ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Rainer Manthey and Viacheslav Wolfengagen (Eds)

Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

The Cause-Effect Rules of ROSES

D.Costal, M.R.Sancho, A.Olivé, M.Barceló, P.Costa, C.Quer,
and A.Roselló

Published in collaboration with the
British Computer Society

The Cause-Effect Rules of ROSES

D.Costal, M.R.Sancho, A.Olivé, M.Barceló, P.Costa, C.Quer, A.Roselló

Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya

Jordi Girona Salgado 1-3, Mod. C6. E08034, Barcelona, Catalonia

Abstract

This paper presents a new behavioural specification approach for conceptual modelling of Information Systems. We call it the Cause-effect rule approach because the specification consists of a number of independent rules, each of which establishing a relationship between a cause and an effect.

1 Introduction

This paper presents a new behavioural specification approach for conceptual modelling of Information Systems (IS). We call it the Cause-effect rule approach because the specification consists of a number of independent rules, each of which establishing a relationship between a cause and an effect. Such rules are common in the active database systems field [18], but their potential use in conceptual modelling of IS has not been explored in depth.

Our approach has been developed in the context of the ongoing ROSES (Rules, ObjectS and EventS) project, whose aim is, among others, to explore the application of cause- effect rules in the formal specification of the behaviour of IS. However, we believe that the approach is general, and that it might inspire some ideas helpful in other contexts or languages. In some respects, our work has some points in common with the work reported in [9, 12], and some similarity with the Chimera language [3], as will be described later.

We review, in the next section, the components of a conceptual model and characterise the main existing approaches with respect to the behavioural model. Sections 3 to 6 describe our approach, with emphasis on the behavioural model, and section 7 gives some elements for a comparison between our approach and the most popular ones.

2 Conceptual models

An information system performs two main functions: To maintain a consistent representation of the state of a given domain, and to produce reactions when certain predefined conditions are satisfied. The former is usually referred to as the passive function, since the system merely keeps track of the evolving state of the domain, while the latter is usually called the active function, since the system performs actions that influence the domain. The active function includes, among others, the response to queries on the state of the domain, and the generation of events that change that state (such as, for example, to issue automatically an order when the quantity on hand of a product is below the reorder point).

A conceptual model of an information system describes [16]:

- the knowledge the system has to maintain about the state of the domain (Information Base, IB),
- the possible evolution of the state due to the occurrence of (external or generated) events or to the passing of time,
- the events that must be generated when some conditions are satisfied, and

- the response to the queries issued by the users, or the output (if any) to be produced when an event is detected.

A conceptual model consists of two (sub)models: The *structural* and the *behavioural* model. The first describes:

- the object (or entity) types characterising the objects in the domain.
- the structural relationships between the above types, mainly generalization or specialization.
- the object attributes and relationships.
- the derivation rules (if any) defining the population of derived object types, the values of derived attributes, or the derived relationships between objects.
- the static and transition integrity constraints.

This paper focuses mainly on the behavioural specification. A behavioural model describes:

- the event types, and their parameters. Sometimes we call Event Model to the list of the event types, with their respective parameters. We distinguish among three kinds of events:
 - (1) *External*, which are those communicated to the IS by the environment, through the I/O interface;
 - (2) *Generated*, which are those events meaningful in the domain but detected by the IS and such that, when detected, some action must be performed by the IS itself on behalf of the domain; and
 - (3) *Internal*, which are also detected by the IS, but only for internal purposes of the behavioural model.
- the conditions defining the occurrence of generated and internal events.
- the integrity constraints associated with the events, that is, those constraints that cannot be defined in the structural model.
- the effect of the events on the IB.

Usually, a behaviour model is very large, and it becomes necessary to structure the model in some way, in order to make it manageable. This is achieved by means of a *structuring concept*. The idea is to structure the behaviour model in a number of specification units (or, for short, units), one for each instance of the structuring concept, and in such a way that the overall behaviour model is given by the set of its units.

The most widely used structuring concept is the **event type**. We call event-centred the approach to behaviour modelling that uses this structuring concept. In this approach, there is a unit (usually called transaction) for each external or generated event type. A unit of this kind describes the conditions that must be satisfied by the instances of its associated event type, and the changes induced in the Information Base (or the events generated) by the occurrence of such events [17].

Some recent object-oriented conceptual modelling languages centre the behavioural specification on **object types** [7]. For each object type, there is a specification unit that defines the possible states of the objects, the events that are relevant to objects of that type, with their precondition, and the effects of these events on the population of the object type, and on the attributes and relationships of its objects.

We can distinguish here two main subapproaches, depending on how objects interact. In the first, object interaction is *implicit*. All events are in principle available to all objects, and each object must determine whether or not it is interested in a particular event and, if so, take the appropriate action. On the other hand, objects may generate events, which become also available to all objects. The Syntropy method takes this approach [5].

In the second subapproach, object interaction is *explicit*. Each event is sent to a particular object, or to a set of objects, and if some object receives an event which may also be of interest to another object then it has to "forward" the received event to this latter object. Event generation is handled in the same way, by sending it to one of the interested objects. Most O-O methods take this approach. One recent example is UML [15].

In this paper, we present a new approach, in which the structuring concept is **the cause-effect rule**. Before describing it, we need to introduce some relevant concepts of the structural model and the events. This is done in the following sections. The cause-effect rules are then described in section 6.

3 Structural models in ROSES

The structural modelling concepts of ROSES are based on the well-known concepts of objects, partitions, attributes, derivation rules and constraints [14, 11, 15]. On the other hand, our language adopts the temporal approach, in which object existence and attribute values depend on time [8, 13].

At the structural level, ROSES provides a number of new temporal features (related to class populations and to attributes) that allow to define, in a simple way, common dynamic (or temporal) constraints. As we will see in the next section, these features have a significant impact on the possible structural events.

For the purposes of this paper, it is not necessary to give here the formal and complete details of the structural model. We will show instead an annotated example, introducing the main features that will be used in the rest of the paper. The example deals with persons and their marital status. The main object class could be person:

```
abstract object class person with
  attributes
    name: string(30);
    parents: set of person;
  population features
    permanent instances;
    always initially member;
    single permanent membership interval;
  attribute features
    name: always initially valued;
        single permanent existence interval;
        non-modifiable;
    parents: always initially valued;
        insertions not allowed;
        deletions not allowed;
        size between 2 and 2;
end
```

Here we define that person is an object class with three population features: *Permanent instances* means that once a person is created, he exists until the end of the system's lifespan; *Always initially member* means that all persons must be instance of this class when they are created (other options are [never|sometimes] initially member); and *Single permanent membership interval* means that a person will remain as member of this class until the end of the person's lifespan (other options are [single non-permanent |multiple] membership intervals).

We have defined that name is a single-valued attribute of persons, with three attribute features: *Always initially valued* means that when a person is created he must have a name (other options are [never|sometimes] initially valued); *Single permanent existence interval* means that once a person has a name he must have always a name (other options are [single non-permanent |multiple] existence intervals); and *Non-modifiable* means that a person's name cannot be changed (opposite is modifiable).

We have also defined that parents is a multivalued attribute, with three attribute features: *Always initially valued* means that parents must be known when the person is created. *Insertions not allowed* means that once a person has been created it is not possible to add new parents to him (opposite is insertions allowed). *Deletions not allowed* means that it not possible to delete parents of a person (opposite is deletions allowed).

In the general case, generalization and specialization is defined through partitions:

```
complete object partition male_female of person
end
base object class man in male_female
```

```

end
base object class woman in male_female
end

```

Note that man and woman are defined as base classes. Objects are created and destroyed only through structural events (see next section) defined in base classes.

A given object class can be partitioned according to several criteria. The next sentences define two subclasses of persons according to their existence status:

```

complete object partition existence of person
end
event-selected object class alivePerson in
  existence with
  attributes
    children: set of person derived;
  population features
    always initially member;
    single non-permanent existence interval;
  attribute rules
    Self.children(at T)=X if
      alivePerson(Self)(at T),
      X=(Y where person(Y)(at T),
        Self in Y.parents(at T))
end
event-selected object class deadPerson in
  existence with
  population features
    never initially member;
    single permanent existence interval;
end

```

In general, objects of event-selected classes are explicitly inserted into, and removed from, them by means of special structural events. The population feature `single non-permanent existence interval` of `alivePerson` means that a person may leave this class (while still being a person).

Note the definition of a derived attribute in class `alivePerson`. The derivation rule states that, at time T , the set X of children of an alive person `Self` is the set of persons Y such that `Self` is one of his parents at time T .

To complete the example, we give below a specialization of `alivePerson`:

```

complete object partition marital_status of
  alivePerson
end
event-selected object class single in
  marital_status with
  population features
    always initially member;
    single non-permanent existence interval;
end
event-selected object class marriedPerson in
  marital_status with
  attributes
    spouse: marriedPerson;

```

```

    attribute features
      spouse: always initially valued,
             single permanent existence interval;
             non-modifiable;
    end
  event-selected object class divorced in
    marital_status
  end
  event-selected object class widowed in
    marital_status
  end
end

```

On the other hand, we may define *generated* events in the structural model. Such events are generated according to a rule defined in some object class. Typically, they are used to signal an state of the IB for which some action is required.

4 Structural events

Conceptual modelling languages provide (either implicitly or explicitly) a set of structural events based on the structural model. Such events allow common changes such as inserting or deleting an object instance, changing the value of attributes, and so on. How such events are invoked depends on the specific approach or language used.

In ROSES we have elaborated a little bit more the concept of structural event. We take into account the structure of object classes, and the set of population and attribute temporal features (as well as the common static features) and determine **automatically** the set of possible structural events. Each structural event has associated a set of constraints and effects on the Information Base. For example, insertions of men can only be done through the event:

```

structural event class insert_man with
  attributes
    name: string(30);
    parents: set of person;
    time: time;
  end

```

with the following associated constraints and effects:

- The name cannot be nil.
- A new man is created, with the given attributes. The man becomes also an instance of class `person`.
- The man is made instance of class `alivePerson` (because we have `always initially member` in this class)
- The man is made instance of class `single` (for a similar reason).

All events include a `time` attribute, which gives their occurrence time.

It is not possible to delete persons (men or women) since we have defined their classes with the population feature `permanent instances`. The rest of the structural events corresponding to the object model given in the previous Section would be:

```

structural event class insert_woman with
  attributes
    name: string(30);

```

```

        parents: set of person;
        time: time;
end
structural event class insert_deadPerson with
    attributes
        person: person;
        time: time;
end
structural event class insert_marriedPerson with
    attributes
        person: alivePerson;
        spouse: person;
        time: time;
end
structural event class insert_divorced with
    attributes
        person: alivePerson;
        time: time;
end
structural event class insert_widowed with
    attributes
        person: alivePerson;
        time: time;
end

```

Note that there are not structural events allowing to change attributes of persons. This is due to the attribute features defined for those attributes.

5 External and internal events

ROSES provides a rich set of language constructs to define external events, including generalization/ specialization hierarchies, simple/complex events, derived attributes and constraints. However, we have found that very often our structural events themselves correspond (including their associated constraints and effects) to external events and, thus, they need not to be defined explicitly as such. For instance, the above `insert_man` structural event may be used as external event to communicate the birth of a man.

In other cases, the external events do not correspond to one of the structural events, and must be defined explicitly. For instance, marriages can be communicated with the external event:

```

base external event class marriage with
    attributes
        bridegroom: man;
        bride: woman;
        time: time;
    constraints
        marriedAlready(person:P,time:T) if
            marriage(Self),Self.time=T,
            (Self.bridegroom=P or Self.bride=P),
            marriedPerson(P)(at previous(T));
end

```

The constraint states that a marriage `Self`, with occurrence time `T`, violates constraint named `married-Already` if the bridegroom (or the bride) of `Self` is `P`, and `P` is a `marriedPerson` at `previous(T)`.

Similarly, divorces can be communicated with the external event:

```
base external event class divorce with
  attributes
    person: person;
    time: time;
end
```

In some complex cases, it may be convenient to define *internal* events, which lie between external and structural events. Their only purpose is to provide a way to give a structure to event rules.

6 Behavioural models in ROSES

We need now to establish the relationship between external and structural events. This is done by means of event rules. The trivial case is when a structural event is used as external event. In this case, the rule is implicit: when the external event occurs, the corresponding structural event is induced and, if its associated integrity constraints are satisfied, the IB is modified according to its associated effects. In the example, this case happens with structural events `insert_man`, `insert_woman` and `insert_deadPerson`.

In general, an event rule has the form:

```
rule ruleName
  event_1 (att_1:X,...,att_n:Y,time:T) if
    event_2 (Z), Z.time = T, F;
  [duplicates [non-]allowed];
  [before|after] ruleName;
end
....
```

where:

- `event_1` is a (structural, internal or external) event with attributes `att_1, ..., att_n` and time.
- `event_2` is a (external, internal, generated or structural) event, and
- `F` is a formula with at least variables `X, ..., Y`.

The meaning of the above rule is: Each occurrence `Z` of `event_2`, with occurrence time `T`, if formula `F` is true, induces an occurrence of `event_1` with attribute values given by `Z, T` and variables in `F`.

In the rule, `event_2` is the triggering event, and `event_1` is the induced (or triggered) event. We also say that, in the rule, the triggering event is the cause, and that the induced event is the effect. When the effect is a structural event, its induction implies that the associated effect is performed on the Information Base. An event may be a triggering event in several rules and, on the other hand, an event may be a triggered event in several rules.

For example, the rule:

```
rule marriage_1
  insert_marriedPerson (person:P,spouse:S,time:T)
  if
    marriage (M), M.time = T,
    M.bridegroom = P, M.bride = S;
end
```


declares that the occurrence of a marriage (external event) induces an occurrence of `insert_marriedPerson` (structural event). In this case, formula `F` only serves to give the values of attributes `person` and `spouse`. In other cases `F` includes references to objects.

Before continuing the explanation of the rules, it may be interesting to show the strong similarity of our rules with logic programming rules [10]. In particular, our rules, like logic programming rules, can be read in two directions, allowing a double interpretation. From right to left they give a procedural view: when the triggering event occurs, if `F` is true then induce the triggered event. From left to right, the rule gives a declarative view: the rule head's event is induced if the triggering event occurs and `F` is true.

There is also a strong similarity of our rules with ECA rules in active database systems, which allows us to use in our context the rule analysis (termination, confluence, determinism) algorithms developed in that field [6]. The main difference lies in the action part: we only allow a single action (induced event), and the action is always an event (which, if it is a structural one, has an implicit effect on the IB). On the other hand,

due to our temporal approach, in the condition part we allow access to any past state of the IB and to any prior event.

In particular, ROSES shares several points with the Chimera language [3], including at least an object orientation for the structural model, derivation rules, a declarative specification of object constraints and the use of active rules. Furthermore, we have adopted elements of the Chimera syntax. However, there are also important differences, originated from the fact that Chimera is design-oriented, while ROSES is oriented towards conceptual modelling. Some of the differences are:

- Chimera is not a temporal model (although some work has been done in providing a temporal extension [2]).
- Chimera does not provide an adequate (from the conceptual modelling point of view) treatment of events.
- Chimera active rules allow a complex action part. In ROSES, the action is a single induced event, and the event rules are very similar to derivation and constraints rules.

If two or more rules have the same triggering event, they can be combined into a single rule. This must be understood as a syntactic simplification only, with no semantic effect. For example:

```
rule marriage
  insert_marriedPerson(person:P,spouse:S,time:T),
  insert_marriedPerson(person:S,spouse:P,time:T)
  if
    marriage (M), M.time = T,
    M.bridegroom = P,M.bride = S;
end
rule divorce
  insert_divorced(person:P,time:T),
  insert_divorced(person:S,time:T)
  if
    divorce (D),D.time = T,
    D.person = P,P.spouse(at previous(T)) = S;
end
```

The last rule of our example, shows a structural event that acts as a triggering event:

```
rule becomingWidowed
  insert_widowed(person:S,time:T) if
    insert_deadPerson (D),D.time = T,
    D.person = P,P.spouse(at previous (T)) = S;
end
```

which defines that the death of person P induces the structural event insert_widowed of person S if S was P's spouse at the time previous to death's occurrence.

The options `duplicates [non-]allowed` can be used when, for a given triggering event, formula F can instantiate in two or more different ways variables X,...,Y - Z,T. In this case, the option `duplicates allowed` means that there must be an induced event for each instantiation, while `duplicates non-allowed` would give a unique induced event.

For example, suppose that sometimes users wish to send letters to persons that have married during the current year. We would define a new external event:

```
base external event sendLetters with
  attributes
    time:time
end
```

that will be used at any time to tell the system to send the letters. Assume also that the sending of the letters will be associated with the internal event:

```
base internal event sendTo with
  attributes
    name: string(30);
    time:time;
end
```

The event rule could be:

```
rule sendingOfLetters
  sendTo (name:N,time:T)
  if
    sendLetters (S),S.time=T,
    marriage(M),M.time=T1,
    year(T)=year(T1),
    (M.bridegroom=P or M.bride=P),
    P.name=N;
  duplicates non-allowed;
end
```

where the option `duplicates non-allowed` prevents the system to send two or more letters to persons that have married two or more times during the year. Note in this case that, with our temporal approach, we can refer to marriage external events that occurred in the past. The expression:

```
marriage(M),M.time=T1,year(T)=year(T1)
```

is satisfied by all marriages M such that the year of their occurrence time is year(T).

The options `[before|after] ruleName` can be used to establish a priority in rule evaluation when several orders are possible.

Practical experience shows that the modelling constructs:

- object classes hierarchies, and derived object classes,
- external events hierarchies, and derived external events, and
- event rules, with external events as triggers and structural events as induced events,

are sufficient to model behaviour of most IS. In particular, observe that a single external event occurrence may belong to several external event classes and, thus, it may trigger simultaneously several rules. For example, assume that:

```
abstract external event class changeOfMaritalStatus
...
end
```

is a generalization of events marriage and divorce, and that repetition is a derived external event subclass of marriage defined as:

```
...
    population rule
        repetition (Self) if
            marriage(Self), Self.time=T,
            M.bridegroom=M, M.bride=W,
            marriedPerson(M)(at T1), T1<T,
            M.spouse(at T1)=W;
...

```

Then, an occurrence of marriage is also a changeOfMaritalStatus event and, if the two involved persons were already married in the past, it is also an occurrence of repetition event. In this case, a single occurrence of a marriage would trigger all rules having as triggering event one of these three events.

However, in some cases, one would like to have a structuring mechanism of event rules, specially when the number of such rules is large [1]. In ROSES, three of such mechanisms are possible:

- Grouping the rules with the same induced event. This may be facilitated by a simple CASE tool.
- Grouping the rules with the same triggering event. This can also be facilitated by a simple CASE tool.
- Using internal events. One or more rules may induce, in different situations, a given internal event, which in turn can trigger one or more different rules. This mechanism provides a stratification of the event rules.

7 Analysis

In the previous sections we have presented the cause-effect rule approach of ROSES to behaviour modelling. We would like now to compare it with respect to some other approaches. The key idea is to consider that each approach provides a concept to structure the (usually large) behavioural model.

A comprehensive comparison of all approaches would require an in depth analysis of the effort required to construct, understand and change the corresponding specification units in all approaches. Such an analysis is beyond the scope of this paper. Here, what we suggest instead is to analyse the units in each of the approaches with respect to three simple properties: number, coupling and cohesion. The importance of these properties may be arguable, but at least they may help to understand better each approach. In what follows, we summarise the preliminary conclusions of our analysis.

The first property is the number of units obtained in each approach. By itself, this property has a limited interest. But, if we consider that the global "functionality" of a behaviour model is distributed, without too much redundancy, across the units, then the inverse of that property gives us the average size of functionality per unit. This property could be more interesting. At least, in the Software Engineering field, it is a well established principle that, if the units (modules) of a program are independent of one another, the smaller the size the better [19].

We may estimate the relative number of units obtained in each approach as follows:

- In the event-centred approach there is a unit for each event type.

- In the object-centred approach there is a unit for each object type. In general, the number of object types is less than the number of event types.
- In the cause-effect rule-centred approach there is a unit per cause-effect. In general, there are more cause-effect rules than event types.

Therefore, we conclude the number of specification units in the cause-effect rule approach is larger than in the other approaches. Inversely, the average size of functionality per unit is the smallest.

The second property is coupling. In Structured Design, coupling is a measure of the strength of the interconnection between program modules. Strong coupling complicates a system since a module is harder to understand, change or correct by itself if it is highly interrelated with other modules. The number of connections between modules, and the degree to which each connection couples (associates) two modules determines the complexity of the program. Complexity can be reduced by designing programs with the weakest possible coupling between modules [19]. In Object-Oriented Design, [4] defines coupling of an object class as the count of the number of other classes to which it is coupled. Two classes are coupled when methods declared in one class use methods or instance variables of the other class. As before, a large coupling is considered to have undesirable effects on understandability and changeability.

As defined above, coupling is not directly applicable to our units since, in general, they cannot be considered program modules nor object classes. The types of connections between modules (content, control, common, etc.) or object classes are not exactly the same as those between specification units. However, we can try to adapt the concept and make coupling also a relevant property of specification units. To this end, we distinguish the following eight types of connections between units:

- Reference ("access" or "read") to a fact (or sets of facts) of the structural model
- Reference to an external or generated event.
- Production of a generated event.
- Reference to an internal event.
- Fact modification.
- Establishing a postcondition.
- Fact definition.
- Production of internal event.

A preliminary analysis of the distinct approaches with respect to the above connections seem to suggest that the event-centred, object-centred (implicit) and the cause-effect rule-centred approaches give a similar level of coupling, while the remaining one (object-centred (explicit)) is the one giving specification units with the strongest coupling.

Finally, we consider cohesion. In Structured Design, cohesion is a measure of the functional relatedness of elements of a single program module. Cohesion is measured in a qualitative scale with seven levels. In order of increasing strength of cohesion, from least functionally related to most, these are: coincidental, logical, temporal, procedural, communicational, sequential and functional. A functionally cohesive module performs a single, well-defined task on a single subject [19].

In Object-Oriented Design, [4] defines cohesion of an object class as the degree of similarity of its methods. Two methods are similar if they perform operations on the same set of instance variables. As before, it is considered that a high cohesion decreases complexity.

As defined above, cohesion is not directly applicable to our units since, in general, they cannot be considered program modules nor object classes. However, we can adapt the concept and make cohesion also a relevant property of specification units, measuring the relatedness between the components of a unit, and the degree of minimality of the whole unit.

A preliminary analysis of the distinct approaches with respect to cohesion seem to suggest that the object-centred approach (both subapproaches) is the one giving specifications with less cohesion, followed by the event-centred and with the cause-effect rule-centred one giving specification units with the strongest cohesion.

This results, however, should be taken as conjectures to be validated experimentally. On the other hand, other properties should be analysed to obtain a complete and fair evaluation of the specification approaches.

Acknowledgment

This work has been partly supported by the Comisión Interministerial de Ciencia y Tecnología (CICYT), project TIC95-0735.

References

- [1] Baralis,E.; Ceri,S.; Paraboschi,S. Modularization Techniques for Active Rules Design, ACM TODS, Vol.21,No.1, 1996, pp. 1–29.
- [2] Bertino,E.; Ferrari,E.; Guerrini,G. A Formal Temporal Object-Oriented Data Model. Proc. EDBT96, Avignon, 1996, pp. 342–356.
- [3] Ceri,S.; Fraternali,P. The Story of the IDEA methodology, Proc. CAiSE97, Barcelona, LNCS 1250, Springer, 1997, pp. 1–17.
- [4] Chidamber,S.R.; Kemerer,C.F. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, Vol. 20, No. 6, June, 1994, pp. 476–493.
- [5] Cook,S.; Daniels,J. Designing Object Systems. Object-Oriented Modeling with Syntropy, Prentice Hall, 1994.
- [6] Fraternali,P.; Tanca,L. A Structured Approach for the Definition of the Semantics of Active Databases, ACM TODS, Vol. 20,No. 4, 1996, pp. 414–471.
- [7] de Champeaux,D.; Faure,P. A comparative study of object-oriented analysis methods, JOOP, March/April, 1992, pp. 11–33.
- [8] Gustafsson,M.R.; Karlsson,T.; Bubenko,J.A. A Declarative Approach to Conceptual Information Modeling. In Information Systems Design Methodologies: A Comparative Review, North-Holland, 1982, pp. 93–142.
- [9] Hagelstein,J.; Roclants,D. Reconciling Operational and Declarative Specifications, Springer-Verlag, LNCS 593, 1992, pp. 221–238.
- [10] Kowalski,R. Logic Programming, Proc. IFIP 83, Elsevier Science Publishers, 1983, pp. 133–145.
- [11] Martin,J.; Odell,J.J. Object-oriented Methods: A Foundation, Prentice Hall, 1995.
- [12] Moffett,J.; Hall,J.; Coombes,A.; McDermid,J. A model for a causal logic for requirements engineering, Requirements Engineering, Vol.1,No.1, 1996, pp. 27–46.
- [13] Olivé,A. On the design and implementation of information systems from deductive conceptual models, Proc.VLDB89, Amsterdam, 1989, pp. 3–11.
- [14] Peckham,J.; Maryanski,F. Semantic Data Models, ACM Computing Surveys, 20, 3 (Sept.), 1988, pp. 153–189.
- [15] Rational Software Corporation, Unified Modeling Language (UML), Version 1.0, January 1997.

- [16] van Griethuysen,J.J. Concepts and terminology for the conceptual schema and the information base, ISO/TC97/SC5/WG3. 1982.
- [17] Veloso,P.A.S.; Furtado,A. Towards simpler and yet complete formal specifications. In Sernadas,A.; Bubenko jr.,J.; Olivé,A. Information Systems: Theoretical and formal aspects, North-Holland, 1975, pp. 175–189.
- [18] Widom,J.; Ceri,S. Active Database Systems. Triggers and Rules for Advanced Database Processing, Morgan Kauffmann Publishers Inc. 1996.
- [19] Yourdon,E.; Constantine,L. Structured Design. Yourdon Inc, New York. 1975.