PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

On the Evolution of Objects in a Logic Programming Framework

F. Nihan Kesim Marek Sergot

Department of Computing, Imperial College
180 Queens Gate, London SW7 2BZ, UK
fnk@doc.ic.ac.uk, mjs@doc.ic.ac.uk

Abstract

The event calculus is a general approach to the representation of time and change in a logic programming framework. We present here a variant which maintains a historical database of changing objects. We begin by considering changes to the internal state of an object, and the creation and deletion of objects. We then present separately the modifications that are necessary to support the mutation of objects, that is to say, allowing objects to change class and internal structure without loss of identity. The aims are twofold: to present the modified event calculus and comment on its relative merits compared with the standard versions; and to raise some general issues about object-orientation in databases which do not come to light if dynamic aspects are ignored.

1 Introduction

There has been considerable research on combining logic-based and object-oriented systems, and reasoning with complex objects. Many proposals have been put forward for incorporating features of object-oriented systems into logic programming and deductive databases [Abiteboul and Grumbach 1988, Zaniolo 1985, Chen and Warren 1989, Kifer and Lausen 1989, Dalal and Gangopadhyay 1989, Maier 1986, Bancilhon and Khoshafian 1986]. But opinions vary widely as to what are the characteristic and beneficial features of objects and comparatively little attention has been given to the dynamic aspects of objects. Yet change in internal state of an object as it evolves over time is often seen as a characteristic feature of object-oriented programming; and the ability of object-oriented representations to cope gracefully with change has often been cited as a major advantage of this style of representation. It is these dynamic aspects that we wish to address in this paper. We are not concerned with object-oriented *programming*, but with object-oriented representation of data in (deductive) databases. We address such problems as how objects change state, how deletion and creation of objects can be described and how an evolving object can change its class over time.

In order to avoid the discussion of destructive assignment, we formulate change in the context of a historical database which stores all past states of objects in the database. Historical databases are logically simpler than snapshot databases because change is then simply *addition* of new input. A snapshot of the historical database at any given time is an object-oriented database in the sense that it supports an object-based data model.

In this paper we present an object-based variant of the event calculus[Kowalski and Sergot 1986] which is a general approach to the treatment of time and change within a logic programming framework. We use this modified event calculus to describe changes to objects. The objectives of this paper are twofold: to present the object-based variant of the event calculus; and to raise some general issues about object-orientation in databases that we believe do not come to light if dynamic aspects are ignored. These more general points are touched upon in the course of the presentation, and identified explicitly in the concluding section.

In the following section we give a brief summary of the original event calculus. Section 3 presents the basic data model that is supported by the object-based variant. In section 4 we present this object-based variant and discuss how it can be applied to describe changes in objects. In section 5 we address the mutation of objects, where objects are allowed to change their classes during their evolution. We conclude the paper by summarising, and making some remarks about object-based representations in general.

2 The Event Calculus

The primitives of the event calculus are *events* together with some kind of temporal *ordering* on them, *periods* of time, and *properties* which are the facts and relationships that change over time. Events initiate and terminate periods of time for which properties hold. The effects of each type of event are described by specifying which properties they initiate and terminate. Given a set of events and the times at which they occurred, the event calculus derives (computes) which facts hold at which times. As an example, consider a fragment of a departmental database. An event of type

promote(X, New)

initiates a period of time for which employee X holds rank *New* and terminates whatever rank X held at the time of the promotion:

initiates(promote(X,New), rank(X,New)).
terminates(promote(X,New), rank(X,_)).

Given a fragment of data :

happens(promote(jim,assistant), 1986).
happens(promote(jim,lecturer), 1988).
happens(promote(jim,professor), 1991).

the event calculus computes answers to queries such as :

?- *holds_at(rank(jim,R), 1990).*
R=lecturer
?- *holds_for(rank(jim,lecturer), P).*
P=1988-1991

The original presentation of the event calculus showed how a computationally useful formulation can be derived from general axioms about the properties of periods. It gave particular attention to the case where events (changes in the world) are not necessarily reported in the order in which they actually occur. For the purpose of this paper, it is sufficient to consider only the simplest case, where the assimilation of events into a database is assumed to keep step with the occurrence of changes in the world, and where the times of all event occurrences are known. Under these simplifying assumptions, the event calculus can be formulated thus:

holds_at(R, T) ←
happens(Ev, Ts), Ts ≤ *T,*
initiates(Ev, R),
not broken(R, Ts, T).

broken(R, Ts, T) ←
happens(Ev, T*),*
Ts < *T** ≤ *T,*
terminates(Ev, R).*

We have omitted the clauses for *holds_for* which are similar. The interpretation of *not* as negation by failure in the last condition for *holds_at* gives a form of default persistence: property R is assumed to hold at all times after its initiation by event Ev unless there is information to the contrary.

The event calculus has been developed and extended in various different ways (see for instance [Sripada 1991, Eshghi 1988]). But what is important for present purposes is to stress that the underlying data model in all of these applications is relational. The *properties* that events initiate and terminate are facts like *rank(jim,professor)*. In database terms they are tuples of relations; in logic programming terms they are ground unit clauses or ground atoms or standard first order terms, depending on what is taken as the semantics of *holds_at*. A snapshot of the historical database at any given time is a relational database. In this paper we modify the event calculus in order to describe changes to a database which supports an object-oriented data model.

Before moving on to present this modification, we wish to make one further remark about the representation of events. One of the most common motivations for introducing object-oriented extensions to logic programming languages [Chen and Warren 1989, Ait-Kaci and Nasr 1986, M. Kifer and Wu 1990] is to overcome the restrictions imposed by the fixed arity of predicates and functors. These restrictions are particularly evident in the representation of events: Jim was promoted to professor in 1989, Jim was promoted from lecturer, Jim was promoted by his department in 1989 could all be descriptions of the same promotion recording different amounts of information about the event. In general, it is difficult or impossible to devise a fixed arity representation for events, because these representations cannot cope gracefully with the range of descriptions that can be expected even for events of the same type. (The philosopher Kenny refers to this phenomenon as the 'variable polyadicity' of events.) The standard way of dealing with 'variable polyadicity' is to employ binary predicates. Thus [Kowalski and Sergot 1986] represents events in the following style:

event(e1).
act(e1, promote).
object(e1, jim).
newrank(e1, prof).
happens(e1,1989).

Chen and Warren [Chen and Warren 1989] have developed this usage of binary predicates and have given it a formal basis. Their language C-logic allows the use of structured terms which can be decomposed into subparts. These terms are record-like tuples with

named labels. In the syntax of C-logic (also resembling the syntax of *LOGIN* [Ait-Kaci and Nasr 1986] and O-logic [Maier 1986]) the event *e1* can be described thus:

happens(event : e1[act ⇒ promote, object ⇒ jim, newrank ⇒ prof], 1989).

e1 is an identity which uniquely determines the event, and the labels are used to complete the specification of the event. Chen and Warren give a semantics to C-logic directly, and also by transformation to an equivalent first-order (Prolog) formulation that uses unary predicates for types and binary predicates for attributes. In this paper we use C-logic syntax as a convenient shorthand for describing events, and we exploit C-logic's transformation to Prolog by mixing C-logic and standard Prolog syntax freely. Thus we shall also write, for example,

event:e1[act⇒promote, object⇒ jim, newrank⇒prof].
happens(e1,1989).

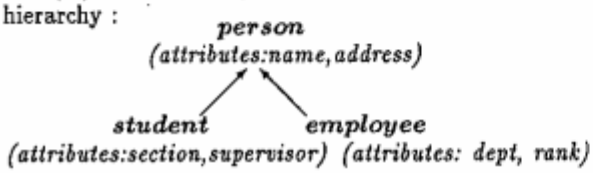
Chen and Warren's transformation to Prolog make all of these formulations equivalent.

3 The Data Model

Our objective in this paper is to focus attention on the dynamic aspects of objects. For this purpose, we take a very simple data model which exhibits only the most basic features associated with object orientation. As will be illustrated, this simple data model already raises a number of important problems; further elaborations of this data model are mentioned in the concluding section of the paper.

The basic building block of the model is the concept of an object. An object corresponds to a real world entity. Each object has a unique identity to distinguish it from other objects. The objects have attributes whose values can be other objects (i.e. their identities). We assume that all attributes are single-valued.

Objects are organized into class hierarchies, defined explicitly by *is_a* relationships among classes. A class denotes a set of object identities; the class-subclass relation (*is_a*) is the subset relation. A class describes the internal structure (state) of its instances by attribute names. The state of an instance is determined by the values assigned to these attributes. A subclass inherits the structure (attribute names) of its superclass(es). As an example consider the following class hierarchy :



The instances of the class *student* have the internal structure described by the attributes *name, address, section* and *supervisor*. Similarly the state of an *employee* instance is described by *name, address, dept* and *rank*. The class hierarchy is represented by *is_a* relations as:

is_a(student, person).
is_a(employee, person).

The relation between a class and its instances is represented by the *instance_of* relation. The instances of a class *C* are also instances of the superclasses of *C*. The *instance_of* relation can be represented thus:

instance_of(tom, student).
instance_of(mary, employee). etc.

together with

instance_of(X, Class) ←
is_a(Sub, Class), instance_of(X, Sub).

These definitions will be adjusted in later sections when we consider time dependent behaviour.

Multiple inheritance without overriding can be expressed by the *is_a* and *instance_of* relationships. This type of multiple inheritance causes no additional difficulty and is not mentioned again.

4 Object-Based Event Calculus

Database applications require an ability to model a changing world. The world changes as a result of the occurrence of events and hence it is very natural to describe such a changing world using a description of events. Given a description of events, it is possible to construct the state of the world using the the event calculus.

4.1 State Changes

One way of dealing with the evolution of an object over time (as suggested to us by several groups, independently) is to view the changing object as a collection of different though related objects. Thus, if we have an employee object *jim* in the database, which changes over time, *jim* at time t_1, *jim* at time t_2, *jim* at time t_3 are all different objects. Their common time-independent attributes are inherited from *jim* by some kind of 'part_of' mechanism. This approach has a certain appeal, but a moment's thought reveals it must be rejected for practical reasons. Each time an object is modified a new object is created. This new object becomes the most recent state of the object with a different identity. In this case, all other objects referring to the modified object should also be modified to refer to the new version. However updating them means creating other new objects in turn,

which results in an explosion in the number of objects in the database. In [M. Kifer and Wu 1990] a system of this type is described. They have to use equality in order to make certain denotations (i.e. object ids) in fact refer to the same object and provide some navigation methods through versions in order to get appropriate versions of an object.

The alternative is to have one object *jim* and to parametrize its attributes with times at which these attributes have various values. A state change in an object now corresponds to changing the value of any of its attributes. For instance if a person moves to a new place, the value of the address attribute changes; if an employee is promoted the rank attribute changes accordingly. Formulation of this idea in the spirit of the event calculus is straightforward. Instead of

happens(promote(jim,professor), 1991).

it is convenient to separate out the object that has been affected by the event :

happens(jim, promote(professor), 1991).

Now events are indexed by object; every object has associated with it the events that affected it. Events initiate and terminate periods of time for which a given attribute of a given object takes a particular value :

initiates(Obj, promote(NewRank), rank, NewRank).

Given a set of event descriptions which are indexed by object identities, the modified event calculus constructs the state of an object. We can ask queries to find out the value of an attribute of an object at a specific time or we can access the state of an object at any time by querying all of its attributes :

?- *holds_at(jim, rank, R, 1989).*
?- *holds_at(jim, Attr, Val, 1989).*

The following is the basic formulation of the object-based event calculus used to reason about the changing state of objects :

holds_at(Obj, Attr, Val, T) ←
happens(Obj, Ev, Ts), Ts ≤ *T,*
initiates(Obj, Ev, Attr, Val),
not broken(Obj, Attr, Val, Ts, T).

broken(Obj, Attr, Val, Ts, T) ←
happens(Obj, Ev, T*),*
Ts < *T** ≤ *T,*
terminates(Obj, Ev, Attr, Val).*

terminates(Obj, Ev, Attr, _)* ←
initiates(Obj, Ev, Attr, _).*

Informally, to find the value of an attribute of an object at time T, we find an event which happened before time T, and initiated the value of that attribute; and then we check that no other event which terminates that value has happened to the object in the meantime. The last clause for *terminates* is to satisfy the functionality constraint of the attributes. Since we are considering only single-valued attributes we can simply state that the value of an attribute is terminated if an event initiates it to another value. (The usage of the anonymous variable '_' in this clause is not a mistake).

The original event calculus can compute the periods of time for which a property holds. We can have the same facility for the attributes of objects. The following compute the periods of time for which an attribute takes a particular value :

holds_for(Obj, Attr, Val, (S - E)) ←
happens(Obj, Ev, S),
initiates(Obj, Ev, Attr, Val),
terminated(Obj, Attr, Val, S, E).

terminated(Obj, Attr, Val, S, E) ←
happens(Obj, Ev, E),
terminates(Obj, Ev, Attr, Val), S < *E,*
not broken(Obj, Attr, Val, S, E).

holds_for is used to find the period of time for which an attribute has a particular value. The time period is represented by its start (S) and end (E) points. We also require another clause for *holds_for* to deal with periods that have no end-point (i.e. an attribute is initiated but there is no event which terminated its value). This can be written in a similar style, which we omit.

Since objects are organized into classes, it is natural and convenient to structure the specification of the effects of a given event according to the class of object it affects. If an event is defined to affect the instances of a class, then the same event specification applies to the instances of subclasses. For example, consider a departmental database in which objects are organized according to the class hierarchy given in section 3. We can specify the effects of these events in the following way :

initiates(Obj,move(Address),address,Address) ←
instance_of(Obj,person).
initiates(Obj,promote(NewRank),rank,NewRank) ←
instance_of(Obj,employee).

The effects of changing the address are valid for all persons (i.e. all students and employees as well). However promotion is a type of event which can happen to employee objects only. In the formulation as presented here, it is possible to assert that an object of class person was promoted - but this event has no effect (does not initiate or terminate anything) unless the object is also an instance of class employee. An

alternative is to arrange for event descriptions to be checked and rejected at input if the class conditions are not satisfied. This alternative requires more explanation than we have space for; it is peripheral to our main points, and we omit further discussion.

We have discussed how event calculus can be used to describe changes to the values of attributes of objects. Apart from the events that cause state changes of existing objects, there are other kinds of events which cause the creation of new objects or deletion of objects.

4.2 Creation of Objects

The creation of a new object of a given class means adding new information about an entity to the database. In the real world being modeled, there are events which create new entities. Birth of a person, manufacturing of a vehicle or hiring a new employee are examples of such events. We can think of describing object creation by events whose specification will provide the necessary information about the initial state of the object.

For creation, we need to say what the class of an object is and specify somehow its initial state. In a practical implementation, generation of a unique identity for a newly created object can be left to the system; conceptually, all object identities exist, and the 'creation' of an object is simply assigning it to a chosen class. Assigning the new identity to the class initiates a period of time for which the new object is a member of that class. This makes it necessary to treat class membership as a time-dependent relationship. We introduce a new predicate *assigns* to describe instance addition to classes. For the time being we assume that once an object is assigned to a class it remains an instance of this class throughout its lifetime. Class changes are discussed separately in section 5.

We can handle creation of objects by specifying which events assign objects to which classes. We use the same event description to initialize the state of the object. As an example consider registration of a student *ali*, which causes the creation of a new student object in the database. The specification of the event and the necessary rules to describe creation are as follows :

```
event : e23 [act ⇒ register,
             object ⇒ ali,
             section ⇒ lp,
             supervisor ⇒ bob].

assigns(event:E[act⇒register, object⇒Obj],
             Obj, student).

initiates(Obj, E , section, S) ←
   event : E[act⇒register, object⇒Obj, section⇒S].
initiates(Obj, E, supervisor, S) ←
   event : E[act⇒register,object⇒Obj,supervisor⇒S].
```

The *assigns* statement is used to assign the identity of the object *Obj* to the class *student*; the *initiates* statements are used to initialize the object's state. Now the occurrence of the event is recorded by :

```
happens(e23, 1991).
```

To specify that the event has happened to the object *ali* we use the rule:

```
happens(Obj, Ev, T) ←
    happens(event:Ev[act⇒register, object⇒Obj],T).
```

Note that we have two *happens* predicates: one binary (for asserting that events happened at a given time), and one ternary (to index events by objects affected).

We have to notice also that creating a new object of class C, creates a new instance of the superclasses of C as well. There are several ways to formulate this. The simplest is to write:

```
assigns(Ev, Obj, Class) ←
            is_a(Sub, Class), assigns(Ev, Obj, Sub).
```

4.3 Deletion of Objects

There are two kinds of deletions that we are going to discuss in this paper. One is absolute deletion of an object where the object is removed from the database. The other one deletes an object from its class but keeps it as an instance of another class, possibly one of the superclasses. The second case is related to mutation of objects as they change class, which will be discussed in section 5.

For the purposes of this section, we assume that when an object is deleted it is removed from the set of instances of its class and the superclasses, and that all its attribute values are terminated. For example, if a person dies, all the information about that person is deleted from the database. We use a new predicate *destroys* to specify events that delete objects and write the following :

```
terminates(Obj, Ev, Attr, _) ← destroys(Ev, Obj).
```

This rule has the effect that all attributes *Attr* defined in the class of the object and also those inherited from superclasses are terminated. If an event destroys an object O which is an instance of class C, then that event removes O from class C and all superclasses of C.

There is one point to consider when deleting objects in object-oriented databases. If we delete an object x, there might be other objects that have stored

the identity of x as a reference. The deletion therefore can lead to dangling references. A basic choice for object-oriented databases is whether to support deletion of objects at all [Zdonik and Maier 1990]. We choose to allow deletion of objects and we eliminate dangling references by adding another rule for the *broken* predicate:

broken(Obj, Attr, Val, Ts, T) ←
happens(Val, Ev, T*),*
$Ts < T^* \leq T$,
destroys(Ev, Val).*

We obtain the effect that the value *Val* of the attribute *Attr* is terminated by any event which destroys the object *Val*.

4.4 Class Membership

As we create and delete objects the instances of a class change. Class membership, which is described by the *instance_of* relation, is a dynamic relation that changes over time. We can handle the temporal behaviour by adding a time parameter. We now have events that initiate and terminate periods of time for which an object O is an instance of a class C. The *instance_of* relation is affected when a new object is assigned to a class or when an object is destroyed. By analogy with *holds_at*, the following finds the instances of a class at a specific time :

instance_of(Obj, Class, T) ←
happens(Ev, Ts), $Ts \leq T$,
assigns(Ev, Obj, Class),
not removed(Obj, Class, Ts, T).

removed(Obj, Class, Ts, T) ←
happens(Obj, Ev, T*),* $Ts < T^* \leq T$,
destroys(Ev, Obj).*

With this time variant class membership we can ask queries to find the instances of a class at a specific time. For example:

?- *instance_of(Obj, employee, 1980).*

We can also write the analogue of *holds_for* to compute periods, which we omit here.

In the example we have been using, we have represented the rank of an employee object by including an attribute *rank* whose value might change over time. But suppose that instead of using an attribute *rank*, we had chosen to divide the class of employees into various distinct subclasses:

is_a(lecturer, employee).
is_a(professor, employee).

It is at least conceivable that this alternative representation might have been chosen, assuming that all employee objects have roughly the same kind of internal structure. Is the choice between these two representations simply a matter of personal preference? Not if we consider the evolution of objects over time. The first representation allows for change in an employee's rank straightforwardly, since this just changes the value of an attribute. The second does not, since no object can change class in the formulation of this section. The only way of expressing, say, a promotion from lecturer to professor, is by destroying (deleting) the lecturer object and creating a new professor object. But then how do we relate the new professor object to the old lecturer object, and how do we preserve the values of unchanged attributes across the change in class? In the next section we will examine the problem of allowing the class of an individual object to change.

5 Mutation of Objects: Changing the Class

The ability to change the class of an object provides support for object evolution [Zdonik 1990]. It lets an object change its structure and behaviour, and still retain its identity. For instance, consider an object that is currently a *person*. As time passes it might naturally become an instance of the class *student* and then later an instance of *employee*. This kind of modification is usually not directly supported by most systems. It may be possible to create another object of the new class and copy information from the old object to it, but one loses the identity of the old object.

We want to describe this kind of evolution by event specifications. For example graduation causes a student to change class. If we delete student *ali* from class student, then he will lose all the attributes he has by virtue of being a student, but retain the attributes he has by virtue of being a person. The effects of this event can be described by removing *ali* only from class student and terminating his attributes selectively. The attributes that are going to be terminated can be derived from the schema information. For dealing with this type of class change we use a new predicate *removes* in place of the predicate *destroys* of section 4.3:

removes(event: Ev[act ⇒ graduate, object ⇒ Obj],
Obj, student).
terminates(Obj, E, Attr, _) ←
event:Ev[act⇒graduate, object⇒Obj],
attribute(student, Attr).

The clauses for the time-dependent *instance_of* relation must be modified too, to take *removes* into account:

$removed(Obj, Class, Ts, T) \leftarrow$
$\quad happens(Obj, Ev^*, T^*), Ts < T^* \leq T,$
$\quad removes(Ev^*, Obj, Class).$

The graduation of the student *ali* corresponds to moving him up the class hierarchy. Now consider hiring *ali* as an employee. This will correspond to moving down the hierarchy. The specification of an event causing such a change will likely include values to initialize the additional attributes associated with the subclass. So the effects of hiring *ali* will be to assign him to the employee class and initiate his employee attributes. The event might be:

$event : e21[act \Rightarrow hire,$
$\quad object \Rightarrow ali,$
$\quad dept \Rightarrow cs,$
$\quad rank \Rightarrow lecturer]$

And we can declare the following:

$assigns(event{:}E[act{\Rightarrow}hire, object{\Rightarrow}Obj], Obj, employee).$
$initiates(Obj, E, dept, D) \leftarrow$
$\quad event{:}E[act{\Rightarrow}hire, object{\Rightarrow}Obj, dept{\Rightarrow}D].$
$initiates(Obj, E, rank, R) \leftarrow$
$\quad event{:}E[act{\Rightarrow}hire, object{\Rightarrow}Obj, rank{\Rightarrow}R].$

Note that in changing class first from student to person, then from person to employee, *ali* retains all the attributes he has as a person.

We have described this class change by two separate events: graduation and hiring. We can also imagine a single event which would cause an object to change its class from student to employee directly, say *hire-student* event. We could then describe the changes using the description of this event:

$removes(event: E[act \Rightarrow hire\text{-}student,$
$\quad object \Rightarrow Obj], Obj, student).$
$assigns(event: E[act \Rightarrow hire\text{-}student,$
$\quad object \Rightarrow Obj], Obj, employee).$

The initial values of the additional attributes will again be given in the event specification. As in the case of having two separate events, we have not lost the values of the attributes as a person, and we have not removed the object from class person.

We have illustrated three kinds of class changes: changing from a class C to a direct superclass of C, changing from C to a direct subclass of C and changing from C to a sibling class of C in the hierarchy. In general, changing an object from class *C1* to class *C2* involves removing from *C1* and assigning to *C2* and specifying in the event description how the initial values of *C2* attributes are related to the values of old *C1* attributes.

6 Concluding Remarks

We have presented a variant of the event calculus which maintains an object-based data model where the standard versions maintain a relational one. Section 4 considered state changes of objects in this framework, and the creation and deletion of objects. Section 5 discussed the modifications that are necessary to support also the mutation of objects - change of an object's class and its internal structure without loss of its identity.

There are other object-oriented features that can usefully be incorporated into the object-based data model. Removing the restriction that attributes are all single-valued causes no great complication. We are currently developing other extensions, such as the inclusion of methods in classes for defining the value of one attribute in terms of the values of other attributes, and we are investigating what additional complications arise when the schema itself is subject to change.

In object-oriented terminology, event types - like promote, change-address, and so on - correspond to methods: their effects depend on the class of object that is affected; the predicates *initiates* and *terminates* for attribute values, and *assigns*, *destroys* and *removes* for objects and classes are used to implement the methods (they would be replaced by destructive assignment if we maintained only a changing snapshot database). Of course, execution of this event calculus in Prolog does not yield an object-oriented style of computation. At the implementational level, objects are not clustered (except by Prolog's first argument indexing), and the computation has no element of message-passing. The implementation and the computational behaviour can be given a more object-oriented flavour by using for example the techniques described by [Chen 1990] for C-logic, or the class templates of [McCabe 1988]. We are currently investigating what added value is obtained by adjusting these implementational and computational details.

The object-oriented version of the event calculus offers some (computational) advantages over the standard relational versions, that we do not go into here for lack of space. Whatever the merits of the object-based variant of the event calculus, we believe that its formulation forces attention to be given to important aspects of object-orientation that are otherwise ignored. We limit ourselves to two general remarks:

1) In the literature, the terms *type* and *class* are often used interchangeably. Sometimes type is used in its technical sense, but then it is common to see illustrative examples resembling 'Mary is of type stu-

dent'. If we consider the dynamics of object-oriented representations, then these examples are either badly chosen or the proposals are fundamentally flawed. 'Mary' might be a student now but this will not hold forever. We could surely not contemplate an approach where an update to a database requires a change to the type system, and hence to the syntax of the representational language. These remarks do not apply to object-oriented programming where there is no need to make provision for updates that change the type of an object.

The static notion of a type corresponds to the treatment of a class we presented in section 4: an object may or may not exist at a given time, but when it exists it is always an instance of the same class. If we wish to go beyond this, to allow objects to mutate, then a dynamic notion of class is essential. This is not to say that types have no place in object-oriented databases. A student can become an employee over time, but a student cannot become a filing cabinet, and a filing cabinet cannot become an orange. Both static types and dynamic notions of class are useful. The consideration of the dynamics of objects - how they are allowed to evolve over time - suggests one immediate and simple criterion for choosing which notion to use: the type of an object cannot change.

2) In section 4.3, we assumed that all attributes of an object are terminated when the object is destroyed; in section 5, removal of an object from the class C terminates all attributes the object has by virtue of being an instance of the class C. The reasoning behind this assumption is this: attributes are used to represent the, possibly complex, internal state of an object. When an object ceases to exist, it is not meaningful to speak any more of its internal state. Of course, some information about an object persists even after it ceases to exist. It is still meaningful to speak of the father of a person who has died, but it is not meaningful to ask whether this person likes oranges or is happy or has an address. The development of these ideas suggests that we should distinguish between what we call 'internal attributes' and 'external relationships'. Internal attributes describe the state of a complex object, and they cease to hold when the object ceases to exist or ceases to be an instance of the class with which these attributes are associated. External relationships continue to hold even after the object ceases to exist. We are being led to a kind of hybrid data model together with some tentative criteria for choosing between representation as attribute and representation as relationship with other objects. The analysis given here is rather superficial, but it indicates the general directions in which we are planning to pursue this work.

Acknowledgements. F.N. Kesim would like to acknowledge the financial support by TUBITAK, the Scientific and Research Council of Turkey.

References

[Abiteboul and Grumbach 1988] S. Abiteboul and S. Grumbach. COL : A logic-based language for complex objects. In *International Conference on Extending Database Technology- EDBT'88*, pages 271–293, Venice, Italy, March 1988.

[Ait-Kaci and Nasr 1986] H. Ait-Kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *The Journal of Logic Programming*, 1986.

[Bancilhon and Khoshafian 1986] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proceedings of the 5th ACM-SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 53–59, Cambridge, Massachusetts, March 1986.

[Chen 1990] Weidong Chen. *A General Logic-Based Approach to Structured Data*. PhD thesis, State University of New York at Stony Brook, 1990.

[Chen and Warren 1989] W. Chen and D. Warren. C-logic of complex objects. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, 1989.

[Dalal and Gangopadhyay 1989] M. Dalal and D. Gangopadhyay. OOLP: A translation approach to object-oriented logic programming. In *The First International Conference on Deductive and Object-Oriented Databases*, pages 555–568, Kyoto,Japan, December 4-6 1989.

[Eshghi 1988] K. Eshghi. Abductive Planning with the Event Calculus. In *Proc. 5th International Conference on Logic Programming*,1988.

[Kifer and Lausen 1989] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the ACM-SIGMOD Symposium on the Management of Data*, pages 134–146, 1989.

[Kowalski and Sergot 1986] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[M. Kifer and Wu 1990] G. Lausen M. Kifer and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical report, Department of Computer Science, SUNY at Stony Brook, June 1990.

[Maier 1986] D. Maier. A logic for objects. In *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington D.C., August 1986.

[McCabe 1988] F.G. McCabe. *Logic and Objects: Language Application and Implementation.* PhD thesis, Department of Computing, Imperial College, 1988.

[Sripada 1991] S. M. Sripada. *Temporal Reasoning in Deductive Databases.* PhD thesis, Department of Computing, Imperial College, 1991.

[Zaniolo 1985] C. Zaniolo. The representation and deductive retrieval of complex objects. In *Proceedings of Very Large Databases*, page 458, Stockholm, 1985.

[Zdonik 1990] S. B. Zdonik. Object-oriented type evolution. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 277–288. ACM Press, 1990.

[Zdonik and Maier 1990] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*, chapter 4, page 239. Morgan Kaufmann, 1990.