A Logic Programming Framework for Modeling Temporal Objects

F. Nihan Kesim, *Member, IEEE Computer Society*, and Marek Sergot

Abstract—We present a general approach for modeling temporal aspects of objects in a logic programming framework. Change is formulated in the context of a database which stores explicitly a record of all changes that have occurred to objects and thus (implicitly) all states of objects in the database. A snapshot of the database at any given time is an object-oriented database, in the sense that it supports an object-based data model. An object is viewed as a collection of simple atomic formulas, with support for an explicit notion of object identity, classes and inheritance. The event calculus is a treatment of time and change in first-order classical logic augmented with negation as failure. The paper develops a variant of the event calculus for representing changes to objects, including change in internal state of objects, creation and deletion of objects, and mutation of objects over time. The concluding sections present two natural and straightforward extensions, to deal with versioning of objects and schema evolution, and a sketch of implementation strategies for practical application to temporal object-oriented databases.

Index Terms—Object-oriented databases, object versioning, deductive databases, temporal databases, temporal reasoning, event calculus, logic programming.

1 Introduction

The object-oriented and deductive approaches have generated considerable interest in the database and programming language fields. In databases, one of the main driving forces behind the recent interest in object-oriented languages is their support of a rich collection of data modeling and manipulation concepts. Another feature of the approach is the promise it shows in overcoming the so called impedance mismatch between programming languages used to code applications and database languages used to retrieve data. In parallel, the deductive approach has gained popularity as a candidate to solve this mismatch problem, since logic can be used as a computational formalism as well as a database specification and query language. A substantial amount of recent research has aimed at integrating these two paradigms to provide a single powerful framework for future database systems. Although there is still no general agreement on how this integration should be carried out—some authors even argue that one cannot have a system that is both truly deductive and truly object-oriented because of the conceptual mismatch between *value-oriented* logic programming and the notion of *object* as imported from object-oriented programming—there have been some promising developments, especially in the emergence of logics for objects with identity and complex internal structure. Existing proposals are summarized in Section 2.

Most of this work, however, has ignored dynamic aspects, that is to say, the complications that arise when objects evolve over time or mutate into objects with different internal structure. Work on the representation of temporal phenomena, on the other hand, has tended not to involve any explicit notion of 'object.' In temporal databases, research is dominated by approaches based on the relational model, although there are some exceptions. Some references are provided at the end of this section. Outside the database field, in AI in particular, there is extensive work on temporal reasoning, but here again 'fluents'—the propositions whose truth value varies over time—are typically represented as ground terms of some first-order language.

In this paper, we address the representation of temporal information in object-oriented databases. We do this by developing a variant of the event calculus [28], which we call *Object-based Event Calculus* (or OEC in short), for describing and reasoning about changes to objects in a logic programming framework.

The event calculus was introduced in [28] as a general logic programming treatment of time and change. Events, which are taken as the primitive temporal notion, mark the occurrence of change, and initiate and terminate periods of time for which facts hold. Given a record of events that have happened or that may happen, the event calculus can be used to determine what facts hold at any given time, or to compute the periods—the maximal intervals of time—for which a fact holds continuously. In the standard versions these time-varying facts are represented as (ground) first-order terms. From the database perspective, they can be seen as tuples of relations: the event calculus is then a method of deriving, for every such tuple, the periods of time for which it holds (the 'lifespan' of the tuple in the terminology of [15]). A snapshot of what holds at any time has the form of a relational database.

The timestamping of relational tuples with intervals is a

• *F.N. Kesim is with the Department of International Relations, Bilkent University, Bilkent, Ankara 06533, Turkey. E-mail: nihan@bilkent.edu.tr.*
• *M. Sergot is with the Department of Computing, Imperial College of Science, Technology, and Medicine, University of London, 180 Queen's Gate, London SW7 2BZ, United Kingdom. E-mail: mjs@doc.ic.ac.uk.*

Manuscript received Feb. 22, 1995.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K96060.

common technique in many temporal database systems. The main difference in the event calculus is that these intervals are not inserted and modified explicitly but are derived from the record of event occurrences as required: the events effectively give some semantic structure to the endpoints of intervals. A similar idea, expressed in terms of an extended relational algebra, has recently been proposed in [43]. The record of event occurrences is there called a 'journal.' Operators are proposed to derive what holds at intermediate times: A 'history' operator converts event data to intervals, and a 'snapshot' operator determines what holds at a given point in time.

The event calculus is also intended to contribute to the treatment of database updates (see in particular [26]). This is not an aspect that we shall discuss in this paper, however. Similarly, the event calculus has been extended and applied to the construction of temporal databases that support both 'valid time' and 'transaction time' [39]; again this is a further development we do not undertake.

In this paper we construct a version of the event calculus—the OEC—for dealing with changes to objects. As in the original (relational) event calculus, the changing world is described in terms of a record of events (a 'journal'), from which the OEC can reconstruct and access the states of objects at any time. We get a database in which all states of objects are stored (implicitly). A snapshot of this database at any given time is an object-oriented database—'object-oriented' in the sense that it supports an object-based data model. For reasons explained later, we shall adopt the view of an object as a collection of simple atomic formulas with a standard first-order semantics.

This paper is an expansion and further development of our previous presentation [19] where we discussed the evolution of objects using the event calculus. We now develop the basic idea and explore other temporal aspects of objects as well.

The paper has three main components:

1) Since an object, however understood, is a more complicated structure than a collection of relational tuples, several different kinds of change can be identified, each requiring its own treatment. We examine the main kinds of change in detail—in sufficient detail that the resulting formulation can be executed directly, as a Prolog program, say. The same problems arise whatever representational formalism is employed. The formulations proposed could be reconstructed, if preferred, in some other representational formalism, such as the situation calculus.
2) In common with much current usage, the term 'database' is used in this paper to refer to a wide, loosely defined class of applications, not just to large-scale database systems, narrowly understood. In the first instance, the OEC is intended to be used in the construction of 'database' or 'knowledge-base' applications where the problems of scale and performance associated with large-scale database systems are not a major factor. Some examples are mentioned in the text. However, it is also our aim to develop the OEC as a basis for practical, large-scale temporal database systems. In this last respect we shall be concerned with explaining how previously proposed implementational strategies in the temporal database literature may be adapted for use with the OEC.
3) The OEC's mechanism for maintaining the state history of objects leads to the *versioning* of objects as a natural and straightforward development. Event descriptions can be used to keep parallel histories of objects, and these can be used to model multiple versions of the same object at a time.

The paper is organized as follows. Section 2 presents a brief survey of the existing work on reasoning with complex objects for the purpose of identifying, in Section 3, the basic data model that will be supported by our object-based variant of the event calculus. The OEC itself is presented in three separate sections. In Section 4, we present the basic formulation 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. And in Section 6, we extend the formulation to incorporate some other object-oriented features, specifically multivalued attributes and methods for derived attributes. We also show how the OEC can be adapted in a natural way to deal with versioning of objects and schema evolution. Section 7 discusses practical considerations and implementation strategies for temporal databases based on the OEC.

The literature on temporal reasoning and temporal databases is very extensive and we do not attempt a full survey here. Comparisons of the event calculus with situation calculus are provided in [27]. For temporal databases, [25] provides a recent bibliography of work in this area together with pointers to previous bibliographies. The collection [41] gives an excellent overview of the main approaches and discusses many of the issues that are studied in this field. As already mentioned, most work in temporal databases has been undertaken in the context of the relational model. Exceptions include [10], [18], [33], [40], [44]. Comparisons with other proposals and references to specific points are given as they arise in the text.

2 Complex Objects

The purpose of this section is to identify and motivate the choice of data model we have adopted for the OEC.

Although there has been much confusion and controversy about the meaning of object-orientation in general and object-oriented databases in particular, a number of concepts have emerged as characteristic of this approach. Several papers [4], [3] have now proposed a set of base features for object-oriented databases, that is, databases which support an object-oriented data model. There is no single standard model, but there is a set of basic concepts common to all object-oriented programming and knowledge representation languages.

A great number of attempts have been made to use logic in establishing a formal semantics for object-oriented concepts. Some of the existing works take deductive databases as the basis and extend the existing systems with the concept of a structured object. Most of the work in this approach follows the research on non-1NF relations, in order to extend the data structures of logic programming with sets and complex terms [1], [8], [29], [46]. Others attempt to formalize the basic object-oriented concepts by developing a new logic to support various features of complex objects [5], [7], [23], [30]. There is also another stream of work which approaches the problem from a programming language perspective. Here the aim is to extend the logic programming languages with some object-oriented features such as methods and message passing [9], [13], [31], [45]. These proposals are of less interest in the context of this paper since their primary concern is with programming constructs.

When we compare the existing work, we see that the semantics of a complex object differs widely. In the proposals which extend deductive databases with sets and complex terms a complex object is viewed as a tuple in a higher-order relation. In object logics a complex object is either an element in some partially ordered domain or a collection of simple atomic formulas. Below we summarize these different semantics of objects and assess them according to their ability to support the representation of changes to objects.

2.1 Higher-Order Relations

One way of incorporating complex objects is to extend predicate calculus to a higher-order logic so that the value of an argument of a predicate can also be a relation built by using tuple and set constructors. Several higher-order extensions of logic programming, such as LDL [8] and COL [1] have been proposed. They view a complex object as a tuple in a higher-order relation. For example the COL statement:

```
person(john, 38, "London",
        {chess, tennis}, {tom, sue}).
```

describes information about a person and his hobbies and children. Sets can be represented either explicitly as in this example or by data functions. In LDL, a *grouping* construct is used to construct a set by using a rule. For example, the rule

```
children(X, <Y>) ← parent_of(X, Y).
```

groups together all the children of each person. Here < > is the grouping operator.

These proposals can be characterized as attempts to incorporate some notions from the object-oriented paradigm, without compromising the goal of having the relational model as the basis of the extended model. Thus they are often said to be value based. Although higher-order logic provides a formal framework for nested relations and complex objects, it also has some disadvantages. The higher-order semantics of sets presents severe semantic problems for logic programs in these languages, and it is difficult to develop an efficient query evaluation in these approaches.

Another disadvantage is that representing a complex object by a nested tuple is practically not very convenient. Because of syntactical limitations (e.g., fixed arity) these languages do not provide access to substructures of complex objects in a homogeneous way. They are unsuitable for deductive retrieval at arbitrary levels. In the above example, to find the age of the person `john`'s child `tom`, one must start from the top predicate `person` and then continue down to the substructures where the required information can be found. In the case of updates, semantic problems also arise. For example, if `john` develops a new hobby, adding the new information will yield a completely different tuple which does not have any semantic relationship with the original one. Omitting the information about address will produce a tuple of a completely different type.

2.2 Object Logics

The other main stream of work aims at developing a new logic to support various features of complex objects. It is argued that just as for relational databases, a logical framework can be established for object-oriented databases also. The underlying logic must be different from first-order predicate logic because most features of object-orientation require higher orderness. On the other hand it is desirable to have a logic with first-order properties: following the terminology of [6] the language (syntax) of an object model must be higher order to be able to manipulate such concepts while the semantics must be restricted enough to satisfy first orderness. A number of such object logics have been proposed.

The first work, influenced by the ψ-terms of LOGIN [2] was Maier's O-logic [30] which was later extended by a number of proposals, namely C-logic [7] and F-logic [23]. From the object-oriented world these logics acquire the notion of object identities, complex objects, a mechanism for object classification and a structure for property inheritance. From the logic programming world they absorb the concepts of unification, answer substitution and a strategy for deductive query processing.

In these logics, an object is represented as a complex term in the language. For instance in F-logic [23], the `person` object illustrated in the previous section can be represented by a complex term as follows:

```
person:john [age⇒28,
             address⇒"London",
             hobbies⇒{chess,tennis},
             children⇒{person:tom,person:sue}]
```

Here `person` is the class name, `john` is the object identity and the labels denote attributes.

The syntax of these complex terms is influenced by the language LOGIN but their semantics is different. In LOGIN complex terms denote *types* and inheritance is incorporated into the unification algorithm. In the object logics, complex terms are formulas in their own right: written as a formula, a complex term asserts that an individual object with that structure exists. More complex formulas are built by combining object terms using the usual truth-functional connectives and quantifiers.

Although the syntax and the informal reading of complex terms are quite similar in the object logics, the precise semantics given to the complex terms varies. F-logic views an object term as an element in a partially ordered domain. Partial orderings on class names and object identities are defined and using these orderings a partial ordering over complex object terms is obtained and used to capture sub-object or sub-type relationships.

The major disadvantage of this approach is that the logic becomes more complicated as more features, such as methods and inheritance, are introduced. A natural concern is whether there might be an efficient evaluation procedure for queries. Another important concern, directly related to the main topic of this paper, is the ability of this approach to model the dynamic behavior of objects. This question is not addressed in the current literature, and many of the difficulties seem not to have been anticipated. For instance, some common types of change would seem to require changes to the partial ordering on class names and object identities, and hence effectively to the language itself. It is difficult to see how such changes could be accommodated smoothly, and no suggestions on these points have appeared, to our knowledge.

C-logic [7] takes a different approach. Here, complex object descriptions are considered as collections or conjunctions of atomic properties. Each attribute label is viewed as a binary predicate and each class symbol as a unary predicate. An object with several labels can then be described as a conjunction of several atomic formulas. For example the term

```
j3[name ⇒ "John Smith", age ⇒ 28]
```

or as

```
j3[name ⇒ "John Smith"] ∧ j3[age ⇒ 28]
```

or as

```
name(j3, "John Smith") ∧ age(j3, 28)
```

in first-order logic.

This formula approach makes it possible to understand the semantics of complex objects within the predicate logic. Chen and Warren give a semantics to C-logic directly, and also by transformation to an equivalent first-order formula which uses unary predicates for types and binary predicates for attributes. This makes proof procedures and associated computational developments in first-order logic readily available for complex objects.

One advantage of the formula approach taken by C-logic is that it allows information about an entity to be specified and accumulated piecewise, which facilitates the update of subparts of an object independently of others. The explicit notion of object identity also makes sharing and updates easier. Adding new information about an object is just a matter of adding one or more binary predicates. The subparts of an object can be retrieved by using the identity of the object and the attribute name describing the subpart.

3 The Data Model

In this section, we present the data model that the OEC will support.

The data model provides a basic set of features associated with structural object orientation: object identity, complex objects with both single-valued and multivalued attributes, methods for derived attributes, classes, class hierarchies, and inheritance. This is the set of features identified as the essential ingredients of object-oriented data models in [3], to which we have added derived attributes since they are useful in applications and can be supported straightforwardly. The treatment we adopt follows the formula approach exemplified by C-logic [7] as summarized in the previous section.

We view an object as a named collection of *object-attribute-value* triples. Every object is abstracted by a unique identity which distinguishes it from other objects. Following Kifer and Wu [23] 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. (We use the standard Prolog convention for constants and variables throughout the paper: strings beginning with an upper-case letter are variables.) For example `john, X, children(john, mary), path(X, Y)` can all be terms denoting identities. The set of all ground identity terms plays a role analogous to that of the Herbrand Universe in classical logic. Function symbols are used to construct new object identities out of existing ones. The objects have attributes whose values can be other objects (or more precisely their identities).

Objects are organized into class hierarchies, defined explicitly by asserting `is_a` relationships among classes. A class denotes a set of object identities. Each class has a unique name to distinguish it from other classes. The class-subclass relation (`is_a`) is to be read as the subset relation: the set of objects represented by a class includes all the objects belonging to the subclass(es) of that class.

The relation between a class and its instances is represented by the `instance_of` relation. The set of instances of a class changes as new objects are created and cease to exist. This time-dependent behavior of the `instance_of` relation will be discussed in Section 4.5.

A class describes the internal structure of its instances by attribute names. This structure is asserted by the predicate `attribute`. A subclass inherits the structure of its superclass(es).

As an example consider the class hierarchy shown in Fig. 1. Classes `student` and `employee` are subclasses of `person`. The attributes common to all persons (i.e., `name`, `address`) are defined in the class `person` and are inherited by the subclasses. The subclasses also define additional (more specific) attributes. The class hierarchy is described as follows:

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

attribute(person, name).
attribute(person, address).
attribute(student, section).
⋮
```

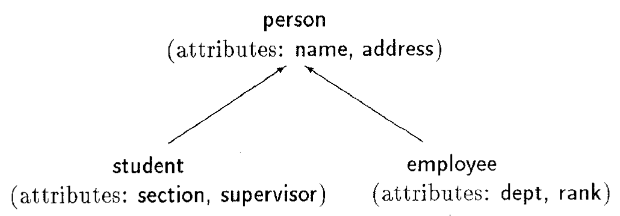

person
(attributes: name, address)
student
(attributes: section, supervisor)
employee
(attributes: dept, rank)

Fig. 1. A simple class hierarchy.

For the purposes of this example, we have assumed that all attributes are single-valued. As will be shown in the next section, the functionality constraint of such attributes is satisfied within the formulation of the OEC. The extension to allow multivalued attributes in addition is straightforward but for explanatory purposes we leave this aside, together with methods for derived attributes, until Section 6.

In order to formulate the inheritance of attribute names by the subclasses we define the predicate `attribute_of` in the following way:

```
attribute_of(Class, X) ←
     attribute(Class, X).

attribute_of(Sub, X) ←
     is_a(Sub, Class),
     attribute_of(Class, X).
```

This formulation for objects and classes allows a very simple form of inheritance. It is limited to the subset relation between classes and monotonic inheritance of attribute names. Multiple inheritance without overriding can also be expressed by the `is_a` predicate. This simple type of multiple inheritance causes no additional difficulty and is not mentioned again.

4 The Object-Based Event Calculus

In this and the following two sections we shall present the OEC, a version of the event calculus that supports the data model described in the previous section. Given a description of event occurrences (changes in the world) the OEC can reconstruct the state of any object in the database at any point in time. It can also be used to compute the period(s) of time for which an object 'exists' (its 'lifespan' [15]) and the periods of time for which given attributes of objects have given values. For simplicity we shall assume for the time being that all attributes are single-valued and we shall ignore methods for derived attributes. These features of the data model will be reintroduced in Section 6.

The OEC is based on a simplified, asymmetric case of the event calculus, where facts are assumed to persist forwards in time until they are terminated by some subsequent event. Correspondingly, the assimilation of events into the database is assumed to keep step with the occurrence of changes in the world. This is in contrast to the original formulation of the event calculus [28] which treats past and future symmetrically and can deal with the case where events are not necessarily recorded in the same order in which they actually occur.

This simplified version of the event calculus corresponds closely to updates in conventional databases [26]. It blurs the distinction between an event occurrence (a change in the world) and the recording of that event in the database. Accordingly, the database that is maintained by the OEC can be seen either as a historical or 'valid time' database recording the evolution of some set of objects in the world, or as a system in which all past states of an object-oriented database are accessible. (And if valid times and transaction times are distinguished but are exactly correlated, then it can be seen as a 'degenerate bitemporal' database [17].)

It would be possible to construct a version of the OEC without these assumptions following the symmetrical treatment of past and future of the original event calculus [28]. It would also be possible to extend the treatment to support both 'valid time' and 'transaction time' as done by Sripada [39] for the relational versions. We do not attempt these further developments in this paper. Similarly, although it is only the relative ordering of events that is significant in the event calculus we shall assume that the times of all event occurrences are given since this is often useful in practice.

We present the OEC in stages. We begin with the simplest kind of change, which is change to an existing object's internal state.

4.1 Change of Internal State

The state of an object is determined by the values assigned to its attributes. Change in internal state corresponds to changing the value of any of the attributes. The basic idea in dealing with the evolution of an object over time is to parametrize its attributes with times at which these attributes have various values. Formulation of this idea within the spirit of the event calculus is straightforward. Events initiate and terminate periods of time for which a given attribute of a given object takes a particular value. Fig. 2 shows the history of an employee object. Here `john` is the identity of the object and `rank, dept, age, address` are the attributes that are initiated to different values at different times. The object-based event calculus constructs such a state history of objects.

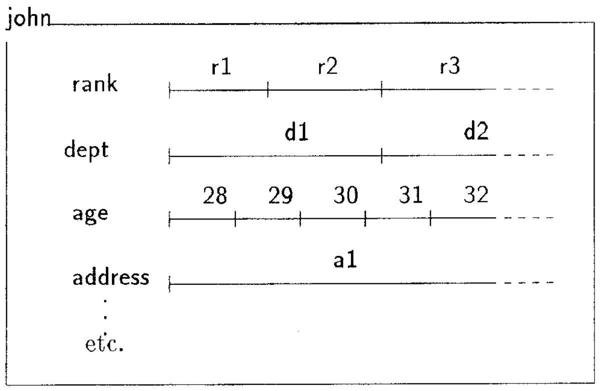


Fig. 2. State history.

The effects of events are described by the predicates **initiates** and **terminates** by means of assertions (or more generally rules) of the form:

```
initiates(EventType, Obj, Attrib, Value).
```

(and similary for `terminates`). For example an event of type 'promote employee `X` to new rank `R`' initiates a period of time for which employee `X` holds rank `R` and terminates whatever rank `X` held at the time of the promotion:

```
initiates(promote(X, R), X, rank, R).
```

Here `promote(X, R)` is a term representing the type of the event. Since we are dealing with single-valued attributes it is not necessary to specify explicitly that the old rank is terminated. The details are shown in a moment.
Given a fragment of data:

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

the OEC can be used to compute values of attributes of objects at given times, as in the following two queries:

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

The formulation of `holds_at` in terms of `initiates`, `terminates` and `happens` is shown presently.

The OEC can also compute the periods of time for which an object's attributes have particular values. In the example, for instance, the query

```
?- holds_for(jim, rank, R, Period).
```

would generate the answers

```
R = assistant, Period  =  1986-1988;
R = lecturer,  Period  =  1988-1991;
R = professor, Period  =  since(1991).
```

A term of the form `Ts-Te` denotes a time interval (closed on the left and open on the right) with start and end points `Ts` and `Te`, respectively; `since(Ts)` denotes an open-ended interval, the set of time points later than or equal to `Ts`. Time points need not be years, as in this example. Notice that we do *not* include in the time line any distinguished time point 'now' or 'unchanged' as seems to be common in many temporal database systems (see for example the collection of papers [41]).

The following is the basic formulation of the OEC to derive the value of an attribute of an object at a specific 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).
```

Informally, these clauses may be read procedurally as follows: in order to find the value `Val` of an attribute `Attr` of an object `Obj` at a time `T`, find an event `Ev` which happened before time `T` and initiated the value of that attribute; and then check that no other event which terminates that value has happened in the meantime. The interpretation of `not` as negation by failure in the last condition for `holds_at` gives a form of default persistence: the value of an attribute is assumed to hold at all times after its initiation by event `Ev` unless there is information to the contrary.

The constraint that attributes are single-valued implies that the value of an attribute is terminated if an event initiates it to another value. This is represented by adding the following general rule:

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

(The use of the anonymous Prolog variable '_' in this clause is just to cover the unlikely case that an event is specified to reinitialize an attribute to its existing value.)

The computation of periods of time is obtained by the following:

```
holds_for(Obj, Attr, Val, Ts-Te) ←
   happens(Ev, Ts),
   initiates(Ev, Obj, Attr, Val),
   terminated(Obj, Attr, Val, Ts, Te).

terminated(Obj, Attr, Val, Ts, Te) ←
   happens(Ev, Te), Ts < Te,
   terminates(Ev, Obj, Attr, Val),
   not broken(Obj, Attr, Val, Ts, Te).
```

We require another clause to deal with periods that have no end-point (i.e., for the case where the value of an attribute is initiated but there is no subsequent event which terminated the value):

```
holds_for(Obj, Attr, Val, since(Ts)) ←
   happens(Ev, Ts),
   initiates(Ev, Obj, Attr, Val),
   not terminated_later(Obj,Attr,Val,Ts).

terminated_later(Obj, Attr, Val, Ts) ←
   happens(Ev, Te), Ts < Te,
   terminates(Ev, Obj, Attr, Val).
```

Given a set of events, the object-based event calculus can be used to answer queries such as finding out the value of an attribute of an object at a specific time, or the period of time for which an attribute of an object has a given value. We can determine the state of an object at any time by finding out the values of all its attributes.

Of course execution of this event calculus, in Prolog say, does not yield an object-oriented style of computation. But conceptually, in object-oriented terminology, we could consider events as messages to modify object states. The specification of how events affect the state of objects would then correspond to methods, and the predicates `initiates` and `terminates` would be the system primitives by which the methods are implemented.

So far we have discussed how event calculus can be used to describe changes to the internal states of objects, i.e., to values of attributes of objects. Apart from the events that cause changes of state of existing objects, there are other kinds of events which cause the creation of new objects or deletion of objects. Before moving on to present other kinds of changes, we wish to make a remark about the representation of events.

4.2 Digression: Representation of Events

In the formulation of the OEC we have adopted C-logic's formula approach for the treatment of objects in the data model. In this paper we also use C-logic syntax as a convenient shorthand for describing events. The transformation of C-logic to Prolog (see Section 2) allows us to mix C-logic and standard Prolog syntax freely, and this is particularly convenient when describing events. For example, an event which is described in Prolog by the following assertions

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

can be written equivalently and more concisely using C-logic syntax as follows:

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

We could also write, for example,

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

Generally we prefer to separate the structure of the event from the record of its occurrence (**happens**), as in the first C-logic version above. Whichever formulation is chosen, the C-logic to Prolog transformation makes all of them equivalent.

It is important to note that the C-logic representation of events is not essential to the main theme of the paper. We are primarily concerned with the treatment of changes to objects, and for this we have followed C-logic's formula approach for the semantics of complex objects. It is of minor importance that we have also chosen to use C-logic syntax for the representation of events. To put it another way, the C-logic representation of events could just as well be employed to reformulate the original (relational) event calculus, but that would not alter the nature of the data model supported by that version of the event calculus.

4.3 Creation of Objects

Creation of a new object of a given class means adding new information about an entity to the database. We can think of describing object creation by events—birth of a person, manufacturing a vehicle, hiring a new employee—whose specifications provide the necessary information about the initial state of the object.

In object-oriented databases, classes provide an instantiation mechanism for creating their new instances. Instantiation is performed by calling on a class to create a new object based on the information given in the class. This object is then initialized by giving each of the attributes an appropriate initial value.

In the context of temporal databases, objects are not created and destroyed. What changes is whether an object with a given identity exists or not. But since every object in our framework belongs to a class, it is unnecessary to introduce a separate **exists** predicate: instead we make class membership, **instance_of**, a time-varying relationship. In other words, to determine whether an object x of class C 'exists' at time t, we determine whether x is an instance of class C at time t. The 'creation' of an object is then a matter of assigning it to a chosen class and specifying its initial state. We handle creation of objects by specifying which events assign objects to which classes, employing for this purpose a new predicate **assigns**. We use the same event description to initialize the state of the object. As an example consider registration of a student. The description of a specific registration event might be as follows:

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

The rules that specify the effects of such registration events are:

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

initiates(Ev, Obj, section, S) ←
     event:Ev[act ⇒ register,
     object ⇒ Obj, section ⇒ S].
initiates(Ev, Obj, supervisor, S) ←
     event:Ev[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. In initializing the state of the object, not all attributes need to be assigned to values. Some attributes may not have any values or they may have "undefined" as a value. The occurrence of the specific registration event described above is recorded by:

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

There is one further point of detail. Assimilation of new event descriptions into the database will generally require introducing one or more new object identities (e23 and presumably **ali** in the above example). In a practical implementation, generation of unique new identities can be left to the system. But notice that generation of object identities is not the same problem as 'creation' of new objects.

Recall from our presentation of the basic data model that instances of a class are also instances of all the superclasses. It is therefore necessary to arrange that any new instance of a class should automatically become a new instance of the superclasses. There are several ways of arranging for this, of which the simplest is to include the following rule:

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

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. That is, objects exists (i.e., they are in the database) or cease to exist at various times. Their existence is described by assigning their identities to their class. Once an object is assigned to a class it remains as an instance of that class during its lifetime and never changes class.

4.4 Deletion of Objects

Deletion of objects can also be described by events. 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: more precisely, since we are dealing with temporal databases, the object ceases to exist, or rather, ceases to be an instance of any class. The other form of deletion 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 over time, which will be discussed in Section 5.

For the purposes of this section, we assume that when an object is 'deleted' not only does it cease to belong to the set of instances of its class and superclasses, but also all of its attribute values are terminated. The reason is that attributes represent the internal structure of an object. If an object ceases to exist then it is no longer meaningful to speak of its internal structure.

We use another new predicate `destroys` to specify events that 'delete' objects. The rule:

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

has the effect that all attributes defined in the class of the object and also those inherited from superclasses are automatically terminated when the object ceases to exist.

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' [47]. We eliminate dangling references by adding another general rule for the `terminates` predicate:

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

The effect is that the value `ValObj` of the attribute `Attr` is terminated by any event which destroys the object `ValObj`.

4.5 Class Membership

As objects are created and deleted/destroyed, the instances of a class change in time. This temporal behavior of class membership can be handled by parametrizing the `instance_of` relation with times. 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 ≤ T,
      assigns(Ev, Obj, Class),
      not removed(Obj, Class, Ts, T).

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

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

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

We can also write the analogue of `holds_for` (i.e., `instance_of`) to compute the periods of time for which an object belongs to a class (or 'exists'). Note that an object can have several distinct periods of membership (or 'existence'). We omit the details of `instance_for` since they can be reconstructed straightforwardly by comparison with the earlier formulation of `holds_for`.

4.6 Discussion and Related Work

We have introduced two separate sets of predicates, one for dealing with change in internal state of objects and one for creation/deletion of objects. The internal states of objects are derived by use of the predicates `holds_at` and `holds_for`. These are defined in terms of predicates `initiates` and `terminates` which specify the effects of events on objects' internal states. The temporal class membership is derived by the predicates `instance_of` and `instance_for`; predicates `assigns` and `destroys` are used to specify how events affect class membership. The formulation of these two sets of predicates are direct analogues of one another. We could combine them into one set of predicates, with one general formulation, and thereby dispense with one set of predicate names altogether. We have not done so because we want to emphasize the conceptual difference between changes in an object's state on the one hand and changes to class membership on the other.

The treatment of change formulated in the OEC is appropriate under the assumption that facts and properties of objects persist over time—that, once initiated, each fact continues to hold without interruption until it is terminated by some subsequent event. Such facts have been termed 'stable' or, perhaps more perspicuously, 'stepwise constant' [34] in the literature on temporal databases. The OEC can be extended to accommodate other kinds of time-varying behavior by incorporating various extensions that have been developed for the original, relational event calculus. In particular continuous change can be treated using the 'trajectories' of [35]. We do not present the details here. The treatment can be imported from the relational versions without modification, and is actually slightly more convenient to formulate within the OEC, since it is continuous change of values of attributes that is of interest; it is difficult to imagine what continuous change of membership of a class would correspond to. Other extensions, such as allowing for different granularities of time within the same data model [12] could also be adapted straightforwardly.

In the database field, the modeling of temporal information has been dominated by approaches based on the relational model. There are exceptions (see [37] for a comparative survey). These proposals differ in the range of modeling features they provide. More significantly, they differ also in their general approach to the representation of temporal information, and to the notion of 'object' itself. We select here two examples, each of which is intended to be representative of a general class of approaches.

The first example is the extended entity-relationship model described in [10]. This has many features in common with the data model supported by the OEC. It has entities, time-invariant identities (called 'surrogates' for entities), attributes, and time-varying membership of classes and subclasses. The temporal extension records a 'lifespan' with each entity, with each attribute-value, and with each class membership instance. These lifespans correspond exactly to the time periods computed by the `instance-for` and `holds-for` predicates of the OEC. The model of [10] also supports relationships between entities, a feature not provided by the OEC. A corresponding extension of the OEC, discussed briefly in [19], could be obtained by combining the OEC with the relational version of the event calculus, but this is something we have not undertaken yet.

The crucial difference between the OEC and the temporal entity-relationship model of [10] lies in their respective treatments of lifespans/periods and their treatments of updates. In [10] updates are treated as *operations* on the database, and the system provides a number of basic operations for this purpose: to create new entities of a given type, to insert and delete entities from classes, and to modify attribute values. In each case, the corresponding lifespan must be manipulated explicitly. The main motivation for the OEC, in contrast, as in the original event calculus, is to provide some semantic structure for updates and for the computation of 'lifespans.' Updates, corresponding to assimilation of information about changes in the world, are treated by *adding* an appropriate *event description* to the 'journal' stored in the database. The effects on the objects and their attributes are not modeled as operations on the database but are *derived* from the event descriptions via the specifications given by the `initiates-terminates` and `assigns-destroys` predicates. We should note, however, that other kinds of change, such as corrections of mistakes, are supported directly by the OEC and must be treated as modifications of the 'journal'. A more powerful and general treatment would require the introduction of a separate transaction time dimension. As already mentioned, this has been done for versions of the relational event calculus [39] but not yet for the OEC.

The second representative is the temporal object-oriented system described in [44]. That system, however, places considerable emphasis on encapsulation and abstract data types. These are object-oriented programming features that are generally not mimicked directly in logic-based treatments of objects. (See the discussion in Section 2.) The proposal in [44] also treats time points and intervals as abstract data types, which we do not attempt.

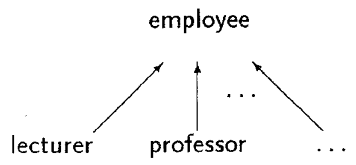
To avoid any misunderstanding, we should perhaps state explicitly that we are not claiming in this paper to have introduced or invented some new modeling concept—temporal or object-oriented—that is not already found in the literature. Conversely: our intention is to show how a base set of object-oriented features may be provided in a natural fashion in an integrated temporal/deductive framework; to demonstrate that this framework can be further extended to provide a wider range of features, some of which we present in detail; and to indicate that the resulting system can be used as it stands for the construction of practical small-medium applications and has the prospect of further development to large-scale database applications. We want to emphasize that it is the general framework that is the basis for meaningful comparisons with other work, and not the list of features currently supported by the OEC.

5 Mutation of Objects: Changing the Class

We have so far assumed that objects exist, cease to exist in the database, but never change class. However in the real world it is common that objects evolve over time. Consider the representation of an employee instance again. We have represented the rank of an `employee` object by including an attribute `rank` whose value can change over time:

```
employee:jim[... rank ⇒ lecturer, ...]
employee:mary[... rank ⇒ professor, ...]
            ⋮
```

But suppose that instead of using the attribute rank, we had chosen to divide the class of employees into various distinct subclasses:



Then employees of different ranks would be considered as different clauses, represented using `is_a`:

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

The choice between the two representations is a data modeling issue. If employees have different additional attributes according to their ranks then it is appropriate to represent different ranks as subclasses. However, even if the structure of all ranks is identical, the choice between the two representations can become significant, if we consider the dynamics of the 'promotion' event. In the first representation, values of the rank attribute can be changed straightforwardly to model the effects of promotion. In the second representation, modeling a promotion from lecturer to professor requires destroying the `lecturer` object and creating a new `professor` object. But then how do we relate the new `professor` and the old `lecturer`, and how should we preserve the values of unaffected attributes common to all employees?

5.1 Classes and Types

The ability to change the class of an object provides support for object evolution. It lets an object change its structure and behavior, and still retain its identity. In [47], a type system which allows this kind of evolution is presented. An object x can have a set of types, and the change from one type to another is a process of selectively adding and deleting types to the set of types of x. The notion of typing is retained whilst allowing some flexibility in system evolution.

In our present framework there is no notion of type. We support the grouping of objects according to common structure and properties by means of class, which is a dynamic notion. This gives more flexibility for representing class changes. However there are other advantages to be gained from having a type system in addition. A further typing mechanism could be added as an extension to our basic data model. There is a tendency in the literature to use the terms 'type' and 'class' interchangeably. For us they are distinct notions: one (*type*) is a static, syntactic feature of the representation language; the other (*class*) is a dynamic grouping of objects according to their structure and properties.

5.2 Realization

We deal with the evolution of objects by allowing events that change an object's class, and some or all of its attributes. For example, graduation causes a student to change class. The effects can be described by removing the student object from class `student` and terminating those attributes he has by virtue of being a student; attributes he has by virtue of belonging to class `person` should however be retained. The selective termination of attributes is obtained by using schema information. In order to deal with this type of class change we introduce another predicate `removes` in place of `destroys`. Again, new predicate names are used in order to emphasize conceptual differences.

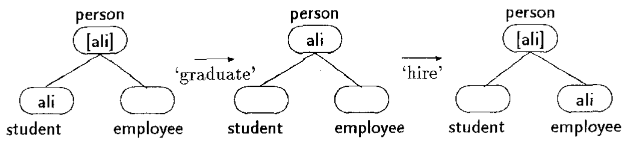


Fig. 3. Class changes.

Consider Fig. 3. A graduation event causes `ali` to move up the class hierarchy. When the student `ali` graduates, he is removed from `student` and becomes an instance of the `person` class only. His attributes by virtue of being a student are also terminated. The effects of the graduation event are described by the following rule:

```
removes(event:Ev[act ⇒ graduate,
        student ⇒ S], S, student).
```

There is in addition a general rule, that removing an object from a class terminates all attributes specific to instances of that class:

```
terminates(Ev, Obj, Attr, _) ←
      removes(Ev, Obj, Class),
      attribute(Class, Attr).
```

The overall effect of a graduation event for `ali` is that, for times after the graduation, it is no longer possible to derive `instance_of(ali, student)`, nor values for any of his `student`-specific attributes since these are all terminated automatically by the graduation event.

Now consider hiring `ali` as an employee. This will cause his class to be changed from `person` to `employee`. Since the class `employee` has some additional attributes `(dept, rank)`, the specification of this event will include values to initiate these attributes. Thus the effects of the hiring event are described by assigning him to the class `employee`, and initiating his `employee`-specific attributes. The description of such an event might be:

```
event:e21[act ⇒ hire, object ⇒ ali,
          dept ⇒ cs, rank ⇒ lecturer]
```

The effects of hiring events in general can be specified as follows:

```
assigns(event:Ev[act ⇒ hire,
        person ⇒ P], P, employee).
initiates(event:Ev[act ⇒ hire, person ⇒ P,
          dept ⇒ D], P, dept, D).
initiates(event:Ev[act ⇒ hire, person ⇒ P,
          rank ⇒ R], P, rank, R).
```

Note that in changing `ali`'s class first from `student` to `person` then from `person` to `employee`, `ali` has not been removed from the class `person` and has retained all his `person`-specific attributes. More importantly the identity of the changing `ali` object remains the same throughout.

We have described moving an object up and down the class hierarchy by two separate event occurrences. We can also imagine a single event which would cause an object to change its class from `student` to `employee` directly ('hire-student' say). The effect of this type of event could be specified as follows:

```
removes(event:Ev[act ⇒ hire-student,
        student ⇒ S], S, student).
assigns(event:Ev[act ⇒ hire-student,
        student ⇒ S], S, employee).
```

As in the case of two separate events, we do not lose the values of the `person`-specific attributes, and we do not remove the object from class `person`.

The question naturally arises of what happens to attribute values as the object moves across the hierarchy. In our framework, the relationships between old and new values in the sibling classes, if any, can be specified explicitly using `initiates` statements, just as in the specification of the initial state of a newly 'created' object (e.g., 'hiring' above). There is nothing special about class-changing events in this respect. We do not believe that any useful general rules can be formulated, even for the case where the sibling classes contain attributes with the same name. It might be supposed that in such a case the values of the common attributes should remain unchanged. We believe this would be a mistake. If the common attribute has a different intended meaning in the two classes, then there is no reason why the two values should be the same, except by coincidence. In the case where the common attribute does have the same intended meaning in both classes (e.g., if an attribute `age` indicates the age of a student and also the age of an employee), then this suggests an inadequacy in the modeling scheme itself. If we want such attributes to retain their values during a class change, then they should belong to a common superclass of the two classes involved. (In the example, `age` should be an attribute of `person` and not of the more specialized subclasses `student` and `employee` separately.) The whole point is that common attributes should be defined as part of the structure of a general class, with each sub-class further introducing the additional attributes specific to its instances.

We have illustrated three kinds of simple class change: 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 the general case, changing an object from class *C1* to class *C2* involves finding a path in the class hierarchy and using rules similar to the preceding ones to move along this path.

5.3 Remarks

In general, class changing events affect both the internal state of objects and also the class membership relation. The `holds_at` and `holds_for` clauses for deriving internal states of objects require no modification, since the effect is accommodated by making `terminates` dependent on `removes` as shown already. However we do need to modify the clauses for `instance_of` (and `instance_for`) to take `removes` into account. The modified formulation of `instance_of` is as follows:

```
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, Class).

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

The last clause states that if an event destroys an object, then that event also removes the object from all its classes.

There are two details left, one comparatively trivial and one more substantial. We take them in order.

In the formulation as we have presented it, the general recursive rule for `assigns`, which describes the subset relation (see Section 4.3), causes the assignment of an object to a class redundantly when there are also other more specific `assigns` statements present. For instance when hiring person x as an employee, a new period of time for the fact `instance_of(x, person)` is initiated even though this fact is already current (`x` is already an instance of `person` when the hiring takes place). This duplication of periods occurs with every class-changing event for which an `assigns` statement assigns an object to a subclass of its current class. The problem manifests itself when the database is queried about the `instance_for` relation because then several different but overlapping periods of time can be generated as answers. There are various solutions to this problem. The simplest is to take into account the possibility of these different time periods in the formulation of `instance_for` so that all these separate periods are amalgamated into one. (This requirement has been termed *coalescing* in the temporal database literature [15].)

The second point is more substantial. Allowing objects to change their class presents a potential problem which is analogous to dangling references. The problem arises when an object, which is the value of an attribute `Attr` of some other object, changes class in such a way that it can no longer be regarded as a meaningful value for attribute `Attr`. For example, assume that the staff instances have an attribute `student` which takes a student as a value. Further assume that the student `ali` is a student of the staff member `john`. When `ali` graduates and changes class to employee, the student attribute of `john` is not valid any longer and should be terminated. The graduation, which is defined as a class-changing event, takes care of terminating and initiating attributes of the student, but the other objects referring to this object as a student are not changed. This problem, sometimes called the *dangling domain problem* [47], would be a type violation in a typed system. In our present framework the problem can be avoided by writing event specifications appropriately, but this obviously requires that all 'dangling domain' problems are identified and accounted for explicitly. This is clearly unsatisfactory. The natural solution is to refine the schema so that the type (or perhaps class) of the value of an attribute is specified as well. In principle, this additional schema information would allow for a reformulation of the OEC so that attributes affected by the 'dangling domain' are terminated automatically in the same kind of way that 'dangling references' are eliminated. However, the details turn out to be quite complicated, and we have not yet explored all the possibilities.

6 Further Considerations

For the purpose of focusing attention on the different kinds of changes to objects and how they can be formulated, we have presented a simplified form of the OEC so far. In this section we sketch how the OEC is modified to provide other features which are required for practical applications. The two main extensions to the basic data model are to allow *multivalued* as well as single-valued attributes, and to support a wider range of *methods* than those introduced so far, specifically for deriving new information from the existing states of objects. We also describe how versioning of objects and the schema can be accommodated within the OEC framework.

6.1 Multivalued Attributes

Multivalued attributes are supported straightforwardly in our framework, since it is actually single-valued attributes that are the special case and that impose additional requirements. We do not attempt to support *set-valued* attributes.

A multivalued attribute denotes a one-to-many relation which maps the identity of an object to one or more objects. Such a relation can be thought of as a set of binary predicates, as in C-logic [7], for example. Indeed in C-logic *all* attributes are multivalued, since an attribute in that language is semantically the same as a binary predicate. Thus the C-logic term

```
person:john[children ⇒ {tom, sue, mary}].
```

is just a shorthand notation for the complex term

```
person:john[children ⇒ tom,
            children ⇒ sue,
            children ⇒ mary].
```

which is semantically equivalent to the following set of assertions in first-order logic:

```
children(john, tom).
children(john, sue).
children(john, mary).
```

This notion of multivalued attribute should be contrasted with approaches where (single-valued) attributes are allowed to take *sets* of objects as values. Set-valued attributes provide increased expressive power (they are no longer just first-order) but at the cost of introducing the very severe semantical and computational problems associated with sets and set unification [29]. As argued in [7], the simpler notion of multivalued attribute already satisfies most of the practical and expressive requirements of set-valued attributes, at enormously reduced complexity.

The reformulation of the object-based event calculus to allow for multivalued attributes requires just a slight generalization of what has been presented in the previous sections. The main adjustment is a refinement of the clause which is used to implement the functionality constraint for single-valued attributes, viz:

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

If we wanted to support only multivalued attributes then it would be sufficient to delete the 'functionality' clause altogether and retain the rest without change. Since we want to support both single and multivalued attributes in one system, the obvious solution is to make the clause above applicable to single-valued attributes only. For this it is necessary to extend the schema information so that it also specifies whether an attribute (in a given class) is single-valued or not. A third argument is therefore added to the predicate `attribute` to specify the kind of the attribute: `single` or `multi`.

The clause implementing the functionality constraint now requires an extra condition so that it does not apply to multivalued attributes. Since the kind of an attribute (`single` or `multi`) may depend on class as well as the attribute name, it is necessary to include the class name as an additional argument in the `initiates` and `terminates` predicates. The modified clause is:

```
terminates(Ev, Class, Obj, Attr, _) ←
      attribute_of(Class, Attr, single),
      initiates(Ev, Class, Obj, Attr, _).
```

Including the class name as an argument in the `initiates` and `terminates` predicates makes it necessary to specify the class of an object at query time. Moreover, the clauses of the object-based event calculus presented earlier all need to be adjusted to take the presence of this new class argument into account. This is a very simple modification that raises no additional questions and so we do not show the whole modified version of the OEC again.

6.2 Derived-Attribute Methods

In object-oriented systems, methods are operations to describe the behavior of objects. This includes both modification and manipulation of the state of objects. We have so far considered only methods that modify the state of objects. Now we want to extend the kind of methods that are supported to include also methods which can derive new information from the existing state of objects. We call such methods *derived-attribute methods* or sometimes just *rules*. For instance the age of a person can be derived from the date of birth; the boss of an employee can be derived from the manager of his department, and so on.

Derived-attribute methods are included in the schema definition according to the classes. There have been numerous proposals for how to define and implement such methods in a logic programming framework (e.g., [9], [13]). Representing methods as deductive rules is the most common approach in the existing languages and the one we follow here.

The definition of derived-attribute methods can be given in a syntax similar to C-logic or other object-logic languages. For instance, in F-logic [23], the boss of an employee is defined by the following rule:

```
E[boss ⇒ M] ←
     E:employee[dept ⇒ D:dept[manager ⇒ M]]
```

stating that the boss is the manager of the department in which the employee works. This kind of syntax can be translated straightforwardly into an internal form which is manipulated by the object-based event calculus. The head of the rule contains the name of the derived attribute and the body contains the object-attribute-value information. Each rule is associated with a class. We employ the following representation in order to index methods by the class names:

```
method(Class, Obj, AttrName, Value, Body).
```

The first argument, `Class`, refers to the name of the class for which the method is defined. `Obj` is the identity of the object for which the method is invoked. The third argument, `AttrName`, is the name of the derived attribute (in the object-oriented terminology this corresponds to the message used to invoke the method). The fourth argument, `Value`, denotes the value returned as the result of the method. And finally, `Body` is the translated form of the body of the rule.

For the purposes of this paper we assume that the body of a derived-attribute method is a conjunction of complex object terms (i.e., object-attribute-value information). Again following C-logic [7], the relational semantics for complex object terms allows every such term to be decomposed into a conjunction of atomic object terms, and so the body of a rule can always be expressed as a conjunction of conditions of the form `o-term(Class, Obj, Attr, Val)`. Hence the example above would be translated into the form:

```
method(employee, E, boss, M,
        [o-term(employee, E, dept, D),
         o-term(dept, D, manager, M)]).
```

Note that conjunctions are here represented as *lists*. Given such a representation, we now require the ability to compute not only the conclusions that are derivable from the rules, but also the time periods for which such conclusions hold.

In its full generality, temporal reasoning with derived information raises a number of unresolved questions which are the subject of much current research (see [39] for a detailed treatment). The usual solution, and the simplest, is to distinguish between *base* and *derived* information and treat these separately. The effects of changes (here, events) are then described by specifying only how they affect *base* information. The effects on derived information are obtained indirectly, because derived information is computed from the base information when it is required.

This is the approach that we follow also. We do not specify how events affect derived attributes directly. Values of derived attributes at any given time are determined by finding the corresponding method in the schema definition and then solving each condition of the rule body at the specified time instant. This is accomplished by adding another clause to the definition of `holds_at`:

```
holds_at(Class, Obj, Attr, Val, T) ←
    method(Class, Obj, Attr, Val, Body),
    solve_at(Body, T).
```

Here `solve_at(Body, T)` is a 'meta-interpreter' which executes the `Body` at the specified time `T`. More general forms of rules for derived attributes can be accommodated by allowing other kinds of conditions in the body of the rules. It is also possible to derive the period of time for which a derived attribute takes a particular value (see [22] for details).

6.3 Versioning of Objects

In this section we describe how versioning of objects can be formulated within the OEC. The formulation of the OEC presented so far provides us with the state history of objects, and already supports the simplest kind of versioning. Checkpointing and parallel versioning can also be accommodated in the object-based event calculus with some modifications to the original. These various modifications are presented and discussed in more detail elsewhere [20]; here we summarize the modifications for parallel versioning only.

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. Also, as versions share some of their properties with the object from which they are derived, there must exist an easy way of finding the previous version. For this purpose we introduce a naming convention that uniquely identifies the versions. Suppose we have an object with the identity `o` and we create versions of this object. One way of distinguishing these versions is to number them. The first version of `o` will be `v(o,1)`, the second `v(o,2)`, the nth `v(o, n)` and so on. When versions of versions are created, the same naming convention can be used to identify the new versions. The first version of the object `v(o,1)` for example, will be named as `v(v(o,1),1)`, the second will be `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 where each history is identified by a version identity. Each version has its own history starting from its creation time. Once a version is created it is treated in the same way as other objects in the database: it can be updated, deleted or versioned. Meanwhile its parent object can be directly updated, even after one or more versions have been derived from it: the derived versions will not be able to see the updates in the parent object.

Creation of a version is described by events. In parallel versioning 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. The effects of version creating events are specified by the predicate `creates_version` which is used to mark the occurrences of such events in the object's 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 every time the user performs a "reconfigure" operation. This can be achieved by the following rule:

```
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 `C` denotes the parent object, and `N` is an integer denoting the version number.

Versions are made instances of the class to which their parent version belongs by using the `assigns` predicate. The version objects can be changed like any other object. Reasoning with the changing state of versions is done by similar axioms, but now the time of creation of the version is taken into account as well. The values of the attributes which are not changed at the creation time or later are derived using the parent version.

Consider Fig. 4 which shows a section of an object's version derivation history. Here `Vid` is a version identifier and `Oid` is the object from which it is derived. `Oid` can be another version or the initial object. `Tc` denotes the creation time of the version `Vid`, `T` denotes the time at which we query its state. The history of `Vid` starts at `Tc` and at that time `Vid` as its initial state has the same state as `Oid` has as of time `Tc`. At any time after its creation, say at `T*`, the version's state can be changed by an event. If no such event happens between `Tc` and `T`, then the state of the version at time `T` is the same as the state of the object `Oid` at time `Tc`. However, if there are some events that have happened after `Tc` and have changed the values of one or more attributes of the version, then we have to consider the effects of these events as well. The reader is referred to [20] for the modified formulation of `holds_at` to reason with the state of objects and versions.

6.4 Schema Evolution

In this section, we look at the problem of changes in class definitions (i.e., the database schema). We address the problem of maintaining consistency between a set of objects and a set of class definitions that can change.

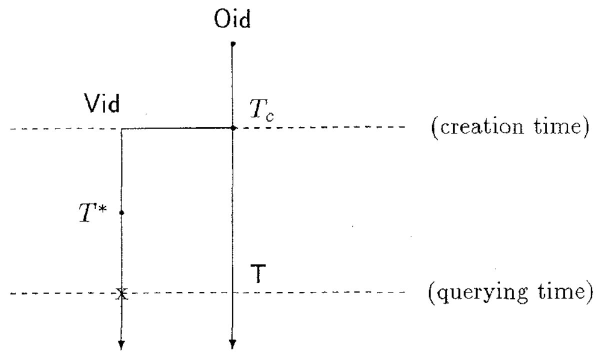


Fig. 4. A section of an object derivation hierarchy.

6.4.1 Problem Definition

There have been several proposals for supporting schema evolution in object-oriented database systems. (See e.g., [32] for a bibliography.) Most of the existing works support single schema modification. That is, at every time instant, there exists only one logical schema that can be used to view the objects. Past states of the schema are not retained. There are two shortcomings to single schema modification. One is that it does not allow the history of modification of objects to be preserved. For example if an attribute of a class is dropped, the values of the attribute in existing instances are irretrievably lost; even if the attribute is added later, it will be treated as a different attribute and the old values of the attribute cannot be seen. Another shortcoming is that it does not prevent a schema change by one user from impacting all other users' views of the database. For example, once any user deletes an attribute or changes the superclass/subclass relationship between a pair of classes, all other users will see the changes.

There is also another dimension to the schema evolution problem: *schema versioning*, where all states of the schema are accessible. In this case the system will manage more than one logical schema for one common database and present different views of the database through different versions of the schema. Schema versioning removes the shortcomings of single schema modification. For example, if a user wishes to drop an attribute from a class in one version of the schema, he may create a new version of the schema. The user will not see the values of an attribute in existing instances of the class. However if the user later chooses to access the class through the previous version of the schema, he will be able to see the values of the attribute in all instances of the class that existed before he created the new version of the schema.

We have extended the OEC to describe changes to the schema so that we can keep all states of the schema as well as the object states. This leads us to model schema versioning in a deductive framework. Our approach is different from the existing approaches since we develop the idea of having time-dependent views of the database.

6.4.2 Realization in the Event Calculus

We now have two levels of data that change: schema and objects. We use the OEC to describe changes at both levels. The object state history is described by a set of real world events and the schema state history is described by a set of system events. This provides us with schema versioning.

When we allow schema modifications, we have to consider a time-dependent `is_a` relationship and time-dependent class definitions. Schema changing events initiate and terminate periods of time for which a class is a subclass (`isa_at`) of another class or a class has a certain attribute or method. We introduce time arguments to the `is_a` and `attribute` relations to model this time-varying behavior:

```
attribute_at(Class, Attr, Type, T):
    Attr is an attribute (single or multi)
    of Class at time T.

method_at(Class, Obj, Mesg, Val, Body, T):
    Mesg is a method valid for Class at time T.

isa_at(Class, Super, T):
    Super is a superclass of Class at time T.
```

With these predicates it is possible to keep the history of the class hierarchy and class definitions. In the following, the clauses for `is_a` are presented. We have omitted the clauses for the other two predicates which are similar.

```
isa_at(Class, Super, T) ←
        shappens(Ev, Ts), Ts ≤ T,
        adds(Ev, Class, Super),
        not dropped(Class, Super, Ts, T).

dropped(Class, Super, Ts, T) ←
         shappens(Ev, T*), Ts ≤ T* < T,
         drops(Ev, Class, Super).
```

The occurrence of a schema event is recorded by the predicate **shappens**; a new predicate is introduced in order to avoid unnecessary search of real world events (recorded with `happens`) when the schema information is derived. The role of the predicates **adds** and **drops** is analogous to that of the predicates **initiates** and **terminates** respectively.

Having two time dimensions gives the user the ability of querying the state of an object at time `T` according to the schema at time `Ts`. Thus a query:

```
?- holds_at(Class, Obj, Attr, Val, T, Ts).
```

asks for the state of an object `Obj` at time `T` according to the schema at time `Ts`. The period of time for which an object holds a particular state can be queried in a similar fashion:

```
?- holds_for(Class, Obj, Attr, Val, P, Ts).
```

In these queries the schema time `Ts` can be seen as a filtering mechanism for viewing the state of objects. The facts about an object are visible only if the definition of the class of the object exists in the schema version at the specified time.

Existing proposals in the literature provide different approaches to schema versioning. For example, in [36] each class, rather than the entire schema, is treated as a versionable object. Since the schema itself is not versioned, a 'virtual' version of the schema is constructed as a lattice of versioned class objects, having only one version of a class object included in any 'virtual' version of the schema. In [24], the entire schema is viewed as a versioned object. Any number of new versions of schema may be derived at any time from any existing schema version. The access scope of a schema version is the set of objects created under that version and those objects in the inherited access scopes of the ancestor schema versions. Thus, it is possible to view and manipulate different sets of objects under different versions of schema.

We have presented a different approach for dealing with versions of schema. We keep the history of the database schema by recording every event that causes a change in the schema. Thus it is possible to access different states of the schema at different times. The access scope of each schema version is the subset of all objects in the database whose classes are defined by the schema version. All updates to objects under one schema version will become visible to all schema versions which include the definition of the classes of the objects.

7 Implementation Issues

The OEC presented in the preceding sections can be implemented in different ways. It can be executed as a Prolog program, by transcribing the clauses given into Prolog syntax, although for reasons identified below the performance of the resulting program is likely to be poor, even in small examples. For practical applications we make use of a Prolog implementation that incorporates an additional 'tabulation' or 'lemma generation' mechanism, described presently. The OEC could also be implemented in one of the object-oriented logic programming languages now available (see Section 2) in order to provide the kind of 'clustering,' normally associated with object-oriented implementations; we have conducted some preliminary experiments using L&O [31] as an implementation language. Or the OEC as presented could be seen as an (executable) specification and suitable algorithms constructed to perform the same tasks in a procedural language.

We have employed the OEC in a number of small–medium applications, of which the most developed is a database dealing with the activities of a university department (from which the examples in this paper have been taken). To give some indication of size, object instances in these applications are numbered in hundreds and event occurrences in thousands. We have recently embarked on the reconstruction of an application in cardio-vascular medicine originally constructed using the standard (relational) event calculus [38].

The purpose of this section is to summarize the implementation techniques used in these applications. As indicated in the introduction to the paper, our aim is also to employ the OEC as the basis of large-scale temporal database systems; in this regard, we wish to explain why implementation techniques under development for temporal databases can be adapted for use with the OEC.

7.1 Current Implementations

The problems encountered in the efficient execution of the OEC can be illustrated by reference to the execution of `holds_at` queries. Exactly similar points can be made for more general `holds_at` queries, for `holds_for`, and for the class membership analogues `instance_of` and `instance_for`.

Obviously, there is the problem of searching efficiently for relevant candidate events that could initiate or terminate the value of the attribute in question, for which some form of indexing is required. But the main factor affecting performance of the OEC (and of the standard relational versions of the event calculus) is the need to determine what the effects of each such candidate event are (what it initiates and terminates). In general, this is not just a matter of looking up the `happens` assertions (the 'journal' of event occurrences), since to determine whether an event actually does initiate or terminate a value for a given attribute may require some further computation; and in a naive implementation this in turn may generate re-computation of the same facts over and over again, whenever `initiates` or `terminates` statements are context-dependent, that is, in the case where the value initiated or terminated by an event at time `t` depends on what other values are current at time `t`. In these circumstances naive execution of the event calculus clauses can lead to very severe redundancies in the computation, significantly affecting performance even in small applications.

Dramatic improvements can be obtained by adding a bottom-up component to the evaluation mechanism, by incorporating some form of 'lemma generation.' This is a standard technique in logic programming, and in deductive databases where it is often referred to as 'tabulation' [42]. For the OEC implementation, all facts regarding the states of objects and the time periods initiated and terminated by the recorded events are stored as they are deduced. We shall refer to these tabulated facts as the Object DataBase (ODB) for convenience. The ODB can be generated bottom-up when a new event is assimilated into the database, or—as we prefer—by tabulation during top-down query evaluation. However it is produced, the ODB contains all the information about objects: their states, their classes and the necessary information to derive the time periods for which these hold. When a query is posed to the system only the contents of the ODB are accessed without searching all the events again.

The obvious set of 'lemmas' or 'tabulated results' to store in the ODB are all the time periods, analogously to the scheme proposed in [39] for (relational) temporal databases based on the event calculus. However we prefer an alternative which is much more flexible and easier to maintain: for each tuple of the form `(Obj, Attr, Val)` 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 are easily 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 and end times for each. A Prolog implementation with this mechanism and a simple form of indexing gives quite acceptable performance for the applications mentioned above. The table in Fig. 5 gives some sample timings. The queries were executed on a database of the kind used as the source of examples in this paper, containing approximately 10,000 events (1,000 objects). These timings are just intended to be indicative of performance. Further details regarding implementation techniques are provided elsewhere [21].

7.2 Future Work

One of our longer term aims is to develop the OEC so that it can support database applications proper. More sophisticated indexing techniques will then be required. We do not want to give the impression that we underestimate the difficulty of the task, but we do want to indicate why we believe this is not an unrealistic ambition. The point is that implementation techniques being developed for temporal (relational) databases are not incompatible with use of the OEC; many can be adapted, or even applied directly.

Query (all solutions)	Net time (millisec) (plain OEC)	(with ODB)
holds_at(manager1,name,N,100)	219.66	1.434
holds_at(manager1,A,V,100)	1679.20	13.165
instance_of(manager1,C,100)	207.33	0.783
instance_of(O,person,100)	5755.85	46.085
holds_for(student5,address,A,P)	26415.00	8.333
holds_for(student5,A,V,P)	1.797e+05	64.500
holds_at(assistant15,A,V,100)	152.34	13.432

The database contains approx. 10,000 events (1000 objects).
(Quintus Prolog Release 3.1.1 on a SUN Sparc workstation)

Fig.5. Some sample timings.

For example, the techniques described in [16] for implementing 'backlogs' can be applied directly. Although these techniques are designed for the implementation of *transaction-time* databases, the feature on which they rely is the 'stepwise-constant' nature of the time-varying data. Since this is also the kind of change supported by the OEC, the same techniques are applicable for implementation of the ODB: indeed, the structures we store in the ODB have (almost exactly) the form of backlogs.

Indexing methods, such as the B^+-tree techniques described in [11], can also be applied, with some modification, to the ODB, or more directly, to provide indexing on the record of event occurrences (the 'journal'). Indeed, a very rudimentary form of this idea is the basis of an event calculus implementation described in [14]. Development of these ideas, and of associated query optimization techniques, remain topics for future work, however.

In addition to temporal indexing, some form of structural indexing—some method of storing the objects and events so that search is restricted to the potentially relevant candidates—is also required. At present all information in the 'journal' of event occurrences and in the ODB takes the form of relational tuples. (In the Prolog-based implementation simple forms of indexing these tuples have proved adequate.) Beyond some preliminary experiments with the L&O language [31], as mentioned above, we have not addressed the question of how to provide indexing methods that reflect the object-oriented structuring at the implementational level.

8 Conclusions

The integration of deductive and object-oriented approaches presents many open questions and issues to be overcome. The main problem is caused by the opposition between a value based approach and an identity based approach. After an analysis of existing proposals, we were attracted by the virtues of the simplest approaches where a framework for complex objects is built within first-order logic. The formula approach, exemplified by C-logic, gives a semantics to an object by viewing it as a named collection of atomic formulas. Based on this approach we presented an object-based data model with support for object identities, single-valued and multivalued attributes, class hierarchies, and derived-attribute methods.

The main contribution of this paper is the detailed development of the Object-based Event Calculus (OEC), which is intended as a general approach for representing and manipulating temporal objects in a logic programming framework. We have shown how the OEC may be used to represent and manipulate complex objects in a natural and descriptive way. We are not aware of any other work which deals with the different kinds of changes to objects in a single logical framework. We are also not aware of any other work which incorporates temporal information to the object states in a deductive framework.

From the representational point of view, there are some benefits offered by the OEC over the standard relational versions of the event calculus. Organizing the specification of events by the class of object affected gives more structure to the representation, which can be of significant value in practical applications. Other benefits arise because the structure of objects, attributes, values and classes is richer than that of the relational data model, and this structure can be exploited. For example, the use of single-valued attributes and their treatment within the OEC reduces the need for general forms of integrity constraint, which otherwise are required for use with the event calculus. Similarly, much of the detail in the formulation of the OEC is concerned with the indirect effects of creating and deleting objects which themselves may be the values of attributes of other objects. As a result, the grouping of objects into classes and subclasses gives a comparatively simple device for dealing with some of the more common types of indirect change, or 'ramification,' a problem which in its general form is a topic of much current research in temporal reasoning.

From the point of view of temporal databases, we have presented an approach to the construction of historical ('valid time') databases in which all states of objects are stored (implicitly) and are accessible (by deduction). We addressed several different kinds of change that can be identified in the context of an object-based data model, and we proposed a general approach for modeling these changes in a declarative way. For practical application, we were particularly concerned to explain how the first-order semantics of objects allows implementations of the OEC to take advantage of indexing and other implementational techniques that are being developed for (relational) temporal databases.

ACKNOWLEDGMENTS

The bulk of this work was completed while Nihan Kesim was supported by the Scientific and Technical Research Council of Turkey (TUBITAK). The authors would like to thank an anonymous referee for suggesting a number of useful references to recent work in temporal databases and for many helpful suggestions which have improved the presentation of the paper.

REFERENCES

[1] S. Abiteboul and S. Grumbach, "COL: A Logic-Based Language for Complex Objects," *Proc. Int'l Conf. Extending Database Technology—EDBT '88*, pp. 271–293, Venice, Italy, Mar. 1988.

[2] H. Ait-Kaci and R. Nasr, "Login: A Logic Programming Language with Built-In Inheritance," *J. of Logic Programming*, 1986.

[3] M. Atkinson, et al., "The Object-Oriented Database System Manifesto," *Proc. First Int'l Conf. Deductive and Object-Oriented Databases*, pp. 40–57, 1989.

[4] F. Bancilhon, "Object-Oriented Database Systems," *Proc. Seventh ACM-SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pp. 152–162, Austin, Texas, Mar. 1988.

[5] C. Beeri, "Formal Models for Object Oriented Databases," *Proc. First Int'l Conf. Deductive and Object-Oriented Databases*, Kyoto, Japan, Dec. 4-6, 1989.

[6] W. Chen, M. Kifer, and D.S. Warren, "Hilog: A First-Order Semantics for Higher-Order Logic Programming Constructs," *North American Conf. Logic Programming*, Oct. 1989.

[7] W. Chen and D. Warren, "C-Logic of Complex Objects," *Proc. Eighth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, 1989.

[8] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo, "The LDL System Prototype," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 1, pp. 78–90, Mar. 1990.

[9] M. Dalal and D. Gangopadhyay, "OOLP: A Translation Approach to Object-Oriented Logic Programming," *Proc. First Int'l Conf. Deductive and Object-Oriented Databases*, pp. 555–568, Kyoto, Japan, Dec. 4-6, 1989.

[10] R. Elmasri, G.T.J. Wuu, and V. Kouramajian, "A Temporal Model and Query Language for EER Databases," A. Tansel et al., eds., *Temporal Databases: Theory, Design, and Implementation*, chapter 9. Benjamin/Cummings, 1993.

[11] R. Elmasri, G.T.J. Wuu, and V. Kouramajian, "The Time Index and the Monotonic B+ Tree," A. Tansel et al., eds., *Temporal Databases: Theory, Design and Implementation*, chapter 18, pp. 433–456. Benjamin/Cummings, 1993.

[12] C. Evans, "The Macro-Event Calculus: Representing Temporal Granularity," *Proc. Pacific Rim Int'l Conf. AI*, pp. 363–368, Nagoya, Japan, 1990.

[13] K. Fukunaga and S. Hirose, "An Experience with a Prolog-Based Object-Oriented Language," *OOPSLA '86 Proc.*, pp. 224–231, 1986.

[14] R.V. Indiketiya, "Event Calculus Based Temporal Database Management System," Master's thesis, Imperial College, 1992.

[15] C.S. Jensen, J. Clifford, R. Elmasri, S.K. Gaida, P. Hayes, and S. Jajodia, "A Glossary of Temporal Database Concepts," *SIGMOD Record*, vol. 23, no. 1, Mar. 1994.

[16] C.S. Jensen and L. Mark, "Differential Query Processing in Transaction-Time Databases," A. Tansel et al., eds., *Temporal Databases: Theory, Design, and Implementation*, chapter 19. Benjamin/Cummings, 1993.

[17] C.S. Jensen and R.T. Snodgrass, "Unifying Temporal Data Models Via a Conceptual Model," *IEEE Information Systems*, vol. 19, no. 7, pp. 513–547, 1994.

[18] W. Kaefer, N. Ritter, and H. Schoening, "Support for Temporal Data by Complex Objects," *Proc. 16th Int'l Conf. Very Large Data Bases*, Brisbane, Australia, 1990.

[19] F.N. Kesim and M. Sergot, "On the Evolution of Objects in a Logic Programming Framework," *Proc. Int'l Conf. Fifth Generation Computer Systems*, vol. 2, June 1992.

[20] F.N. Kesim and M. Sergot, "Versioning of Objects in Deductive Databases," *Proc. Third Int'l Conf. Deductive and Object-Oriented Databases*, Dec. 1993.

[21] F.N. Kesim and M. Sergot, "Implementing an Object-Oriented Deductive Database Using Temporal Reasoning," Technical report, Bilkent Univ., Dec. 1994.

[22] F.N. Kesim, "Temporal Objects in Deductive Databases," PhD thesis, Dept. of Computing, Imperial College, 1993.

[23] M. Kifer and G. Lausen, "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme," *Proc. Eighth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pp. 134–146, 1989.

[24] W. Kim and H.T. Chou, "Versions of Schema for Object-Oriented Databases," *Proc. 14th Int'l Conf. VLDB*, pp. 148–159, Los Angeles, 1988.

[25] N. Kline, "An Update of the Temporal Database Bibliography," *SIGMOD Record*, vol. 22, no. 4, Dec. 1993.

[26] R.A. Kowalski, "Database Updates in the Event Calculus," *J. of Logic Programming*, vol. 12, pp. 121–146, 1992.

[27] R.A. Kowalski and F. Sadri, "The Situation Calculus and Event Calculus Compared," *Proc. Int'l Symp. Logic Programming*, pp. 539–553, MIT Press, 1994.

[28] R.A. Kowalski and M. Sergot, "A Logic-Based Calculus of Events," *New Generation Computing*, vol. 4, pp. 67–95, 1986.

[29] G.M. Kuper, "Logic Programming with Sets," *Proc. Sixth ACM-SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, San Diego, 1987.

[30] D. Maier, "A Logic for Objects," *Proc. Workshop Foundations of Deductive Databases and Logic Programming*, pp. 6–26, Washington D.C., Aug. 1986.

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

[32] J. Roddick, "Schema Evolution in Database Systems—An Annotated Bibliography," *SIGMOD Record*, vol. 21, pp. 35–40, Dec. 1992.

[33] E. Rose and A. Segev, "TOODM—A Temporal Object-Oriented Data Model with Temporal Constraints," *Proc. 10th Int'l Conf. on the Entity-Relationship Approach*, pp. 205–229, 1991.

[34] A. Segev and A. Shoshani, "A Temporal Data Model Based on Time Sequences," A. Tansel et al., eds., *Temporal Databases: Theory, Design and Implementation*, chapter 11, pp. 248–270. Benjamin/Cummings, 1993.

[35] M.P. Shanahan, "Representing Continuous Change in the Event Calculus," *Proc. ECAI-90*, Stockholm, Sweden, 1990.

[36] A.H. Skarra and S.B. Zdonik, "Type Evolution in an Object-Oriented Database," B. Shriver and P. Wegner, eds., *Research Directions in Object-Oriented Programming*, pp. 393–413, MIT Press, 1987.

[37] R.T. Snodgrass, "Temporal Object-Oriented Databases: A Critical Comparison," W. Kim, *Modern Database Systems: The Object Model, Interoperability and Beyond*, chapter 9. Addison-Wesley/ACM Press, 1994.

[38] P. Soper, G. Abeysinghe, and C. Ranaboldo, "A Temporal Model for Clinical and Resource Management in Vascular Surgery," *Proc. Int'l Conf. Database and Expert Systems Applications*, pp. 549–552, Berlin, 1991.

[39] S.M. Sripada, "Temporal Reasoning in Deductive Databases," PhD thesis, Dept. of Computing, Imperial College, 1991.

[40] J. Su, "Dynamic Constraints and Object Migration," *Proc. Conf. Very Large Data Bases*, Barcelona, Spain, 1991.

[41] A. Tansel et al., eds., *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.

[42] D.S. Warren, "Memoing for Logic Programs," *Comm. ACM*, vol. 35, no. 3, pp. 94–111, 1992.

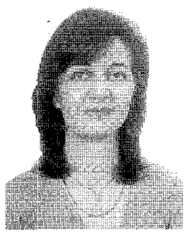
[43] G. Wiederhold, S. Jajodia, and W. Litwin, "Integrating Temporal Data in a Heterogeneous Environment," A. Tansel et al., eds., *Temporal Databases: Theory, Design, and Implementation*, chapter 22. Benjamin/Cummings, 1993.

[44] G.T.J. Wuu and U. Dayal, "A Uniform Model for Temporal and Versioned Object-Oriented Databases," A. Tansel et al., eds., *Temporal Databases: Theory, Design, and Implementation*, chapter 10. Benjamin/Cummings, 1993.

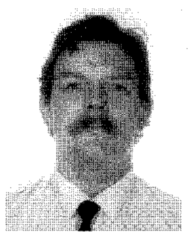
[45] C. Zaniolo, "Object-Oriented Programming in Prolog," *Proc. 1984 Int'l Symp. Logic Programming*, Atlantic City, New Jersey, Feb. 1984.

[46] C. Zaniolo, "The Representation and Deductive Retrieval of Complex Objects," *Proc. Very Large Databases*, pp. 458, Stockholm, Sweden, 1985.

[47] S.B. Zdonik, "Object-Oriented Type Evolution," F. Bancilhon and P. Buneman, eds., *Advances in Database Programming Languages*, pp. 277–288. ACM Press, 1990.



F. Nihan Kesim received the BS degree in computer engineering from the Middle East Technical University, Ankara, Turkey, in 1986; the MS degree in computer science from Bilkent University, Ankara, in 1988; and the PhD 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, University of London. He graduated in mathematics at Trinity College, Cambridge, in 1973; completed a postgraduate course in applied mathematics at Cambridge in 1974; and worked in mathematical modeling 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. His particular interests are in temporal reasoning, legal reasoning, and formal theory of organizations.