Implementing an Object-Oriented Deductive Database Using Temporal Reasoning

Nihan Kesim
Bilkent University

Marek Sergot
Imperial College

A general approach for temporal reasoning, the event calculus, has been modified and applied to the development of a historical deductive object base. The event calculus is a theory of time in first-order logic augmented with negation as failure. It is shown how an object-based variant of the event calculus may be used for representing changes to the states of objects. We first present the formulation and describe the maintenance of a historical object-oriented database by the use of events. The resulting formalization can be executed as a logic program. We then discuss the implementation of a practical database system based on the theory we develop. The additional detail needed to develop a system of realistic scale is outlined. The aim is to present the object-based event calculus as a unifying framework for the implementation of a deductive and object-oriented database system.

Deductive object-oriented databases are the most recent research area falling within the intersection of logic and databases. They aim at providing both database designers and users with a modelling tool which is semantically richer than the relational data model as well as using logic as a tool for the formalization of both the static and the dynamic aspects of databases.

Conventional deductive databases, i.e. those which have the relational data model as the underlying data model, are well-understood and several deductive database system prototypes have been implemented (Ceri, 1990; Chimenti, 1990; Ramamohanarao, 1988; Vieille, 1991) . The transition from relational databases to deductive databases was smooth and natural because the relational data model has a simple and well-defined expression in terms of first-order logic (Gallaire, 1978). Therefore coupling a relational database to a logic programming language, such as Prolog, was natural and the result was an enhancing of the expressiveness of relational databases with deduction.

The perspective is totally different if object-oriented databases are considered instead. There is a conceptual mismatch between logic programming languages and the notion of object as imported from object-oriented programming languages. This conceptual mismatch raises several difficult problems that must be tackled for deductive object-oriented databases to be seen as a step forward. Nevertheless, deductive object-oriented databases have become a focus of interest because current deductive databases are based on the relational data model, and as a consequence they inherit the modeling shortcomings of the relational model for supporting non-business applications such as design activities and graphic data manipulation.

There have been many proposals on combining the deductive and object-oriented approaches. Some of the current work take deductive databases as the basis and extend the existing systems with some subset of object-oriented features (Abiteboul, 1988; Chimenti, 1990; Kuper, 1987; Zaniolo, 1985). Others take object-orientation as the basis and logic as the framework, and try to formalize object-oriented data modelling (Ait-Kaci, 1986; Chen, 1989; Kifer & Wu, 1989; Kifer & Lausen, 1989). There is also another stream of work which approaches the problem from a programming language point of view. The aim is to combine object-oriented programming and logic programming by extending the existing declarative languages (e.g. Prolog, Datalog) with object-oriented notions like methods, message passing and inheritance (Dalal, 1989; Fukunaga, 1986; McCabe, 1988). A detailed discussion of all these different approaches is presented in Fernandes (1993); Kesim (1994, 1993).

Manuscript originally submitted February 17, 1995; Revised August 15, 1995; Accepted August 29, 1995 for publication.

Current proposals mainly deal with structural aspects of objects by defining semantics of basic object-oriented features in a logical framework. However, there is little work on representing and dealing with the dynamic aspects of objects, such as state changes of objects. It is unclear how changes to object states, creation and deletion of objects, and changing the class of objects can be described in a deductive and object-oriented framework. In this paper we take a different approach by combining two kinds of models. We consider not only the structural aspects of objects but also their dynamic aspects. We use an object model to represent the structure of objects, and use a dynamic model to represent the behaviour of objects over time.

We have developed the object-based event calculus (OEC in short) to combine these two views of modelling the system. The OEC is a variant of the event calculus of Kowalski and Sergot (Kowalski, 1986), which is a theory for reasoning about time and change within a logic programming framework. The OEC is used to describe various temporal aspects of objects. Change is formulated in the context of a historical database which stores implicitly all past states of objects in the database. It is possible to determine which objects "exist" at which times and derive any past state of an object using the axioms of the OEC and a given set of event descriptions. It is also possible to keep and reason about different versions of an object at a time.

The formulation of the OEC has been presented in Kesim (1992) and the main idea has been extended to explore other temporal aspects of objects and classes, such as versioning of objects and schema evolution (Kesim & Sergot, 1993; Kesim, 1994). In this paper we mainly discuss the implementation aspects of a practical system based on the proposed framework. A direct coding of the OEC as a logic program has some efficiency problems due to the reasoning behind the theory. It generates a large search space and redundant computations in the evaluation of queries. These problems can be solved by a more sophisticated implementation which makes use of the derived conclusions. Instead of deriving the states of objects only when they are needed, all known facts about the objects can be stored in an extensional database and query evaluation can be performed on this database. In this paper we present such an implementation of the OEC to be used as a basis for a practical temporal deductive object base. We describe a conceptual architecture of such a system where the persistent storage of objects can be maintained as a relational database and an object-oriented layer can be built on top as an interface.

The database which keeps the history of objects will be very large in size. The problems encountered in implementing such a system is closely related to the problems in implementing temporal relational databases. Many efficient storage and indexing techniques have been proposed for temporal relational databases (Snodgrass, 1986; Tansel, 1993). These techniques can be applied in the implementation of the persistent object store. For instance in Jensen (1991), an implementation model is presented for the standard relational data model extended with transaction time where updates to relations are entered as time-stamped change requests into backlogs. The backlogs of Jensen et al. are very similar to the structure of our object store. However, the actual application of these techniques to the OEC is left as a future work.

The rest of the paper is organized as follows. Section 2 defines the basic modelling framework in which events and objects are related. In Section 3 we summarize the formulation of the object-based event calculus and discuss how it can be applied to describe different temporal aspects of objects including versioning of objects. Section 4 discusses the computational issues related to the implementation and identifies some efficiency considerations. Section 5 gives an overview of the architecture of a practical database system based on the OEC. In Section 6 we discuss the usage of the lemma generation approach in implementing an external database to store objects. We describe an efficient query evaluation technique and discuss the maintenance of the underlying database every time a new event is recorded into the system. Section 7 presents an evaluation of the current implementation and we conclude the paper in Section 8.

Events and Objects

We combine two models to describe a deductive object-oriented database: a dynamic model and an object model. The dynamic model is used to describe those aspects of the system concerned with time and change. The object model, on the other hand, describes the static structure of objects in a system - their identity, their relationships to other objects, their attributes, and classes.

The major dynamic modelling concepts are events, which represent external stimuli, and states, which represent values of objects. An event is something that happens at a point of time, such as user depresses left button or flight 123 departs from Chicago. An event has no duration. Of course, nothing is really instantaneous; an event is simply an occurrence that is fast compared to the granularity of the time scale of a given abstraction. A state, however has a duration; it occupies an interval of time. Events and states are duals of one another; an event separates two states, and a state separates two events. Thus a state is assumed to persist into the future until it is terminated by an event.

The state of objects is described by attributes which hold data values or identities of other objects. A state specifies the response of the object to input events. A state change corresponds to changing the value of any of the attributes of the object. For instance, if a person moves to a new place, the value of the address attribute changes; if a car is painted, the color attribute changes accordingly. Thus, the state of an object depends on the past sequence of events that affects it.

We take a simple object model to describe the state of

objects. The model supports only a subset of the well-known object-oriented features: object identity, (single-valued) attributes, classes and the is_a relationship. The object model is based on the relational semantics of complex objects given in the transformation of C-logic into first-order logic (Chen, 1989). We view an object as a named collection of attribute-value tuples. As in Kifer & Wu (1989), we use individual terms to denote object identities. A term representing the object identity is composed of function symbols, constants and variables in the usual way. For example, jim, X, version(chip, N), spouse(X, Y) can be terms denoting object identities. The explicit construction of the identities can be given by the user or be automatically generated by the system.

Objects are grouped into classes which provide abstraction. The relation between an object and its class is represented by the instance_of relation. The instances of a class can change in time, as there can be new objects added into the database or existing objects can be deleted from the database. It is also possible that an object may change its class during its evolution. This temporal behaviour of the instance_of relation is also described by events. Events initiate and terminate periods of time for which an object is an instance of a class.

Classes are organized into class hierarchies. A class hierarchy is defined explicitly by is_a relationships among classes. The class-subclass relation (is_a) is the subset relation. That is, the set of objects represented by a class includes all the objects belonging to the subclass(es) of that class. The notion of inheritance is limited to only the subset relation between classes and monotonic inheritance of attribute names. We do not support overriding of an attribute name by a subclass. Subclasses should introduce attribute names that do not conflict with those inherited from the superclasses.

We consider only this basic object model for the purposes of this paper, but the model can be easily extended to include more object-oriented features like multi-valued attributes, derived-attributes (i.e., methods or rules) and multiple inheritance (Kesim, 1994, 1993). These extensions do not cause any additional difficulty or major change to neither the theory nor the implementation of the system. In order not to diverge the scope of the paper, these extensions are not mentioned again in this paper.

The dynamic model which expresses the relation between events and object states is based on the event calculus. The event calculus uses general rules to derive a new relationship that holds as a result of an event and it associates time periods with relationships (Kowalski, 1986). In the event calculus change is represented by adding descriptions of new events occurring at particular times to the database which therefore comprises a historical record of changes in the application model. Its main intended application is the representation of events in database updates and discourse representation. The approach is closely related to McCarthy's situation calculus (McCarthy, 1969) and Allen's interval temporal logic (Allen, 1984) (see Sadri, 1987 for a comparison).

The object-based event calculus is based on a special, simplified, asymmetric case of the event calculus, where periods of time are assumed to persist only into the future. In the OEC, instead of general relationships, the values of the attributes of objects are derived as a result of a given set of event descriptions. In the formulation of the OEC we assume that:

- no erroneous information is recorded in the database. Thus we neither need to consider revision of the histories of the objects nor deal with the integrity constraints over the states of the database.
- the events recorded into the database contain information about a real world event. The event occurrence time is in fact the valid time. All relevant events are recorded.

There are certain special characteristics of these assumptions. First of all the events are recorded in the database in the order in which they occur. Second, the database always contains a complete record of all the events that have occurred (i.e., no incomplete information). For these reasons we will deal with periods of time which persists forwards into the future only.

Formulation of the OEC

In the OEC knowledge is formalized in terms of events and the attribute values which they initiate and terminate. For example, "John is promoted to professor'' might be described as an event which initiates the value professor for the attribute rank and terminates whatever rank John held at the time of the promotion.

There are three levels of knowledge that can be distinguished in the OEC: particular event descriptions such as John's promotion; domain dependent knowledge about what events affect what attributes; and domain independent knowledge concerning the general properties of the system.

Domain Independent Knowledge

In this section we first present the axioms of the OEC which are used to derive object histories and then we present the extensions of the axioms which are used to derive versions of objects.

Object State Histories. The following are the basic domain independent rules of the object-based event calculus which are used to derive the value of an attribute of an object at a particular time:

holds _at(Obj, Attr, Val, T)←happens(Ev, Ts), Ts ≤ T,
initiates(Ev, Obj, Attr, Val),
not broken(Obj, Attr, Val, Ts, T).
broken(Obj, Attr, Val, Ts, T) ←happens(Ev*, T*),
Ts < T*≤ T,
terminates(Ev*, Obj, Attr, Val).

The first rule expresses the assumption that the attribute Attr of object Obj holds the value Val at time T if a prior event Ev initiates value Val and it cannot be shown (the not denotes negation by failure) that Val is interrupted between the occurrence of Ev and T. The second rule states that Val is interrupted between Ev and T if there is an intervening event Ev* which terminates Val.

Together the axioms express default persistence: once initiated, a value continues to hold unless an event is known to terminate it. The domain independent rules can be extended in many ways, for example by including: start and end points of the intervals for which an attribute holds a particular value; persistence of relations backwards, as well as forwards in time (Kowalski, 1986); transaction time (Sripada, 1991). Such extensions are not needed for our presentation, but their feasibility is a strong argument for the expressive power of the event calculus.

Other domain independent rules deal with describing the instances of a class at particular times, and creation and deletion of objects. The time dependent behaviour of class membership is modeled by parameterizing the instance_of relation with times. This relation is affected when a new object is assigned to a class or when an object is destroyed or when an object changes class. By analogy with holds_at, the following computes the instances of a class at a specific time :

instance_of(Obj, Class, T)←happens(Ev,Ts), Ts≤ T,
assigns(Ev, Obj, Class),
not removed(Obj, Class, Ts, T).

removed(Obj, Class, Ts, T) ←happens(Ev*, T*),
Ts < T*≤ T,
removes(Ev*, Obj).

removes(Ev, Obj, _) ← destroys(Ev, Obj).

Here the predicates assigns and removes are the analogues of initiates and terminates respectively and they are used to start or end a period of time for which an object is an instance of a class. When an event is defined to create a new object, it assigns the object to its class. When an event causes a class change, then it removes the object from one class and assigns it to another. And object deletion is described by the predicate destroys.

Creating a new object of class C, creates a new instance of the superclasses of C as well. This subset relation is expressed by the following rule:

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

Finally the following domain independent rules are used to handle single-valued attributes, object deletion and possible dangling references caused by deletion of objects:

terminates(Ev, Obj, Attr, _)← initiates(Ev, Obj, Attr, _).
terminates(Ev, Obj, Attr, _)← destroys(Ev, Obj).
terminates(Ev, Obj, Attr, Val)← destroys(Ev, Val).

The first clause is used to satisfy the functionality constraint of single-valued 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.

When an object is deleted all its attribute values are terminated. The second rule expresses this effect of the events which destroy objects. With this rule not only the attributes defined in the class of the object but also those inherited from superclasses are terminated.

When an object x is deleted, there might be other objects that refer the identity of x with some attribute. The deletion therefore can lead to dangling references. Such references are eliminated by the third rule which has the effect that the value Val of the attribute Attr is terminated by any event which destroys the object Val.

Versions of Objects. Object versioning is an essential mechanism for design (or planning or engineering) database applications. There are various approaches for object versioning in object-oriented databases (Beech, 1988; Bjornerstedt, 1989; Chou, 1986; Klahold, 1986). In design databases, versions are maintained for both assembly objects and component objects. Version histories are typically maintained in a hierarchical (e.g., branching time) ordering rather than a linear one. Child versions represent design derivatives from a parent version, and sibling versions represent design alternatives. The ability to keep all past states of an object facilitates the description of distinguishable versions of objects over time. In the following we show a versioning model for temporal objects in the OEC framework.

Versions are objects which are derived from existing objects as a result of some requirements and modifications. Versions are closely related to their parent versions, but they are still different objects and they must be uniquely identifiable. We use first-order terms to uniquely identify the versions of an object. For instance the first version of an object o will be v(o,1), the second v(o,2), the n th v(o,n) and so on. There might be versions of versions. The same naming convention can be used to identify the new versions. For instance, the first version of the object v(o,1) will be referred to as v(v(o,1),1), the second v(v(o,1),2) and so on.

The basic idea to represent an object having several versions at a time is to keep parallel histories for the object. Each history is identified by a version identity. Each version has its own history starting from its creation time. At its creation time, the version, as its initial state, has the same state as the parent object except the values of the attributes which might be affected by the version creating event. Once a version created, it can be updated, deleted or versioned like any other object in the database.

Creation of a version is described by events. In keeping the state history of objects, every event which causes a change in the value of the object's attribute creates a new state of the object, which can be viewed as a new version. In versioning not every event is considered as a version-creating event. Only certain events can cause the creation of identifiable versions. Some attributes can be classified as version-significant attributes, whose update would force the creation of a new version bearing the modified value of that attribute. Events that are specified as having effects on these attributes can be defined to be version-creating events and their effects are specified by the predicate creates_version.

The following is the extended formulation of holds_at to derive the states of versions of objects.

```
holds_at(Vid, Attr, Val, T)←
        happens(Ev, Tc), Tc ≤ T,
        creates_version(Ev, Vid),
        ( happens(Ev*, T*),
          Tc≤ T* ≤ T,
          initiates(Ev*, Vid, Attr, Val),
           not broken(Vid, Attr, Val, T*, T) )
        or
        ( prev_version(Vid, Oid),
          holds_at(Oid, Attr, Val, Tc) ).
```

We first find the creation time Tc of the version object Vid. We then check whether the attribute is changed by an event affecting the version object directly or by an event affecting the parent object before the version creation time. The first condition in the disjunction checks if there is an event initiating the attribute in the version object; the second condition is used to find the value of the attribute in the parent object at the time of version creation.

This formulation can be used to navigate through the versions easily, terminating when the recursion reaches the root object. This modified holds_at can also be used to reason with the state of objects without any versions, provided that the event creating the object is defined to be a version-creating event as well. There are some other version models which can be described within the OEC, but the details of these extensions are presented and discussed elsewhere (Kesim & Sergot, 1993).

Domain Dependent Knowledge

Domain dependent knowledge is expressed by rules which describe the effects of specific events. These rules are used to describe which attributes are initiated and terminated by which events encountered in the domain. In an application domain we need to:

1. describe the possible event types encountered in the domain categorized according to the class hierarchy.
2. describe the attributes of the objects initiated and terminated by these events.

In describing the effects of an event we separate out the object that has been affected by the event and use the predicates initiates, terminates, assigns, removes and destroys. For instance, the following rule describes the effect of a promotion event on the rank attribute of an employee object jim:

```
initiates( Ev, Obj, rank, NewRank) ← act( Ev, promote),
        object( Ev, Obj),
        rank( Ev, NewRank).
```

In order to describe object creation, the event which causes the creation of the object specifies the class of the new object and also its initial state. For example in a library database, the event of acquiring a book creates a new object in the book class and initiates the title attribute. Thus we write:

```
assigns( Ev, B, book)←
      act ( Ev, acquire), book ( Ev, B).
initiates( Ev, B, title, T)←
      act ( Ev, acquire), book ( Ev, B), title( Ev, T).
```

In addition to the title attribute, other attributes such as author, classification number etc. can also be specified in the event description. Here the variable B represents the identity of the book. Suppose a book's history is the time from its acquisition to its sale. So when a book is sold it is deleted from the database and deletion of an object is described by the predicate destroys:

```
destroys( Ev, B)← act( Ev, sale), book ( Ev, B).
```

As regards the class changes, the following rules illustrate the domain specific knowledge to describe mutation of objects. For instance, if class person is the superclass of classes student and employee, the event graduation will remove a student from the student class, leaving him/her as the instance of person class only. And the event of hiring will change the class of an object from person to employee.

```
removes( Ev, S, student)←act( Ev, graduate),
      student_of( Ev, S).
assigns( Ev, P, employee)←act( Ev, hire),
      person_of( Ev, P).
```

When the object changes class from student to person, all the attributes it has by virtue of being a student must be terminated, but the values of the attributes by virtue of being a person should be retained. This is described by the domain independent rule:

```
terminates( Ev, Obj, Attr, Val)← removes( Ev, Obj, Class),
```

attribute(Class, Attr).

A class change from person to employee necessitates the initialization of the additional attributes introduced in the employee class. The new values for these attributes can be specified in the event description. For instance:

initiates(Ev, P, salary, S)← act(Ev, hire), person(Ev, P),
salary(Ev, S).

Here we illustrated moving the object only one level up and down the class hierarchy. The treatment of the general case, i.e. changing the class of an object to an arbitrary class in the hierarchy has been discussed separately in (Kesim, 1992; 1994).

Finally we consider versioning of objects. As stated earlier, the effects of version-creating events are specified with the predicate creates_version which is used to mark the occurrences of such events in the object history and also to generate a unique identity for the version. For example, consider the design of a VLSI chip. Different versions of the chip may be derived, say to reduce the chip size or reduce the power consumption etc. Every time a "reconfigure" operation is performed, a new version is assumed to be created. The following rule is used to describe the situation:

creates_version(Ev, v(C, N))←
event: Ev[act ≤ reconfigure, chip≤ C, number ≤ N].

The functional term v(C, N) is the identity of the new version where N is an integer value denoting the version number. The variable C denotes the object whose version is being created.

Another point to consider in creating versions is to assign the new versions to classes. Here we take a simple approach by assuming that versions are also instances of the class to which their parent version belongs. Continuing with our example, the event of reconfiguring a chip assigns the new version to the class VLSI-chip:

assigns(Ev, v(C, N), VLSI-chip)←
event : Ev[act≤ reconfigure, chip≤ C, number≤ N].

Case Specific Knowledge and Queries

The case specific knowledge consists of the event descriptions entered into the system. As illustrated above, we have used binary predicates in the representation of events. In general it is difficult or impossible to devise a fixed-arity representation for event descriptions. Because fixed-arity representations cannot cope gracefully with the range of descriptions that can be expected even for events of the same type. Therefore, we use a binary predicate representation of events as suggested in Kowalski (1979). For example, the following events might be entered for a particular patient during hospitalization:

event(e1). event(e2).
act(e1, interview). act(e2, arteriogram).
patient(e1, p1). patient(e2, p1).
diagnosis(e1, gastrit). diagnosis(e2, occlusive_arterial).

The actual occurrences of the events are recorded into the database by the predicate happens:

happens(e1, 9/3/93).
happens(e2, 9/10/93).

Given a deductive database along the above lines, 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. For example, the following queries can be used to derive the state of a patient object jim at various times.

?- holds_at(jim, diagnosis, D, 9/5/93).
?- holds_at(jim, Attr, Val, 10/2/93).

Likewise the class of objects can be queried using the instance_of predicate:

?-instance_of(jim, C, 1983).
?-instance_of(Obj, patient, 1993).

In object-oriented terminology, the specification of how events—like promote, interview, sale - affect the state of objects correspond to methods: their effects depend on the class of object that is affected. The predicates initiates and terminates for attribute values, and assigns, removes and destroys for objects and classes are used to implement the methods. Of course the execution of this event calculus in Prolog does not yield an object-oriented style of computation, but conceptually we can consider the specification of events as methods that modify object states.

Implementation Issues

It is our aim to employ the OEC in the design and implementation of a realistic size database system. In this section we discuss the prospects of achieving this aim, and we consider some additional tools which would be needed for practical applications.

The OEC can be implemented in different programming languages in different ways. It can be translated more or less directly into a Prolog program; it can be implemented in L&O which is an object-oriented and logic programming language (McCabe, 1988); or a suitable algorithm can be constructed to perform the same task in a procedural language. In any of these choices, when we consider an efficient implementation, we encounter two serious problems. The first one is related to

searching the database efficiently. Due to the reasoning formalized by the event calculus, a large search space is generated to find the relevant events in solving a query. This is not a severe problem in small examples but it clearly becomes very significant in large-scale database applications. After finding a candidate event, it is necessary to establish that no terminating event for that relationship has occurred in the meantime. In the worst case this requires searching the complete record of event occurrences again, because of the negation in the formulations. This problem is significant even when there is a small number of event types but a large number of event occurrences in the database.

The second problem is the re-computation of the same facts over and over again not only when the same query is posed repeatedly to the database, but also within a single query execution. All the search and computation will be repeated without considering the previous derivations at all. This problem can lead to very severe redundancies in the computation, which are significant even in small applications.

These problems have been discovered in the implementations of the original (relational) event calculus and various solutions have been devised to overcome them. Many of these techniques can be adopted for improving the execution of the OEC as well. The main solutions that have been devised to overcome these implementation problems are indexing methods and lemma generation techniques.

Indexing

There are basically two kinds of indexing. One is temporal indexing based on the times at which events happen. The other indexing is structural which is based on searching the domain specific rules. Either indexing scheme can help reducing the search space considerably (Shanahan, 1992).

We illustrate the use of indexing by reference to the execution of holds_at, specifically for the task of determining the value of a given attribute of a given object at a given time. Exactly similar points can be made for more general holds_at queries, and for the class membership analogue instance_of.

Intuitively, if we are interested in establishing the value of an attribute at time t, then it is unnecessary (in this version of the event calculus) to consider any events occurring later than time t. It has been shown (Indiketiya, 1992) that even a simple indexing scheme which just segments the time line and stores each segment separately (possibly on some secondary storage) can make substantial improvements in the speed of query evaluation; more importantly for database applications, performance does not degrade appreciably for very large numbers of event occurrences. These benchmarks have been performed for the relational versions of the event calculus (with some restrictions) but this simple indexing technique would apply equally to the OEC version.

A different temporal indexing is applied in (Shanahan, 1992) where the speed of the computation is improved by using techniques from constraint logic programming. The event calculus clauses are transformed so that constraints on times are passed as parameters and then checked immediately after the goals which are known to bind them sufficiently. This kind of indexing would also apply to the object-based version.

The other kind of indexing is structural indexing, which is devising a method of storing the objects and events so that search is restricted to the potentially relevant candidates only. Consider the following reformulation of holds_at:

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

Supposing that the number of events in the database is much larger than the number of application specific rules used to describe the effects of these events, this formulation will do less search[1] . First, domain rules which are now specified by the predicates initiated_by and terminated_by, will be searched to determine which events potentially initiate the value of the specified attribute for the object. Then we just need to find an event which happened before time T among the set of possible events. Compared with the number of happens records, the number of domain specific rules to be searched will be trivially small.

Assuming that the number of events and objects is small, and there are not many application specific rules, a simple database system can be implemented efficiently by using one of the indexing schemes. It will be a memory-based system where every time a query is posed, the state of objects is derived using the event descriptions. However as we mentioned before, there is another efficiency consideration in executing the object-based event calculus axioms. We do not want to derive the same facts again and again every time a query is posed. In the case of having a large database where the number of events and rules is quite large and to which many queries are posed many times, it would be very impractical to search the database to derive the required facts about objects. In order to avoid this problem, a persistent storage can be used to store all the derived objects, their states and classes. Query evaluation can then be performed using this object store only. This approach is based on lemma generation.

Lemma Generation

Lemma generation has been used in the implementations of the relational event calculus and it has been shown that it can improve the execution of the event calculus dramatically (Shanahan, 1992; Sripada, 1991). The idea is to use the clauses of the event calculus once to deduce all possible facts regarding the relationships and time periods that are initiated and terminated by the recorded events. The information regarding each conclusion is then stored explicitly in a separate database and queries are solved using the information in this database only. However given such a separate database, an-

other possible source of computational redundancy occurs when the database is updated by a new event. When a new event is input, new conclusions may need to be added to the database and some old conclusions may become invalid. An important implementation issue is the maintenance of this database. A sophisticated storage manager should be incremental where only those computations which are affected by the update are redone.

We apply the lemma generation approach in the design of a practical database system on top of the object-based event calculus. We store all derivable facts about objects in a separate database which we call Object DataBase (ODB). The ODB contains all the derived information about objects: their states, their classes and the necessary information to derive the time periods in which these are valid. Given such an external database query evaluation can be considerably more efficient. In the following sections we present the details of applying this approach to the implementation of the OEC.

The Design of a Practical System

The overall conceptual architecture of the system is given in Figure 1. There is an interface which is used to communicate with the user. Through this interface the user can define the schema and events with their effects. The user can also interact with the database in an ad hoc fashion to query the database states at different times. The interface receives and processes user commands and invokes various procedures in the appropriate manager modules. The schema manager receives the schema definition from the user interface and records the information in an internal form using is_a and attribute relations. The schema also contains the domain specific rules which are organized according to classes. The query manager receives queries from the user interface,

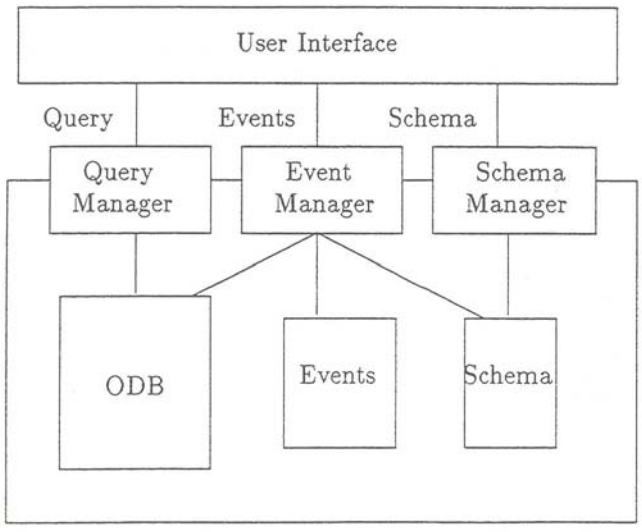

Figure 1 : Conceptual Architecture

translates them into an internal form and evaluates them using only the information available in the object database (ODB). The event manager accepts input of new events and stores them in the binary predicate format in the events database. The event manager is also responsible for maintaining the ODB. Every time a new event is input the ODB is updated incrementally by considering only the facts that are going to be affected by the new event. When a new event is recorded in the database, only the potentially affected objects are checked to see if any of them is changed. If so, corresponding facts are added to the object database.

A typical database application using this system will start with a schema definition. The next step is to record the event occurrences. After that, the database can be queried or updated with the addition of more events. The user generates the schema using a data definition language. The data definition language is a part of the user interface which can be designed as an interactive graphical tool or a classical alphanumeric interface. The schema declarations can be translated by the schema manager into the internal form which we use to describe the schema.

Schema definitions cannot be complete without defining the relevant events for each class. As we discussed before, events correspond to methods which can change the states of objects. The implementation of methods (or specification of events) is coded using the predicates initiates, terminates, assigns} and destroys. In order to implement this we can either use Prolog as the programming language of the system or invent a (perhaps graphical) language to specify the effects of the events.

The event manager is responsible for data manipulation. The only means to update the database is the addition of new event occurrences. The objects, their states and classes can only be derived by event descriptions. A proper interface should be provided for inputting event occurrences. The possible events and their effects are defined during the schema definition. The user can be provided with a graphical interface to browse and select the right event from the schema and record its occurrence by providing the relevant values for the place holders of the event description.

The user views the system as a deductive and object-oriented database in which all past states of objects are stored. In a typical application environment, the user will be interested in querying the states of objects and relationships between objects at certain times. The query manager provides a query language as a simple means of ad hoc interaction with the system and answers queries which are posed by the user.

Implementation

For an efficient query evaluation it is important to decide what to store in the ODB and how to store it. The other implementation issue is the maintenance of the object store. It must be updated incrementally every time a new event is input to the

system.

The Object Database (ODB)

The obvious set of lemmas to store in the ODB are all the time periods, analogously to the scheme proposed in (sury91 Sripada, 1991) for temporal relational databases based on the event calculus. Query evaluation would then be quite easy. In order to find a period of time for which an attribute holds a value, we need to search the database to find the corresponding fact. Once the time period is found it is easy to determine whether the value is valid at a specific time by just checking if it is within the time period. However storing the time periods explicitly complicates the maintenance of the ODB. Because the tuples with open time intervals need to be updated when an event terminates those time periods; and this update requires the deletion of tuples and addition of them again with the new time periods.

We prefer an alternative which is more flexible and easier to maintain. What holds_at derives is a tuple of the form (Obj, Attr, Val) at specific points in time. For each such tuple we record the starting time(s) at which that tuple is initiated, and, separately, a record of the time(s) at which the tuple is terminated by another event. The time periods for which the tuple holds can then easily be derived from these start and end points as required. Similarly the class(es) to which an object belongs in time are stored as tuples of the form (Obj, Class) together with the start or end times for each. When we keep the start and end points separately, the updates to the database will be additive and query evaluation will be fairly efficient. Therefore, we have chosen this second approach.

Since we assume that event occurrences are in order with the real world events, there can be two kinds of time periods that we can have in the ODB:

1. A time period which persists into the future. That is, the event initiating the value of an attribute is known, but there are no other events terminating the value of that attribute. In the ODB, we use the term start(e1) to denote the start of the time period which is initiated by an event e1.
2. A time period for which both the initiating event e1 and terminating event e2 are known. In this case we are dealing with time periods which have both start and end points. The start of the period is denoted by the term start(e1) and the end is denoted by end(e2).

An object's state can be stored in different ways. We have used Prolog in the prototype of the system and we used the external predicate object to store the objects. In this representation, the ODB contains a set of predicates of the form object (Obj, Attr, Val, Time) which are indexed by the object identities. Here Time can be in the form start(Ev) or end(Ev). The exact time of the event Ev is found out from its description. Similarly the tuples which assert the class(es) to which an object belongs at different times are coded using another external predicate, instance.

Example : Suppose the following events are recorded related to the state of an employee instance Ali:

e1 : Ali is hired as an assistant in CS department.
e2 : Ali is promoted to lecturer.
e3 : Ali moves to EE department.

Given these events the object-based event calculus will derive values and time periods for the attributes dept and rank for Ali. The derived information is stored in the ODB in the following form:

object(ali, dept, cs, start(e1)).
object(ali, rank, assistant, start(e1)).
object(ali, rank, assistant, end(e2)).
object(ali, rank, lecturer, start(e2)).
object(ali, dept, cs, end(e3))
object(ali, dept, ee, start(e3))
instance(ali, employee, start(e1)).
instance(ali, person, start(e1)).

Whenever a new event description is added to the database, the object database is updated incrementally by adding new conclusions that follow from the new event. The old conclusions that are terminated by a new event e} are stored by using the term end(e). Hence no deletions are made from the database, all changes to the ODB are additions. For instance if later a new event, say e4, occurs to promote the employee Ali to professor the object database will be updated in the following way:

Add : object(ali, rank, lecturer, end(e3)).
Add : object(ali, rank, prof, start(e3)).

Storing the derived facts in the ODB in this form provides us with all the necessary information about the time periods for which an object holds a particular state. From this we can easily derive the state of objects at specific time points.

Query Evaluation

Query evaluation is performed by the query manager. Finding the period of time for which an attribute holds a value is now a process of finding the relevant facts in the object database. Thus we define holds_for for the query evaluation as follows[2]:

holds_for*(Obj, Attr, Val, since(T)) ←
object(Obj, Attr, Val, start(Ev)), time (Ev, T),
not∃ Ev*[object(Obj, Attr, Val, end(Ev*)), Ev* > Ev].

holds_for*(Obj, Attr, Val, during(T1, T2))←
object(Obj, Attr, Val, start(Ev1)), time(Ev1, T1),
object(Obj, Attr, Val, end(Ev2)), time(Ev2, T2), T1 < T2
not∃ Ev*[object(Obj, Attr, Val, end(Ev*)), Ev1 < Ev* < Ev2].

This formulation deals with the two kinds of time periods. The predicate time is used to determine the time at which the events happen.

We can also find the value of an attribute at a specific point in time. In querying the ODB we use the following definition for holds_at :

holds_at*(Obj, Attr, Val, T) ←
object(Obj, Attr, Val, start(Ev)), time(Ev, T1), T1 - T,
not ∃ Ev*[object(Obj, Attr, Val, end(Ev*)),
time(Ev*, T2), T2 < T].

We can derive time periods for the class membership similarly. Thus we define instance_for for the query evaluation as follows:

instance_for*(Obj, Class, since(T))←
instance(Obj, Class, start(Ev)), time(Ev, T),
not ∃ Ev*[instance(Obj, Class, end(Ev*)), Ev* > Ev].

instance_for*(Obj, Class, during(T1, T2))←
instance(Obj, Class, start(Ev1)), time(Ev1, T1),
instance(Obj, Class, end(Ev2)), time(Ev2, T2), T1 < T2
not ∃ Ev*[instance(Obj, Class, end(Ev*)), Ev1 < Ev* < Ev2].

Finally we define instance_of in a similar fashion:

instance_of*(Obj, Class, T) ←
instance(Obj, Class, start(Ev)), time(Ev, T1), T1≤ T,
not ∃ Ev*[instance(Obj, Class, end(Ev*)),
time(Ev*, T2), T2 < T].

Maintaining the ODB

The contents of the ODB are maintained by the event manager. The main idea behind maintaining the ODB is to deduce all new conclusions whenever a new event description is added to the database and to store these new conclusions in the ODB indicating time periods by the terms start(e1) and end(e2).

The maintenance procedure is based on generating a list of actions to be performed upon the addition of a new event description to the database. The actions are only additions of new facts to the ODB. We use a meta-predicate adds to determine which facts will be added to the database. The binary predicate adds(Ev, Conclusion) means that Conclusion is a new fact to be added to the ODB as a result of adding the new event Ev to the database. In the following we present the clauses for the predicate adds that are used to generate the new facts depending on the effects of events.

We know that the periods in the ODB are either closed intervals or open intervals which persist into the future. A new event e either starts a new period of time (i.e., start(e)) for a fact or ends a period of time which was started by another event (i.e. end(e)). And the facts added as a result of the new event are either related to the attributes of objects or the class membership.

Let us start with an event type which initiates a value for an attribute of an object and thus starts a new period of time. The effect of such an event is to add a new object predicate to the ODB storing the new value for the attribute and the start of the time period. Our first rule to add a new fact to the ODB as a result of such an event is as follows:

adds(Ev, object(Obj, Attr, Val, start(Ev)))←
initiates(Ev, Obj, Attr, Val).

If an event initiates the value for an attribute of an object then by this rule a new tuple is stored for that attribute in the ODB with a time period persisting into the future.

Our second rule concerns the events which terminate a period of time for the values of attributes. When an event terminates the value of an attribute, again a new fact is added to the ODB to denote the end of the time period for which the value held:

adds(Ev, object(Obj, Attr, Val, end(Ev)))←
terminates(Ev, Obj, Attr, Val),
already_in(ODB, object(Obj, Attr, Val, start(Ev1))), Ev1 < Ev,
not∃ Ev*[object(Obj, Attr, Val, end(Ev*)), Ev1 < Ev* < Ev].

Here the predicate already_in is used to ensure that the attribute was initiated at an earlier time than the time of the new event. Its definition requires searching all the ODB, but since the arguments Obj, Attr and Val will be bound, the search space is reduced considerably. Without this check we could end up terminating the value of a single-valued attribute by the new event which is in fact initiating the value. The third condition in the clause is used to check that there is no other fact in the ODB asserting that the attribute has been terminated by an earlier event.

The ODB also contains information about classes of objects. Every time an event affecting the class membership of an object is recorded, the ODB needs to be updated accordingly. As discussed earlier, the class membership is described by the predicates assigns and destroys and it is affected by the events which describes the creation or deletion of objects. When a new object o is created as an instance of class c by an event e1, a fact of the form instance(o,c, start(e1)) is added to the object database. This is achieved by the rule:

adds(Ev, instance(Obj, Class, start(Ev))) ♦ assigns(Ev, Obj, Class).

If the object o is deleted or removed from a class later by another event, say e2, then the ODB is updated by adding a fact of the form instance(o, c, end(e2)). In order to add this fact we have the following rule:

adds(Ev, instance(Obj, Class, end(Ev))) ←
removes(Ev, Obj, Class),
already_in(ODB, instance(Obj, Class, start(Ev1))), Ev1 < Ev.

Implementing Versions

Versions are also objects whose state is represented as a set of identity-attribute-value tuples. These tuples can be stored in the ODB as they are generated by new events. Their identity provides the identity of the object from which they are derived. We describe the implementation by an example. In designing a document object doc, suppose that a version of the document v(doc,1) is created by an event. Assume that event e1 creates the document doc, e2 is an event to change the contents of the document and event e3 creates the version of the document by changing its title as well as the contents of the document. Let us consider the facts that are stored in the ODB related to the states of the object v(doc,1) and its parent object doc. After recording the events e1, e2 and e3, the ODB will contain the following facts:

object(doc, author, mjs-fnk, start(e1)).
object(doc, title, odaoo, start(e1)).
object(doc, text, 4sections, start(e2)).
object(v(doc,1), author, fnk-mjs, start(e3)).

instance(doc, document, start(e1)).
instance(v(doc,1), document, start(e3)).

The version creation and effects of events on version objects are described by initiates, terminates and assigns, removes statements. Hence the ODB maintenance rules as presented in this chapter are sufficient to store the derived facts about version objects. From this stored information it is possible to derive states of the two document objects at various times. However the query evaluation rules need to be modified to take versions into account. The modifications are related to considering the state of the parent object in deriving the state of a version object.

Performance Evaluation

We have used Prolog to illustrate the implementation issues of the proposed database system. As we have already presented the formulations in a form very similar to the Prolog syntax, it is quite straightforward to code them in Prolog. We have implemented all formulations presented in this paper in Quintus Prolog, however the implementation of the practical database management system as described in Section 5 is still under development.

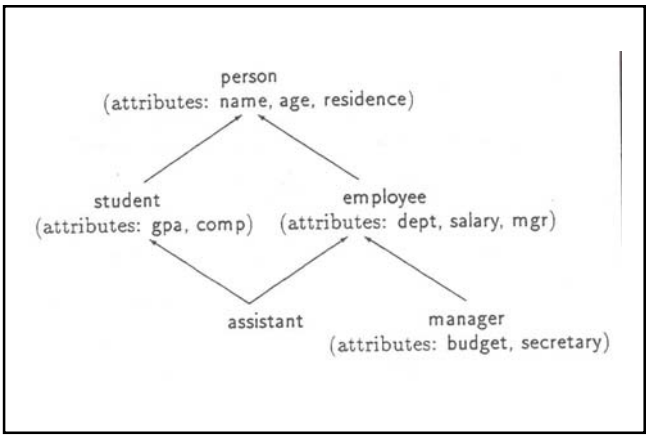

Figure 2: Sample class hierarchy

We have tested our prototype implementations for varying size databases and evaluated the performance of all formulations. We examined the performance of the system for different partially instantiated queries by measuring the response times. The schema of our experimental databases is shown in Figure 2. There are five classes each having several attributes. We have considered different types of events to model changes in a database supporting this schema. For each class there are events to create and delete an object. For instance for the class employee these events are named hire_emp and sack. In addition, there are class changing events. The event hire which is defined for the class student changes the class of a student object to class assistant. There are also events to change the values of attributes. For instance the event move changes the attribute residence of person objects and the event promote changes the attribute salary of employee objects. We did not include versioning of objects in these experimental databases.

We have generated different size databases based on this schema. Figure 3 and Figure 4 show the results of two implementations, one is based on plain event calculus, the other one is based on the ODB. Both implementations were executed in Quintus Prolog, Release 3.1.1 on a SUN SPARC workstation running under UNIX operating system. We have selected a sample set of partially instantiated queries which we think are typical in an object-oriented environment. We tested the performance of the system to find out response times for queries such as finding:

- one attribute of a specified object,
- all attributes of a specified object,
- the class of a specified object,

QUERY	Net time (milisec.) 200 events (~100 objs.)	500 events) (~250 objs.	1000 events (~500 objs.)	10000 events (~1000 objs.)
holds_at(manager1,name,N,100)	15.66	21.34	31.66	219.66
holds_at(manager1,A,V,100)	129.15	179.15	259.15	1679.20
instance_of(manager1,C,100)	6.5	13.33	21.67	207.33
instance_of(O,person,100)	175	347.5	646.65	5755.85
holds_for(assistant5,dept,D,P)	132.76	426.66	1247.23	26415
holds_for(assistant5,A,V,P)	1035	3285	8675	1.797e+05
holds_at(assistant15,A,V,100)	28.34	30.34	37.66	152.34

Figure 3: Sample timings for plain OEC implementation

QUERY	Net time (milisec.) 200 events (~100 objs.)	500 events) (~250 objs.	1000 events (~500 objs.)	10000 events (~1000 objs.)
holds_at(manager1,name,N,100)	0.64	2.2	2.13	3.4
holds_at(manager1,A,V,100)	3.84	8.60	8.25	13.2
instance_of(manager1,C,100)	0.6	0.65	0.58	0.78
instance_of(O,person,100)	21.9	28.2	36.58	46.1
holds_for(assistant5,dept,D,P)	5.11	5.33	7.55	8.3
¨holds_for(assistant5,A,V,P)	40	43	58	64.5
holds_at(assistant15,A,V,100)	0.06	3.76	3.8	13.43

Figure 4: Sample timings for ODB implementation

• instances of a specified class

at a certain point of time. The figures also show the results obtained by executing holds_for queries. And finally the last query in the tables is an example of a failing query. More complex queries can be built as conjunctions of these basic queries.

The same queries were executed on four databases containing 200, 500, 1000 and 10000 events respectively. Depending on the effects of the events, the objects in these databases were randomly created, updated and deleted. As we expected, the implementation based on the ODB performed much better than the plain implementation of the OEC. When the size of the database increases the performance of the system based on the plain OEC becomes unacceptably slow. However when we use the ODB in querying the database we obtain great improvements in response times. Especially when we consider queries executed against larger databases the response time can be 3000 times faster.

These results show us that the ODB implementation is quite promising for realizing larger scale applications. Further improvements on the implementation of the ODB using techniques from temporal relational databases is left as a future work.

Conclusion

In this paper we have presented an object-based variant of the event calculus to describe changes to the states of objects in a historical database. The OEC is a general approach for modelling various dynamic aspects of objects in databases. We have shown how state changes and versions of objects, and the creation, deletion and mutation of objects can be described in this framework. We then discussed the imple-

mentation of a database system based on the theory we developed. We keep a separate database, called ODB, to store all derivable facts about objects and this database contains all derived information about objects: their states, classes and the time periods in which these are valid. We described the design and implementation of a practical database system which uses the ODB in query evaluation.

The construction and maintenance of a separate database of lemmas trades a reduction in costs of query evaluation for increased complexity in event assimilation. Input of a new event will generally require new conclusions to be added to the ODB and some old conclusions may become invalid. As we presented, in the implementation and maintenance of the ODB the existing techniques developed for temporal relational databases can be used and a practical database system can be built based on the object-based event calculus. However, the complete realization of such a system is still a future work.

In conclusion, we believe that the problems in integrating deductive and object-oriented approaches cannot be realized and solved if the dynamic aspects of objects are ignored. We also believe that objects and events combined under a single framework provide a higher-level modelling methodology for structural semantics and dynamic aspects of objects and it can provide a foundation for a temporal and deductive object-oriented database system.

Endnotes

[1] We use Prolog-style computation only to illustrate the points. It could be possible that switching order of subgoals in this way, could make no difference in another language (e.g., Datalog (Ceri, 1989) or Megalog (Freeston, 1988)).

[2] We use an asterisk (*) to distinguish the predicates which are used in the query evaluation.

References

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

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

Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence*, 23:123-154.

Beech, D.& Mahbod, B. (1988). Generalized version control in an object-oriented database. In the *Proceedings of the 4th International Conference on Data Engineering*, Los Angeles, CA, 14-22.

Bjornerstedt, A., & Hulten, C. (1989). Version control in an object-oriented architecture. In W. Kim & F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, ACM Press, 451- 485.

Ceri, S., Gottlob, G. , & Tanca, L. (1990). *Logic Programming and Databases.* Springer-Verlag.

Ceri, S. et al. (1989). What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering* ,1(1), 146-166.

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

Chimenti, D., et al. (1990). *The LDL system prototype. IEEE Transactions on Knowledge and Data Engineering*, 2(1), 78-90.

Chou, H.T., & Kim, W. (1986). A unifying framework for version control in a CAD environment. In the *Proceedings of the 12th International Conference on VLDB*, Kyoto, Japan, 336-344.

Dalal, M., & Gangopadhyay, D. (1989). OOLP: A translation approach to object-oriented logic programming. In the *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, 555-568.

Fernandes, A., et al. (1993). *Approaches to deductive object-oriented databases.* Technical report, Dept. of Computer Science, Heriot-Watt University, Edinburgh.

Freeston, M. (1988). Grid files for efficient Prolog clause access. In Gray, P.M.D., & Lucas, R.J., editors, *Prolog and Databases : implementations and new directions*, Ellis Horwood, 181-211.

Fukunaga, K., & Hirose, S. (1986). An experience with a Prolog-based object-oriented language. In *OOPSLA'86 Proceedings*, 224-231.

Gallaire, H., & Minker, J. (1978). *Logic and Databases*, Plenum.

Indiketiya, R.V. (1992). Event calculus based temporal database management system. Master's thesis, Department of Computing, Imperial College.

Jensen, C.S., Mark, L., & Roussopoulos, N. (1991). Incremental implementation model for relational databases with transaction time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4).

Kesim, F. N. (1993). Temporal Objects in Deductive Databases. PhD thesis, Department of Computing, Imperial College.

Kesim, F. N., & Sergot, M. (1992). On the evolution of objects in a logic programming framework. In the *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2.

Kesim, F. N., & Sergot, M. (1993). Versioning of objects in deductive databases. In the *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, Phoenix, Arizona.

Kesim, F. N., & Sergot, M. (1994). A Logic Programming Framework for Modelling Temporal Objects. Technical Report. To appear in the *IEEE Transactions on Knowledge and Data Engineering.*

Kifer, M., & Lausen, G. (1989). F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In the *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 134-146.

Kifer, M., & Wu, J. (1989). A logic for object-oriented logic programming (Maier's O-logic revisited). In the *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.*

Klahold, P., Schlageter, G., & Wilken, W. (1986). A general model for version management in databases. In the *Proceedings of the 12th International Conference on VLDB*, Kyoto, Japan, 319-327.

Kowalski, R.A. (1979). *Logic for Problem Solving.* North Holland, New York.

Kowalski, R.A., & Sergot, M. (1986). *A logic-based calculus*

of events. New Generation Computing, 4:67-95.

Kuper, G.M. (1987). Logic programming with sets. In the *Proceedings of the 6th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,* San Diego, CA.

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

McCarthy, J., & Hayes, P. J. (1969). Some philosophical problems from standpoint of artificial intelligence. In B. Meltzer & D. Michie, editors, *Machine Intelligence*, Edinburgh University Press, 4: 463-502,.

Ramamohanarao, K., et al. (1988). The NU-Prolog deductive database systems. In *Prolog and Databases: Implementations and New Directions*, Ellis Horwood, 212-250.

Sadri, F. (1987). Three recent approaches to temporal reasoning. In Temporal Logics and Their Applications.

Shanahan, M. (1992). Computational aspects of the event calculus. Technical report, Imperial College, Department of Computing.

Snodgrass, R., & Ahn, I. (1986). Temporal databases. *IEEE Computer*, 19(9), 35-42.

Sripada, S.M. (1991). Temporal Reasoning in Deductive Databases. PhD thesis, Department of Computing, Imperial College.

Tansel, A., Clifford, J., et al. (1993). *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings.

Vieille, L., et al. (1991). *An overview of the EKS-V1 system. Technical Report* TR-KB-38, ECRC.

Zaniolo, C. (1985). The representation and deductive retrieval of complex objects. In the *Proceedings of Very Large Databases,* Stockholm, 458-465.

Nihan Kesim received the B.S. degree in Computer Engineering from the Middle East Technical University, Ankara, Turkey in 1986, the M.S. degree in Computer Science from Bilkent University, Ankara in 1988 and the Ph.D. degree in Computer Science from Imperial College, University of London in 1993. She is currently an Assistant Professor at Bilkent University. Her research interests include deductive databases, object-oriented databases, logic programming, data modeling and knowledge representation. Dr. Kesim is a member of the IEEE Computer Society.

Marek Sergot is Reader in Computational Logic at the Department of Computing, Imperial College, London. He graduated in Mathematics from Trinity College, Cambridge in 1973, completed a postgraduate course in Applied Mathematics at Cambridge in 1974, and then worked in mathematical modelling before joining the Logic Programming Section in the Department of Computing at Imperial College in 1979. His research is in the use of logic and logic programming techniques in Artificial Intelligence and databases, and in the formal specification of computer systems. He has particular interests in temporal reasoning, legal reasoning, and the formal theory of organisations.