existence, and attribute values to events. Object interaction is not made explicit.

We have then presented a formal procedure, based on the concepts of internal event and internal events model, that produces the object interaction patterns implied by an ODCM. We have also formalized the concept of valid interaction.

We have discussed the issue of whether explicit object interaction is a desirable feature of conceptual models. Some (hopefully) relevant points to that issue have been made, based on the above results.

We plan to implement the object interaction patterns generation procedure for several object interaction models, and the automatic generation of methods.

## Acknowledgements

We would like to thank Dolors Costal, Carme Martin, Enric Mayol, Joan Antoni Pastor, Maria Ribera Sancho, Jaume Sistac, Ernest Teniente and Toni Urpí for many useful discussions. We are also grateful to the referees for their interesting comments and suggestions.

This work has been partially supported by the CICYT PRONTIC program project TIC 680.

## References


[Bai89] Bailin, S.C. "An object-oriented requirements specification method", Comm. of the ACM, Vol.32, No.5, May 1989, pp. 608-623.

[CoY91] Coad,P.; Yourdon,E. "Object-oriented analysis". Yourdon Press,1991.

[dCP92] de Champeaux,D.; Faure,P. "A comparative study of object-oriented analysis methods". Journal of Object-Oriented Programming, March-April 1992, pp.21-23.

[EFG+91] Estier,Th.; Falquet,G.; Guyot,J.; Leonard,M. "Six spaces for global information systems design". In [VMR91], pp.343-359.

[FaL89] Falkenberg,E.D.;Lindgreen,P. (Eds.). "Information system concepts: An in-depth analysis", North-Holland, 1989.

[GKB82] Gustafsson,M.R.;Karlsson,T.;Bubenko,J. "A declarative approach to conceptual information modeling". In Olle,T.W.; Sol,H.G.;Verrijn-Stuart,A.A. (Eds.) "Information systems design methodologies: A comparative review", North-Holland, 1982, pp. 93-143.

[HeE90] Henderson-Sellers,B.;Edwards,J. "The object-oriented systems life cycle". Comm. of the ACM, Vol.33, No.9, September 1990, pp. 143-159.

[JaR84] Jardine,D.A.;Reuber,A.R. "Information semantics and the conceptual schema". Information Systems, Vol.9,No.2,1984, pp. 147-156.

[JSH+91] Jungclaus,R.; Saake,G.; Hartmann,T.; Sernadas,C. "Object-Oriented Specification of Information Systems: The TROLL Language". Technische Universität Braunschweig. Informatik-berichte, 91-04.

[MSS92] Mayol,E.; Sancho,M.R.; Sistac,J. "The ODISSEA project: an environment for development of Information Systems". Proceedings of the third International Workshop on the Deductive Approach to IS and DB. Roses. Sept, 1992. pp.127-156.

[Oli86] Olivé, A. "A comparison of the operational and deductive approaches to conceptual information systems modelling". In "Information Processing 86", Elsevier, 1986, pp. 91-96.

[Oli89] Olivé, A. "On the design and implementation of information systems from deductive conceptual models". Proc. of the 15th. VLDB, pp. 3-11.

[Prak92] Prakash,N. "An object-oriented methodology for information systems design". In Falkenberg,E.D.,Rolland,C.,El-Sayed,E.N. (Eds). "Information system concepts: improving the understanding", North-Holland, 1992, pp.87-122.

[Que92] Quer,C. "Combining the object-oriented approach and the deductive approach for conceptual modelling". Proceedings of the third International Workshop on the Deductive Approach to IS and DB. Roses. Sept, 1992. pp.127-156.

[San90] Sancho, M.R. "Deriving an internal events model from a deductive conceptual model". Proc. Intl. Workshop on the deductive approach to IS and DB, S'Agaró, 1990, pp. 73-92.

[SFS+89] Sernadas,A.;Fiadeiro,J,;Sernadas,C.;Ehrich,H-D. "The basic building blocks of information systems". In [FaL89], pp. 225-246.

[SRG+91] Sernadas,C.;Resende,P.;Gouveia,P.;Sernadas,A. "In-the-large object-oriented design of information systems", in [VMR91], pp. 209-232.

[Wie91] Wieringa,R.J. "Object-oriented analysis, structured analysis and Jackson system development", in [VMR91], pp. 1-21.

[VMR91] Van Assche,F.;Moulin,B.;Rolland,C. (Eds.) "Object oriented approach in information systems". North-Holland, 1991.