Towards Event-Driven Modelling for Database Design

Maguelonne Teisseire[1] Pascal Poncelet[2] Rosine Cicchetti[2]
[1] Digital Equipment - [2] IUT Aix-en-Provence

LIM - URA CNRS 1787 - Université d'Aix-Marseille II
Faculté des Sciences de Luminy, 163 Avenue de Luminy, Case 901, 13288 Marseille Cedex 9 FRANCE
e-mail: teisseir@lim.univ-mrs.fr

Abstract

This paper is devoted to the dynamic aspect of the IFO_2 conceptual model, an extension of the semantic IFO model defined by S. Abiteboul and R. Hull. Its original aspects are a "whole-event" approach, the use of constructors to express combinations of events, and the modularity and re-usability of specifications in order to optimize the designer's work. Furthermore, it offers an overview of the represented behaviour. To complement the modelling part, IFO_2 includes a derivation component which performs the implementation of specifications by using an active DBMS.

1 Introduction

Current conceptual approaches [BM91, LZ92, PS92, Per90, QO93, RC91, Saa91, SF91, SSE87] strive to meet the needs of both traditional and advanced applications. This goal is ambitious since it consists in preserving the benefits of semantic approaches [HK87] while integrating the strengths of Object-Oriented Data Base (OODB) models. In other words, these approaches not only have to handle complex constructors and propose concepts such as modularity and re-usability but they also have to offer an overview that comes as close as possible to the modelled real world. Furthermore, they must be independent of target systems but they have to be complemented by a derivation process in order to perform the application implementation. The difficulties increase when we consider the twofold aspect of the problem: application modelling includes both structural and behavioural representations. For this reason, conceptual models have often dealt with either one or the other, but not both of these aspects.

The IFO_2 conceptual model integrates both the structural and the behavioural representation of applications in a consistent and uniform manner in terms of both the formalization introduced and the associated graphic representation. It is based on the semantic IFO model of S. Abiteboul and R. Hull [AH87].
This paper is devoted to the dynamic aspect of the model (its structural part is described in [PTCL93]) and to its derivation which implements the described behaviour by using an active DBMS [CBa90].

After a survey of related work, we give our reasons for proposing the IFO_2 behavioural model (section 2). The various concepts introduced to represent the application behaviour are presented in section 3. The dynamic operation of a modelled system, which we call its activity, is then described (section 4). In section 5, we present the derivation of the dynamic part of IFO_2. We conclude with an examination of the links established between the structural and behavioural specifications. In the appendix, we summarize the IFO_2 structural part in order to give an overview of the model and to highlight its uniformity.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

2 Related work and motivations

Conceptual models which give priority to behavioural specification [Per90, QO93, RC91, FS88, SF91] use an OODB model to represent the structural part of applications [RC92]. They deal with problems which are similar to those of concurrent system design and software engineering [LZ92].
In these approaches, the behaviour of applications is viewed as the set of reactions of the modelled objects when certain events occur. In fact, these models differentiatex between local and global behaviours. Local behaviours focus on the object dynamics within classes while global behaviours represent the interactions between classes. Consequently two kinds of events are considered: local and shared events.
The object dynamics can be seen as states and transitions between states. In fact, local behaviour consists of valid transitions triggered by local events. An additional mechanism is required to coordinate the changes of object states belonging to different classes when shared events occur. These approaches make use of statecharts or temporal logic to represent the behaviour of the system.
This view of behaviour is simple, clear and works well with object-oriented models. For local behaviours, a behavioural facet integrated in the class description, specifies events, occurrence conditions and triggered actions (methods or other events which are produced). The description of global behaviours is either encapsulated in the classes (through interaction equations in [SSE87]) or specified outside the classes [RC91]. In the latter case, no one particular class is favored but the behaviour in question is excluded from the inheritance hierarchy.

Nevertheless these models do have some drawbacks. First of all, they do not provide an overview of the system's behaviour since it is divided into classes: priority is given to a complete vision of objects (structural and behavioural) rather than an overview of the system. Events are not represented in a uniform way: two abstraction mechanisms are necessary for local and shared events. Futhermore, behavioural inheritance depends on structural representation. In fact behavioural aspects may be re-used only for specialized objects according to the static inheritance hierarchy.
In these models, the conditions over event occurrences make use of object states. These states and the underlying objects are not always easy to identify since they do not necessarily correspond to attributes existing in the real world [AG93, RC92]. Consequently, it is necessary to introduce "artificial" objects in the structural representation in order to express behavioural constraints. This view of object behaviour as states has two disadvantages: the problem of making a complete inventory of states [Har88] and, above all, a structural representation which is no less faithful to the real world because of "artificial" objects. Finally, the concepts defined for the static and dynamic representations are very different and the designer's required skills need to be extended.

In proposing the IFO_2 model, our basic idea is that a conceptual model must offer the same qualities for the structural representation as for the behavioural modelling. More precisely, it must provide the designer with an overview of specifications not only for the structural part but also for the application behaviour. These specifications must be as faithful as possible to the real world. The modularity and re-usability mechanisms have to be as powerful in the static context as in the dynamic context.

In IFO_2 the system's behaviour is not understood as the reactions of objects to particular events but rather as the events which may operate on objects. These objects are specified as the parameters of events. We believe that a conceptual model which adopts this view can be as expressive as others while avoiding their drawbacks. Furthermore the modelling process is not object-driven since objects and events have the same importance in IFO_2 and play a symetrical role in the static and dynamic parts of the model. Consequently, we propose a twofold modelling approach to represent applications. The result of this modelling is a structural schema and a behavioural schema which are closely related but clearly distinct. As in the case of the conceptual qualities of the model, the problems for the designer are the same in the dynamic context as they are in the static context. These are: identify the representation units (objects or events) and specify the relations between them (for instance aggregation or specialization links between objects and synchronization or triggering chaining between events). From this idea, we adapt the concepts defined for the structural part of IFO_2 to the application behaviour. These concepts, which give to the static part of the model, the required conceptual qualities described above play the same role in the dynamic part. The main advantage here is the uniformity of the model. The designer handles the same concepts, or rather concepts having the same philosophy, for both the static and dynamic representations.
In the following section, we present the behavioural part of IFO_2 and show that this type of representation offers an expressive power comparable to other models while respecting the conceptual qualities mentioned above. The reader interested in the formalization of

the model may refer to [TPC94].

3 IFO_2 behavioural model

An event is the representation of a fact that participates in the reactions of the modelled system. It occurs in a spontaneous manner (in the case of external or temporal events) or is generated by the application. In both cases, it occurs instantaneously, i.e. it is of zero duration. As in [Cha89, GJS92], we make the following two assumptions: no more than one event can occur at any given instant and the time scale is infinitely dense.

The structural part of IFO_2 is defined with respect to the "whole-object" philosophy. We extend its scope to the behavioural part and refer to a "*whole-event*" representation, i.e. any fact which is involved in the system's reaction is modelled in IFO_2 as an event. In fact, event modelling in IFO_2 complies with a dual precept: typing and identification. For identification, we use the instant of an event occurrence as its identifier. The IFO_2 behavioural model proposes three basic types. Their graphic formalism is illustrated in Figure 1.

- The *Simple Event Type* (TES) represents the events that trigger a method included in the IFO_2 structural description. This means that we do not consider operations in the behavioural part of the model but only the events triggering these operations.
- The *Abstract Event Type* (TEA) is used to specify external and temporal events or events that generate other events.
- The *Represented Event Type* (TER) symbolizes any other type which may then be re-used without knowing its precise description.

Example 1 To illustrate the concepts presented in this paper, we use the example of a lift to represent a modelled system. A simple event type involved in the description of the lift behaviour is "Up", which describes the ascending motion of the lift cage and maps with a method of the structural fragment "Lift" (See Appendix). The TEA "Floor-Request" represents external events which occur when users request a floor (inside or outside the cage). "Satis-Request" corresponds to internal events produced when users reach the requested floor and "Arrival-Floor" stands for a different type of event.

To model the behaviour of a system, it is necessary to express synchronization conditions, i.e. different variants of event conjunction and disjunction. To answer

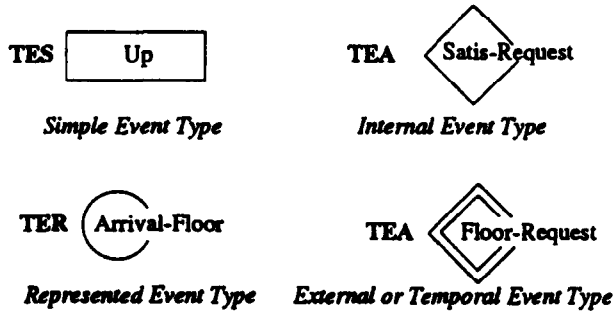

Figure 1: Example of Basic Event Types

this need, we represent complex events by using *constructors*. With this approach, we provide not only the required expressive power but also the uniformity with respect to the IFO_2 structural modelling [PTCL93]. The event constructors, which may be recursively applied, are the following: composition, sequence, grouping and union. The event composition constructor reflects the conjunction of events of different types. The sequence constructor is defined like the composition constructor but with a chronological constraint on the occurrences of the component events. Event collections, i.e. conjunctions of events of the same type, are translated in IFO_2 by the grouping constructor (similar to the HiPAC closure constructor [CBa90]). Finally, the union constructor expresses a disjunction of events of different types. The associated graphic formalism is illustrated in Figure 2.

Example 2 Let us imagine that the designer wants to specify the descending or ascending lift motion. He can use the union type "Up-Down" which is an alternative between the two simple types "Up" and "Down". Each one of these types triggers a method which performs a single floor motion for the cage.

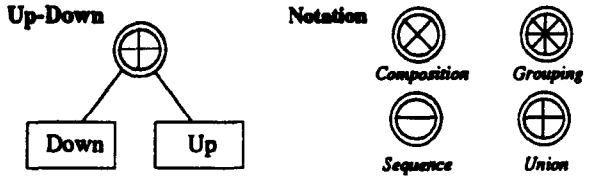

Figure 2: Example of Complex Type

The dynamics of the system may be seen as the linking of events (atomic or composite). These causality links are expressed through functions. In fact, the event types are interconnected by functions through the *event fragment concept*, focused on a principal type called *heart*. Functions may combine the following features:

- simple or complex (mono or multivalued), i.e. an event of their type origin triggers one or several events of their target;
- partial or total, i.e. an event of their type origin can or must trigger an event of their target;
- and deferred or immediate, if there is a delay or not between the occurrences of the origin and target events.

In addition, we differentiate between *triggering* and *precedence functions*. These roughly express the fact that an event of the fragment heart triggers the occurrence of other events or that it is preceded by the occurrence of other events. In order to emphasize this point, let us consider an external event. By its very nature, it cannot be triggered by another modelled event, therefore it is sometimes necessary to express that its occurrence is necessarily preceded by other events.

All the described features of functions express general constraints on event chaining. They are reflected through the proposed graphic formalism (Figure 3). Nevertheless, it is sometimes necessary to refine these constraints by specifying particularly precise conditions over the system's history. In IFO_2, such conditions are expressed on another specification level by using an algebraic language on events. This language is not presented in this paper due to space limitations (some operators are detailed in [TC94]).

Contrary to triggering functions, a precedence function, if it exists, is unique in a fragment since the different triggered events may be constrained by different specific conditions while preceding events can only be synchronized by using the mentioned constructors to trigger one event of the fragment heart.
The concept of fragment, inherited from the IFO model, is very important for the modularity of specifications. The fragment can be really considered as a *unit of description* of the dynamics since it specifies a complete "sub-behaviour" of the modelled system (with the heart events, their preceding and triggered events).

Example 3 Figure 3 illustrates a fragment in which the heart is the external event type "Floor-Request". In this fragment, there is no precedence function. This fragment describes the lift reactions when a user requests a floor inside or outside the cage. The fragment heart is linked with a partial, complex and deferred function to the simple type "Closure". The associated method in the structural schema closes the lift doors. The function is partial because, in some cases, an event of "Floor-Request" would not trigger a door closure. These cases are the following: (i) the user wishes to go to the floor where he is currently located; (ii) or the door closure stems from another event, i.e. a previous request from the same floor. The function is deferred to account for the case where the user requests the lift while the latter is moving up or down.
The complex feature of the function specifies that a floor request may trigger the door closure several times. This situation occurs when a request processing is interrupted to serve floors requested by other users. When other users are satisfied the considered request triggers the closure of the door so that the cage can start again.
The TEA is also related to the composite type "Up-Down" which specifies an alternative between the two TESs "Down" and "Up". The triggering function between the heart and the union type is partial, deferred and complex. It is partial to take into account three cases: cases (i) and (ii) of the previous function and the case where the requested floor is served when satisfying previous current requests. The deferred feature of the function takes into consideration the possible delay between the user request and the resulting lift motion. In fact the methods corresponding to the TESs "Up" and "Down" perform a single floor ascent or descent for the cage. This is what makes the triggering function complex. The union type "Up-Down" is the heart of a subfragment. The triggering function which relates it to the represented type "Arrival-Floor" (standing for a type described in another fragment and describing the cage arrival to the floor) is total and immediate. This means that any event of the types "Up" and "Down" generates an event of "Arrival-Floor".

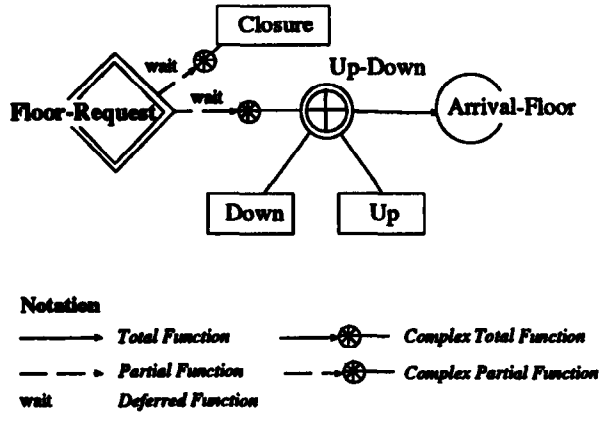

Figure 3: The "Floor-Request" Event Fragment

The role of fragments is to describe a subset of the modelled behaviour that can then be used as a whole by means of the *represented type* concept. More precisely, represented types are related to fragment hearts via *IS_A event links*. Consequently, it is possible to

manipulate another type without knowing its description. The designer may defer a type description or entrust it to somebody else, while using a represented type which symbolizes it. Through the concept of represented type, the re-usability of specifications is real. An inherited behavioural aspect may be re-defined or refined by specifying the concerned represented type as the heart of a new fragment having other preceding or triggered types. Furthermore, IFO_2 takes the multiple inheritance into account since represented types may have several sources. The inheritance mechanism, introduced by event IS_A links, is independent from the structural inheritance hierarchy. It is possible to reuse parts of the modelled behaviour even if they do not concern specialized objects. Consequently the reusability of dynamic specifications is not limited by static considerations.
In order to model the general behaviour of the application, the partial views provided by the fragments are combined (via IS_A links) within an *event schema.*

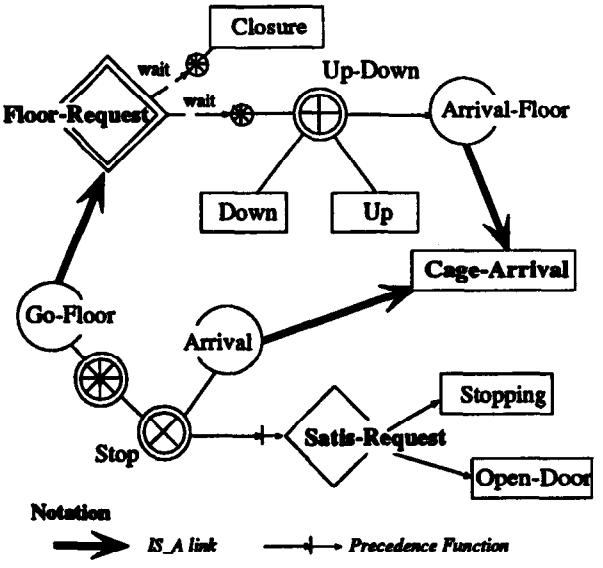

Figure 4: The "Lift" IFO_2 Event Schema

Example 4 Figure 4 shows the IFO_2 event schema "Lift", involving three fragments, each one dedicated to a particular aspect of the lift reactions. "Floor-Request" describes the system behaviour when a user request occurs. "Cage-Arrival" is a particular fragment since it is reduced to its heart which is a simple event type re-used in other fragments. The corresponding method in the structural fragment "Lift" is an alerter which returns the floor reached by the cage. Finally "Satis-Request" is dedicated to the lift behaviour when the cage arrives at the requested floor. The origin of the precedence function is a composite type "Stop". It combines several events of the TER "Go-Floor" (by using the grouping constructor), in fact several floor requests, and a cage arrival. This composite type specifies which current floor requests have to be satisfied (as explained in the next section). These fragments are related by IS_A links through the represented types "Go-Floor", "Arrival-Floor" and "Arrival".

IFO_2 event schemas describe possible application behaviours exclusively in terms of events. The expressive capabilities of the model are comparable to those of object-state and transition based approaches. Despite their different philosophies [SF91, RC92], events and synchronisation conditions are quite close. IFO_2 however introduces synchronisation between operations, through their triggering events while sequential order is adopted in other models. Moreover, general conditions are expressed in an explicite way with constructors and a twofold causality link is proposed through the precedence and triggering functions. Compared with statecharts [Har88], the IFO_2 graphic representation is very concise and offers an overview of the modelled behaviour.

4 Activity of an IFO_2 schema

The application behaviour is represented by the event schema. It may be simulated by navigation through the graph. An outline of this behaviour consists in a propagation of event triggering. It stops when all the actions reflecting the goal sought by the system are achieved. These actions are described in the schema, within one or more fragments called *satisfaction fragments.*

Example 5 In our schema example, there is one satisfaction fragment: "Satis-Request", which specifies that each user who requests a floor has to reach it, in the end.

This section provides further details of this general principle. Within each fragment, the triggering propagation is of course oriented by the precedence and triggering functions. This propagation stems from an external or temporal event or a combination of such events. In this case, the underlying TEA (or the type built up from these TEAs) is either the heart of a fragment without a precedence function or it is the origin of the precedence function in a fragment. It is considered as an *entry* of the IFO_2 event graph.

Example 6 The only fragment illustrating this case, in our schema exemple, is "Floor-Request" which, consequently, is the only entry of the graph. In fact,

the whole triggering cascade stems from the external event: a user requests a floor. Let us suppose that there is just one user at this instant and that he wishes to go to a floor above the one where he is. Now, in this situation, the external event triggers the door closure and then the occurrence of an "Up" event. This event causes the generation of an "Up-Down" event. In a general way, events of the composite type stem from the occurrence of the component types. The propagation continues on the level of the subfragment by the triggering of an "Arrival-Floor" event.

Therefore, in the general case, the event propagation is triggered, within a fragment, by the occurrence of an event in another fragment. The behaviour is then simulated by the navigation along the IS_A link associating these two fragments, i.e. the heart of one to a TER of the other. Along an IS_A event link, the navigation can follow one direction or the other depending on the "position" of the TER within its own fragment. If it is the target of a triggering function, directly or by construction, then the event occurrence of the TER systematically generates an event occurrence of the heart type of the related fragment. Therefore the navigation takes place from the IS_A link target to its source and there is equality between the sets of occurred events of both the fragment heart and the TER, if the latter has only one source[1].

Example 7 In the "Lift" event schema, one TER is a target of a triggering function: "Arrival-Floor". According to our previous assumption, an event of this type has just happened. This triggers the occurrence of a fragment heart "Cage-Arrival" event by navigating along the IS_A link between these two types.

Let us now consider a TER that is the origin of the precedence function, or involved in the construction of this origin. This means that one of its events can only be triggered by the occurrence of an event belonging to the fragment heart to which it is related. The navigation along the IS_A link takes place from its source to its target and there is inclusion or equality between the sets of events occurred for the TER and the heart, if the former has only one source.

Example 8 The two TERs, illustrating this situation in our event schema, are "Go-floor" and "Arrival". They are both used to build up the "Stop" type origin of the precedence function in the fragment "Satis-Request". The occurrence of an event of these types is necessarily caused by an event of "Floor-Request" or "Cage-Arrival" respectively.

This first outline of the system activity shows that "everything begins" with the occurrence of external or temporal events.
When a triggering cascade is started, it must stop in the end. "Everything ends" on the level of satisfaction fragments since they model, as previously mentioned, the ultimate goal of the system. In fact such fragments not only describe the manner in which the propagation stops. They also specify this obligation to stop, by integrating the following constraint: a satisfaction fragment has to include a TER which is actually a triggering type in the fragment (i.e. it is either the source of the precedence function or the heart of the concerned fragment). The obligation to stop is then partially taken into account by the fact that any event of the related fragment heart must also be an event of the TER. The heart events are considered as being satisfied when the TER corresponding events actually produce the set of triggered events in the satisfaction fragment. This vision has to be refined by taking into account iterations that would possibly be performed during the graph navigation. Iterations aroused by the satisfaction fragment are performed by considering triggering functions which are complex or deferred. The chosen iteration is the first one found along the reverse path.

Example 9 Let us resume the activity of our schema example where we last left it, i.e. after the occurrence of a "Cage-Arrival" event. From this event stems an event of the "Arrival" represented type. Similarly, from the initial floor request stemmed a corresponding event for the TER "Go-Floor" in the satisfaction fragment. During the cage motion, let us suppose that another person calls the lift from the floor that is requested by the first user. This call does not yet generate any event in the "Floor-Request" fragment because the lift is engaged, but it triggers a corresponding event for the TER "Go-Floor". At this stage, none of the two floor requests may be satisfied because the cage is not yet at the desired floor. This condition is expressed in the precedence function of the satisfaction fragment by comparing the structural parameters standing for the floor of the event "Arrival" and of the events "Go-Floor". A first iteration is then performed, by following the IS_A links, in order to again trigger the complex function between "Floor- Request" and "Up-Down". Such a process is performed as many times as necessary to reach the required floor. When this happens, the condition of the precedence function in the fragment "Satis-Request" is true and the propagation carries on generating a fragment heart event. Let us examine its preceding event, which is of the

[1] We do not explain the case where a TER has several sources since its description requires the concept of attached events not presented here due to space limitation.

"Stop" type. This type is defined as a composition of the "Arrival" type and a "Go-Floor" grouping. Therefore, the preceding event in question is built up from our two user requests combined with the last "Arrival" event. The event of the type "Satis-Request" immediately generates a "Stopping" event and then triggers the opening of the lift doors.

5 From an IFO_2 event schema to E-C-A rules

In this section, we propose a derivation process to perform, from an IFO_2 event schema, the implementation of the modelled behaviour. Since our aim is to demonstrate the feasibility of translation, we opt for E-C-A rules similar to those of HiPAC [CBa90] (i.e. adopting its philosophy but without strictly following its syntax) because it is recognized as a reference in the area of active DBMS research [Cha89, CBa90, DBM88, DHL91, DPG91, GJS92].

5.1 Presentation

On the basis of an IFO_2 event schema, the algorithm described below is used to generate a set of E-C-A rules. Generally speaking, this process starts with the entries of the IFO_2 graph, examining the corresponding fragments. It continues by transforming the fragments linked to the entry fragments, following the IS_A links from their target to their source. All fragments that are not yet derived are then examined. More precisely, each fragment gives rise to the creation of at least one E-C-A rule, except in the case where it is reduced to a TES. In general, however, a fragment generates several E-C-A rules. Without presenting the algorithm in detail, we give its general principles. The actions of the generated rules are either the methods corresponding to the TESs or the triggering or activation of the rules introduced during the derivation process.

In IFO_2 the conditions of event triggering are expressed when specifying the functions, in a language [TC94] that we do not present in this paper. The conditions must be exhibited in a preliminary stage of the application of the algorithm by adopting the same philosophy which is proposed in the derivation illustration.

The events that trigger the E-C-A rules can be external or temporal events of the IFO_2 schema. When a composite type is the target of a precedence function, the constructor used gives the type of combination of events of the E-C-A rule (conjunction, disjunction, sequence or grouping). If a type is the target of a triggering function, it is translated - in the simplest case of a composition or sequence whose children are TESs - by a series of actions. For the two other types, i.e. union or grouping, the actions generated automatically by the algorithm are rule triggering or rule activations. In the case of union, there are as many actions as component children and, for each of these actions, a new E-C-A rule is generated. In the case of a grouping, a single action is generated and a single new E-C-A rule is introduced, but we call it recursive because it must be executed several times: this is carried out by re-activating the rule.

The TERs are handled in a particular manner because their role in IFO_2 is to represent another type which allows for the modularity of specifications. If they initially participate in a fragment precedence function, their translation has repercussions on the event part of the rule derived from the fragment. In fact, this event part includes a disjunction of events corresponding to the different sources of the TER. If they participate in a fragment triggering function, their translation has repercussions on the actions of the generated rule. With the same philosophy as for derivation of union types, a triggering or activation action of new rules is created for each source of TERs, and these new rules are then generated.

Lastly, the characteristics of the functions influence the derivation that is performed. The deferred triggering functions of a fragment can introduce a deferred coupling mode between the Event component and the Condition component. They can also be translated in the case of recursive rule generation by activation (instead of triggering) of a new rule. The complex functions are translated according to the same principles as the grouping types, since they represent an iteration on the level of the target event types, i.e. through generation of recursive rules triggered by an action of the rule corresponding to the fragment. Let us note that the rules which we call recursive have the peculiarity of not having triggering events; this is because they have to be executed several times. In the ODE model [GJS92], they could correspond to "perpetual" rules. In the ATM model [DHL91], such rules would be described in an even more natural manner because the activity concept allows for the introduction of a loop.

5.2 The Derivation Algorithm

To specify complex events, we use the following notations: | for event disjunction, , for event conjunction, ; for sequence and * for grouping (closure). *Mode* denotes the coupling mode between E-C or C-A.

Derivation Algorithm:
Input: both IFO_2 structural and event schemas.
Output: a set of E-C-A rules.
Step 1: For each fragment entry, T_{entry}, of the event graph: do $RULE(T_{entry}, ((EVENT(T_{entry}), immediate), (true, immediate), \emptyset))$.
Step 2: For each non examined fragment heart, T_{heart}: do $RULE(T_{heart}, (\emptyset, (true, immediate), \emptyset))$.

The ***RULE* function** is defined by:
$RULE(T, ((E, Mode_E), (C, Mode_C), A)) = r_T$
where r_T is an E-C-A rule obtained as follows:

- $r_T = ((E', Mode_{E'}), (C', Mode_{C'}), A')$ with $(E', Mode_{E'}) = (E, Mode_E)$, $(C', Mode_{C'}) = (C, Mode_C)$ and $A' = A \cup ACTION(T)$.
- If T, a principal fragment heart (i.e. not heart of a subfragment), is the target of a precedence function, f_p, of domain T_p:
  1. If T is an external or temporal event or is built up only from such events, then $E' = (E'; EVENT(T_p); EVENT(T))$. Through this sequence, the semantics of the precedence function is captured.
     Else, $E' = (E', EVENT(T_p))$ and $Mode_{E'} = deferred$.
  2. Let c_p be the condition expressed in the function f_p then $C' = C' \wedge c_p$.
- If T is the origin of n functions f_i of codomain T_i, we denote by c_i the possible condition associated to f_i, then: $\forall i \in [1..n]$
  1. If f_i is a total, immediate and simple triggering function (c_i is reduced to "true" except for the union and grouping constructors) then:
     (a) If T_i is a simple event type then $A' = A'$ followed by $ACTION(T_i)$.
     (b) If T_i is a represented event type, for each source T_s of T_i then:
     $A' = A'$ followed by *fire* $RULE(T_s, ((E', Mode_{E'}), true, \emptyset))$.
     (c) If T_i is an union constructor, built up from k types, t_k, the condition c_i is a set of k conditions c_k, then: for each k, $A' = A'$ followed by *fire* $RULE(t_k, ((E', Mode_{E'}), (c_k, immediate), \emptyset))$.
     (d) If T_i is a composition constructor built up from k types, t_k, then:
     $$A' = A' \text{ followed by } \bigcup_{j=1}^{k} ACTION(t_k).$$
     (e) If T_i is a sequence constructor built up from k types, t_k, then:
     $A' = A'$ followed by $ACTION(t_1) \cdots$ followed by $ACTION(t_k)$.
     (f) If T_i is a grouping constructor of child t, the condition c_i captures the number of generated events of t (more precisely, c_i holds until this number is reached by a repeated execution of the underlying rule), then $A' = A'$ followed by *fire* $RULE_{Recursive}(t, (\emptyset, (c_i, immediate), \emptyset))$.
  2. If f_i is a partial, immediate and simple triggering function:
     (a) If T_i is a simple event type then: $A' = A'$ followed by *fire* $RULE(T_i, ((E', Mode_{E'}), (c_i, immediate), \emptyset))$.
     (b) If T_i is a represented event type then: for each source T_s of T, $A' = A'$ followed by *fire* $RULE(T_s, ((E', Mode_{E'}), (c_i, immediate), \emptyset))$.
     (c) If T_i is an union construcor built up from k types, t_k, the condition c_i is a set of k conditions c_k, then: for each k, $A' = A'$ followed by *fire* $RULE(t_k, ((E', Mode_{E'}), (c_k, immediate), \emptyset))$.
     (d) If T_i is a composition or sequence constructor then: $A' = A'$ followed by *fire* $RULE(T_i, ((E', Mode_{E'}), (c_i, immediate), \emptyset))$.
     (e) If T_i is a grouping constructor of child t then: $A' = A'$ followed by *fire* $RULE_{Recursive}(t, (\emptyset, (c_i, immediate), \emptyset))$.
  3. If f_i is a deferred and simple triggering function then:
     (a) If T_i is a simple event type then: $A' = A'$ followed by *fire* $RULE(T_i, ((E', deferred), (c_i, immediate), \emptyset))$.
     (b) If T_i is a represented event type, then: for each source T_s of T, $A' = A'$ followed by *fire* $RULE(T_s, ((E', deferred), (c_i, immediate), \emptyset))$.
     (c) If T_i is an union constructor built up from k types, t_k, the condition c_i is a set of k conditions c_k then: for each k, $A' = A'$ followed by *fire* $RULE(t_k, ((E', deferred), (c_k, immediate), \emptyset))$.
     (d) If T_i is a composition or sequence constructor then: $A' = A'$ followed by *fire* $RULE(T_i, ((E', deferred), (c_i, immediate), \emptyset))$.

(e) If T_i is a grouping constructor of child t then: $A' = A'$ followed by *enable* $RULE_{Recursive}(t,(\emptyset,(c_i,immediate),\emptyset))$.

4. If f_i is a complex triggering function then $A' = A'$ followed by *enable* $RULE_{Recursive}(T_i, (\emptyset, (c_i, immediate), \emptyset))$.

The $RULE_{Recursive}$ **function** is defined by:
$RULE_{Recursive}(T, (\emptyset, (C, Mode_C), A))= r_T$
where $r_T = (\emptyset, (C'', Mode_{C''}), A'')$ is an E-C-A rule obtained as follows:
if $(\emptyset, (C', Mode_{C'}), A')=RULE(T, (\emptyset, (C, Mode_C), A))$, then: $(C'', Mode_{C''})=(C', Mode_{C'})$, $A'' = A'$ followed by *enable* r_T.

The **function *ACTION*** returns the actions of the specified type:

1. If T is an abstract event type then: $ACTION(T) = \emptyset$.
2. If T is a simple event type then: $ACTION(T) = Occ(T)$[2].
3. If T is an union constructor built up from k types, t_k, we obtain k conditions, denoted by c_k, corresponding respectively to one of the event types, then: for each k $ACTION(T) = fire\ RULE(t_k, (\emptyset, (c_k, immediate), \emptyset))$.
4. If T is a represented event type of p sources noted T_{s_p}, then:

$$ACTION(T) = \bigcup_{i=1}^{p} fire\ RULE(T_{s_p}, (\emptyset, \emptyset, \emptyset)).$$

5. If T is a composition constructor built up from k types t_k, then:

$$ACTION(T) = \bigcup_{j=1}^{k} ACTION(t_k).$$

6. If T is a sequence constructor built up from k types t_k, then: $ACTION(T) = ACTION(t_1)$ followed by $ACTION(t_2)\cdots$ followed by $ACTION(t_k)$.

The result of the **function *EVENT*** is an event, possibly complex, corresponding to the IFO_2 specified event type:

1. If T is a simple or an abstract event type then: $EVENT(T) = T$.
2. If T is an union constructor, built up from k types t_k, then:
$EVENT(T) = (EVENT(t_1) \mid EVENT(t_2) \ldots \mid EVENT(t_k))$.
3. If T is a represented event type, it exists p sources denoted t_{s_p},then:
$EVENT(T) = (EVENT(t_{s_1}) \mid EVENT(t_{s_2}) \ldots \mid EVENT(t_{s_p}))$.
4. If T is a composition constructor built up from k types t_k, then:
$EVENT(T) = (EVENT(t_1)\ ,\ EVENT(t_2) \ldots, EVENT(t_k))$.
5. If T is a sequence constructor built up from k types t_k, then: $EVENT(T) = (EVENT(t_1)\ ;\ EVENT(t_2) \ldots ; EVENT(t_k))$.
6. If T is a grouping constructor of child t then: $EVENT(T) = EVENT(t)^*$.

After application of the derivation algorithm, it is possible to proceed to an optimization stage for reducing the number of generated rules. This mainly consists in taking all the rules with identical Event and Condition components and grouping them together into a single rule. For any rule with no triggering event and no condition, i.e. a rule triggered by another rule, it is also possible, on condition that it is not recursive, to include its actions in the calling rule. Lastly, we want to point out that, in the ATM model, all recursive rules could be eliminated by simply substituting their actions for their initial triggering within a "Repeat Until" loop that is part of the definition of an activity.

5.3 Illustration

We apply the previous algorithm to our event schema example (Figure 4). Firstly, we describe, in an intuitive way, the preliminary step for the state specification.
In fact, conditions over event occurrences are expressed through IFO_2 functions by using manipulation operators on events [TC94]. The problem is then to translate constraints over events into constraints over object states. The functions between "Floor-Request" and "Closure" includes two conditions. Firstly, a floor request would trigger the door closure only if the lift door is opened. In our specification language the expression of this condition looks like: "was there a door opening since the last closure?". An attribute, "Door-Status" whose values would be "closed" or "opened" appears suitable for capturing the required semantics. The second condition to be taken into account is the following: the door closure is performed only if there are still requests to be satisfied. These requests are

[2] Occ is a function which gives for the specified event its associated method or its corresponding occurrence.

identified, in the function specification language, by comparing the events of the "Floor-Request" type and the events of "Go-Floor" which actually triggered a "Satis-Request" event. To translate this constraint on events, it is relevant to introduce in the structural fragment "Lift" (See Appendix) an attribute called "Status" which indicates whether the cage is engaged or not ("engaged" or "waiting"). These attributes, shaded in figure 6 in the appendix, are particular since they are "artificial".

We now examine the application of the algorithm. In our example, the only entry of the graph is "Floor-Request". Consequently the derivation starts with the translation of the associated fragment by evaluating the following function:
RULE(Floor-Request, ((EVENT(Floor-Request), immediate), (true, immediate), ∅)).

The generated E-C-A rule includes the activation of two recursive rules translating the complex feature of the triggering functions in the fragment.

$r_{Floor\text{-}Request}$
E: Floor-Request
Immediate
C: True
Immediate
A: enable $r_{Closure}$
enable $r_{Up\text{-}Down}$

The recursive rule $r_{Closure}$ is achieved by:
$RULE_{Recursive}(Closure, (\emptyset, (C_{Closure}, immediate), \emptyset))$
where $C_{Closure}$ is the condition[3] extracted from the function between "Floor-Request" and "Closure". The actions of the rule consist in the closure method call and its own re-activation.

$r_{Closure}$
E:
C: Lift.Door-Status='opened' ∧
((Lift-Cage.Status='waiting'
∧ Para(Floor-Request, Floor) <>
Lift-Cage.Position)
∨ Lift-Cage.Status='engaged')
Immediate
A: Closure
enable $r_{Closure}$

In the same way, the translation of the composition type "Up-Down" generates a recursive rule. Let us note that the union constructor derivation triggers two rules, r_{Up} and r_{Down}, whose conditions are mutually exclusive.
The TER "Arrival" is translated into the method call of its source since the concerned fragment is reduced to a TES.

$r_{Up\text{-}Down}$
E:
C: Lift.Door-Status='closed' ∧
Lift-Cage.Status='engaged'
Immediate
A: fire r_{Down}
fire r_{Up}
Cage-Arrival
enable $r_{Up\text{-}Down}$

The derivation ends with the translation of the satisfaction fragment by applying:
RULE (Satis-Request, ((EVENT(Stop), immediate) , (true, immediate), ∅)).

The generated rule includes, in the event part, the composition which is the source of the precedence function in the fragment. It specifies which occurrences of "Floor-Request" actually cause the triggering of the "Stop" and "Open" methods.

$r_{Satis\text{-}Request}$
E: Floor-Request*, Cage-Arrival
Deferred
C: For each Floor-Request,Para(Floor-Request, Floor) = Cage.Position
Immediate
A: Stop
Open

6 Conclusion

In this paper, we have described the behavioural part of the IFO_2 conceptual model. Its original aspects are a "whole-event" approach, the use of constructors to express complex combinations of events and the re-usability and modularity of specifications in order to optimize the designer's work. The IFO_2 model offers a uniform specification of both the structural and the behavioural parts of applications. We believe that such a uniformity is particularly important on a conceptual level. In the two frameworks, structural and behavioural, the same fundamental concepts, such as re-usability, modularity, identification, and etc are used. Types, constructors and fragments are defined by adopting an analogous formalism and they have the same semantics or at least the same philosophy in the static and dynamic parts of the model. A homogeneous graphic representation is presented and can fa-

[3] Para is the function which associates to the specified event the object of its object type parameter.

cilitate the dialogue between designers [Har88] in order to better take advantage of specification modularity. Links between the structural and behavioural specifications are the following. First of all, basic operations are included in the associated structural schema and are used as simple types in the behavioural description. Object types on which event types operate are specified through the parameter concept. Finally, conditions over objects may be expressed in the specification of fragment functions.
The derivation component which generates E-C-A rules from the IFO_2 behavioural specifications can be associated to the transformation of IFO_2 structural schema into OODB models [PTCL93] in order to perform a complete implementation of the applications.

In an intuitive way, we have shown through our example how to transform conditions over events in conditions over states. This important step in the derivation process of IFO_2 schemas cannot be completely automated. Our aim however is to develop an aid in the identification of this states. This aid will make it possible to guide the designer in exhibiting objects which we call "artificial" and in specifying their domain. This will specifically concern the expressed conditions in the specification language of functions and the parameters of events.

Acknowledgement
The authors wish to thank the anonymous referees for their useful comments and Stefano Spaccapietra for improving the presentation of this paper.

References

[AG93] J. M. Atlee and J. Gannon. State-Based Model Checking of Event-Driven Requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM TODS*, 12(4):525–565, December 1987.

[BM91] M. Bouzeghoub and E. Métais. Semantic Modelling of Object-Oriented Databases. In *Proc. VLDB*, 1991.

[CBa90] S. Chakravarthy, B. Blaustein, and al. HiPAC: A Research Project in Active, Time-Constrained Database Management. Technical report, Xerox Advanced Information Technology, Cambrige, MA, August 1990.

[Cha89] S. Chakravarthy. Rule Management and Evaluation: An Active DBMS Perspective. *Sigmod Record*, 18(3):20–28, September 1989.

[DBM88] U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules Are Objects Too: A Knowledge Model for An Active, Object-Oriented Database System. In *Advances in Object-Oriented Database Systems*, volume 334 of *LNCS*, pages 129–143, 1988.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A Transactional Model for Long-Running Activities. In *Proc. VLDB*, 1991.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Uniform Approach. In *Proc. VLDB*, 1991.

[FS88] J. Fiadeiro and A. Sernadas. Specification and Verification of Database Dynamics. *Acta Informatica*, 25:625–661, 1988.

[GJS92] N. H. Gehani, H.V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *Proc. ACM Sigmod*, 1992.

[Har88] D. Harel. On visual formalisms. *CACM*, 31(5):514–530, 1988.

[HK87] R. Hull and R. King. Semantic Database Modelling: Survey, Applications and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[LZ92] P. Loucopoulos and R. Zicari. *Conceptual Modeling, Databases and CASE: An Integrated View of Information Systems Development.* Wiley Professional Computing, 1992.

[Per90] B. Pernici. Objects With Roles. In *Proceedings of the Conference on Office Information Systems*, pages 205–215, Cambridge, MA, April 1990.

[PS92] C. Parent and S. Spaccapietra. *ERC+: An Object-Based Entity Relationship Approach.* in [LZ92], 1992.

[PTCL93] P. Poncelet, M. Teisseire, R. Cicchetti, and L. Lakhal. Towards a Formal Approach for Object-Oriented Database Design. In *Proc. VLDB*, 1993.

[QO93] C. Quer and A. Olivé. Object Interaction in Object-Oriented Deductive Conceptual Models. In *Proc. CAiSE*, volume 685 of *LNCS*, pages 374–396, 1993.

[RC91] C. Rolland and C. Cauvet. Modélisation Conceptuelle Orientée Objet. In *Actes des 7ièmes Journées Bases de Données Avancées*, pages 299–325, Lyon, France, Septembre 1991.

[RC92] C. Rolland and C. Cauvet. *Trends and Perspectives in Conceptual Modeling.* in [LZ92], 1992.

[Saa91] G. Saake. Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, 6:47–73, 1991.

[SF91] C. Sernadas and J. Fiadeiro. Towards Object-Oriented Conceptual Modeling. *Data & Knowledge Engineering*, 6:479–508, 1991.

[SSE87] A. Sernadas, C. Sernadas, and H. D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In *Proc. VLDB*, 1987.

[TC94] M. Teisseire and R. Cicchetti. An Algebraic Approach for Event-Driven Modelling. In *Proc. DEXA*, LNCS, September 1994.

[TPC94] M. Teisseire, P. Poncelet, and R. Cicchetti. Dynamic Modelling with Events. In *Proc. CAiSE*, LNCS, June 1994.

Appendix

IFO_2 Structure: A brief outline

To present the static part of the IFO_2 model, type concept is firstly described as well as the different constructors. The fragment and the IFO_2 structural schema are then presented.
There are three basic object types (Figure 5):

- Printable Object Type (TOP), used for application I/O (Input/Output are therefore environment-dependent: String, Integer, Picture, Sound, etc.), which is comparable to the attribute type of the Entity-Relationship model;
- Abstract Object Type (TOA) which would be perceived as entity type in the Entity-Relationship model;
- Represented Object Type (TOR) which handles another type through the IS_A specialization link. This concept is particularly interesting when considering modularity and re-usability goals, since it allows the designer to use a type without knowing its precise description.

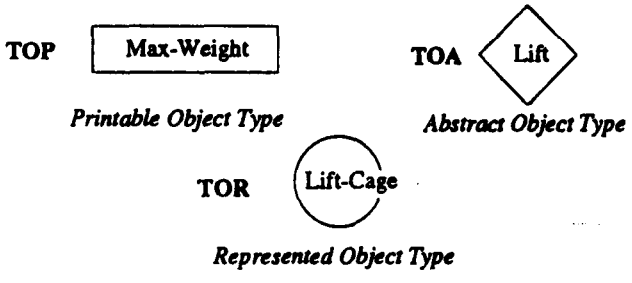

Figure 5: Example of Basic Type

IFO_2 takes into account five constructors:

- **aggregation and composition**: they represent the *tuple* constructor of OO models with an exclusivity constraint for the composition (an object can take part in a unique construction);
- **collection and grouping**: they represent the *set-of* constructor of OO models with an exclusivity constraint for the grouping;
- **union**: it is used for similar handling of structurally different types. This constructor represents the IS_A generalization link enhanced with a disjunction constraint between the generalized types.

These constructors can be recursively applied according to specified rules for building up more complex types. The types can be linked by functions (simple or complex; partial (0:N link) or total (1:N link)) through the fragment concept. The aim of the fragment is to describe properties of the principal type called heart. For each fragment, a set of methods is associated. IFO_2 being a conceptual model, only the signature of these methods is required. These operations are one of the links between the structural and the behavioural specifications.
Finally, an IFO_2 structural schema is a set of fragments related by IS_A specialization links according to building rules.

Figure 6 proposes the IFO_2 structural schema for the lift example. It is made up of two fragments "Lift" and "Cage". They are related by an IS_A link through the represented type "Lift-Cage". The fragment of heart "Lift" has "Id-Number", "Load" (built up as an aggregation of "Max-Weight" and "Max-User" types) and "Lift-Cage" as properties. Since a lift serves several floors, we have a complex function from "Lift" to the simple type "Floor". For the fragments "Lift", the methods are "Up", "Down" and "Cage-Arrival". Among the methods of the fragment "Cage", "Stop" is triggered to stop the cage and "Open" to open the doors.

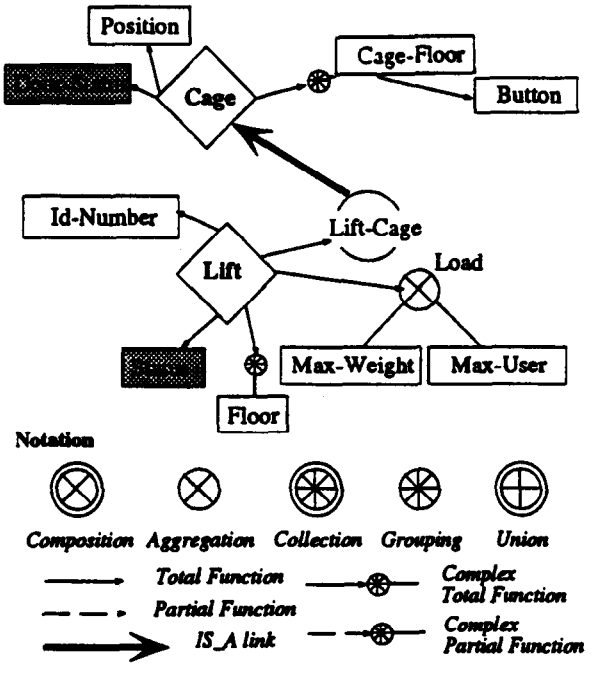

Figure 6: The "Lift" Structural Schema