DATABASE UPDATES IN THE EVENT CALCULUS

ROBERT KOWALSKI

▷ This paper investigates a special case of the event calculus, concerned with database updates. It discusses the way relational databases, historical databases, modal logic, the situation calculus, and case semantics deal with database updates and compares the event calculus with the situation calculus in detail. It argues that the event calculus can overcome the computational aspects of the frame problem in the situation calculus and that it can be implemented with an efficiency approaching that of destructive assignment in relational databases. ◁

INTRODUCTION

The event calculus [9] was developed as a theory for reasoning about events in a logic-programming framework. It is based in part on the situation calculus [13, 14], but focuses on the concept of event as highlighted in semantic network representations of case semantics. Its main intended application is the representation of events in database updates and discourse representation. It is closely related to Allen's interval temporal logic [1, 2] and Lee-Coelho-Cotta's treatment of time in deductive databases [11]. The relationship between the event calculus and the systems of Allen and Lee-Coelho-Cotta has been investigated by Sadri [15].

The event calculus is concerned with formalizing the effect of events on objects, their properties, and their relationships. Thus, for example, given a description of an event in which

Bob gives Book1 to John,

Address correspondence to Robert Kowalski, Department of Computing, Imperial College of Science and Technology, 180 Queen's Gate, London SW7 2BZ, England.

Received July 1989; accepted March 1990.

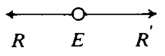
the event calculus derives that for a subsequent period of time, *initiated* by the event,

John possesses Book1

and for a previous period of time, *terminated* by the event,

Bob possesses Book1.

This can be pictured in the form

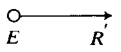


where E names the event, R names the terminated relationship, and R' names the initiated relationship.

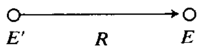
Although events can be assumed to take place instantaneously, the event calculus is actually neutral with respect to whether events are instantaneous or have duration. Relationships (including both properties of objects and relationships between objects), however, are assumed to hold for periods of time, which have duration. A relationship is assumed to persist both into the future until it is terminated by an event and into the past until it is initiated.

In this respect, past and future are treated symmetrically. Although such symmetry is very powerful in the general case, it gives rise to conceptual complexities and computational inefficiencies which are unnecessary in many cases.

In this paper we concentrate on a special, simplified, asymmetric case of the event calculus, where periods of time are assumed to persist only into the future. In this case, an event of Bob giving Book1 to John will automatically initiate a period of John's possession:



and end any earlier, unterminated period of Bob's possession:



This special case of the event calculus corresponds to the situation in conventional databases, where relationships are interpreted as persisting from the time they are recorded until the time they are deleted. It is because of this correspondence that we refer to this case of the event calculus as the *database update case*.

In the first half of this paper, we review the treatment of database updates in relational databases, historical databases, modal logics, the situation calculus, and case semantics. We focus particular attention on the frame problem in the situation calculus. In the second half of the paper, we introduce a special case of the event calculus and discuss its treatment of database updates, focusing on its efficient implementation.

This paper assumes some acquaintance with logic programming, but assumes no detailed knowledge of the other approaches.

RELATIONAL-DATABASE UPDATES

Updates in conventional databases are performed by adding or deleting tuples of relations. This can be viewed as adding or deleting variable-free atoms in a deductive database containing only variable-free atoms. The main advantage of this approach is its efficiency, in terms of both time and space. Perhaps its main disadvantage is that it confuses updates which *simulate* events in the world with updates which *describe* changes of belief.

Suppose, for example, that a relational database consists of the variable-free atoms

Possess(Bob Book1)
Possess(John Book2)
Possess(Mary Book3).

Consider the update

Delete Possess(Bob Book1)
Add Possess(John Book1).

This can represent either a change of state in the "world" due to the event described by

"Bob gave Book1 to John"

or a change of state of belief in the database about who possessed Book1 in the first place.

A conventional database only represents the current state of the world; information about past states is deleted when an update is performed, and there is an implicit restriction that updates can be performed only in the same order as the events they describe in the world. Thus, for example, if the sequence of two updates

(1) **Delete** Possess(Bob Book1)
Add Possess(John Book1)

(2) **Delete** Possess(John Book1)
Add Possess(Mary Book1)

describes events in the world, then there is an implicit assumption that the first event of Bob giving Book1 to John took place before the second event of John giving Book1 to Mary.

Addition and deletion of tuples as a way of performing updates need not correspond to semantically meaningful events. The update

Delete Possess(John Book1)
Add Possess(Mary Book4)

for example, is syntactically acceptable but semantically meaningless. This lack of semantic structure complicates the problem of maintaining database integrity.

Thus for the sake of the one *advantage* of efficiency of implementation, conventional relational databases have the *disadvantages* that:

They confuse simulation with description of the world.

Information about past states is destroyed when updates are performed. (Strictly speaking that is not entirely true, since most databases retain a log of previous transactions, which can be undone to restore the database to a previous state. However, even in this case a query can access at most one state at a time.)

Updates must be made in the same order as events occur.

Syntactically acceptable updates can be semantically meaningless.

HISTORICAL DATABASES

The first three of these disadvantages have been addressed by proposals to extend conventional databases by incorporating the concept of time. Some of these extensions [4, 16] are based on a modal logic which provides a new semantics for the database model; but most (e.g. [7, 18, 19, 20]) represent time by means of explicit temporal attributes, without changing the underlying semantic model. Snodgrass [18], for example, considers *historical databases*, which represent the times for which relationships are *valid* in the world; *rollback databases*, which represent the times for which relationships are *stored* in the database; and *temporal databases*, which represent both. Sripada [21] investigates an extension of the event calculus which represents both valid time and storage time. In this paper, however, we restrict ourselves to the representation of valid time, in order to facilitate comparison between the event calculus and other formalisms, and to focus more clearly on efficiency of implementation.

A historical database records the times at which a relationship starts and ends by means of explicit attributes. For example

```
Possess(Bob   Book1  (1 Sept 71)  ∞)
Possess(John  Book2  (25 Dec 76)  ∞)
Possess(Mary  Book3  (25 Dec 76)  ∞).
```

Here the third and fourth arguments record start and end times respectively. An end time of ∞ can be interpreted as indicating that the real end time is unknown and that the relationship is assumed to persist indefinitely into the future.

The temporal query evaluation procedure determines whether a relationship holds at a given time point by determining whether there is a tuple for the relationship whose start time is before or equal to the time point, and whose end time is after the time point. Because ∞ is later than every time point, forward persistence is an implicit property.

Updates that record changes in the world, rather than changes of belief, can be performed without deletion. At most a one-time replacement of an unknown end time by a real end time might be required. An update, for example of Bob giving Book1 to John on (1 Jan 77), is recorded by changing the unknown end time of Bob's possession to the real end time and by adding a record of John's possession:

Delete	Possess(Bob Book1 (1 Sept 71) ∞)
Add	Possess(Bob Book1 (1 Sept 71) (1 Jan 77))
Add	Possess(John Book1 (1 Jan 77) ∞).

The inclusion of explicit valid times overcomes many of the problems of conventional relational-database updates. Since time is explicit, historical information about past states is recorded and accessible. Moreover, relationships can be recorded independently of their order of occurrence. In particular, information about the future can be recorded along with information about the past, and previously missing information about the past can be added later.

It is also possible to distinguish between an update which represents a change of state in the world from an update which represents a change of belief, such as the change of belief that it was John rather than Bob who possessed Book1 in the first place:

Delete	Possess(Bob Book1 (1 Sept 71) ∞)
Add	Possess(John Book1 (1 Sept 71) ∞).

(In temporal databases, even this deletion of an erroneous record is avoided, by including two extra attributes for storage start and storage end times. Correction of an error is performed by a one-time replacement of an unknown storage end time, represented by ∞, with a real storage end time, which is the time at which the correction is made).

The main disadvantage of the historical (and temporal) database approach is that, because updates are performed by recording the starting and ending of relationships, the semantic structure of the events which terminate and initiate relationships is lost. Expensive integrity-checking procedures might be needed to ensure that updates correspond to semantically meaningful events.

The event calculus addresses this problem by changing the focus of attention from relationships to events. Updates are performed by adding event descriptions. The starting and ending of relationships follows as a logical consequence from event descriptions, by means of general rules which express the semantics of events. Moreover, these general rules have the same syntactic form as other rules in the deductive database.

Compared with historical databases, therefore, the main distinguishing features of the event calculus are that

the treatment of time is based on the notion of event,

events are used to give semantic structure by using general rules to derive the initiation and termination of relationships from event descriptions,

the database and the rules of the event calculus itself are formulated as a deductive database (or logic program).

TEMPORAL LOGIC

Reasoning about time is the subject matter of temporal logic. However, instead of representing time explicitly, temporal logic represents time by means of modal operators, such as "Past" and "Future" and "Always". For example,

Past(Possess(Bob Book1))
Future(Possess(John Book1)).

Advocates of the modal approach regard its proximity to natural language as its main *advantage*.

In contrast with conventional relational databases, the modal approach distinguishes between the times at which relationships hold in the world and the times at which information is assimilated. As a consequence, information about past states can be preserved, and information can be assimilated in an order which is different from the real-world occurrence of events. Thus, for example, given a database state consisting of the single sentence

Possess(John Book1),

we can have a sequence of updates

(1)	**Add**	Past(Possess(Bob Book1))
(2)	**Add**	Future(Possess(Mary Book1))

which first gives us information about the past and then gives us information about the future. The database state after the sequence of updates contains information about the past, present, and future. Such an ability to represent historical information is essential for representing discourse as well as for updating databases which contain incomplete information about the past.

The main *disadvantage* of the modal approach is that its references to time are context sensitive. As a consequence, complex changes may need to be made to the database to preserve its intended meaning when the context changes. Consider, for example, the modal database

Past(Possess(Bob Book1))
Possess(John Book1)
Future(Possess(Mary Book1))
Possess(John Book2)
Possess(Mary Book3).

To record the event

"John gives Book1 to Mary"

we would need to make a large number of changes to the database:

Delete	Possess(John Book1)
Add	Past(Possess(John Book1))
Add	Possess(Mary Book1)
Add	Past(Possess(John Book2))
Add	Past(Possess(Mary Book3))

The complexity of these changes is disproportionate to the semantic complexity of the event.

Because of context sensitivity, updates in the modal approach need to include both addition and deletion of sentences. As with relational databases, such updates can be semantically meaningless; moreover, because of their greater complexity, maintaining semantic integrity is also more complex.

The modal operators of temporal logic deliberately avoid explicit reference to time. They are not applicable therefore when time is an essential component of the information to be represented, e.g.

Bob gave Book1 to John on (1 Jan 77).

To augment the modal logic with terms or predicates explicitly denoting times defeats the purpose of using modal operators in the first place.

Thus the modal approach has naturalness of expression of temporal concepts as its most important *advantage*. As a result it can record information about past, future, and current states of the world, and it can record them in an order which is not restricted by the order of the occurrence of events. Moreover, because updates are performed by addition and deletion, it avoids, to some extent, the frame problem, which arises with the explicit treatment of time in the situation calculus.

The *disadvantages* of the modal operators are that

because they are context sensitive, even a simple change of context can necessitate a complex revision of the database,

they cannot naturally represent explicit references to time,

updates, consisting of additions and deletions of sentences, need not be semantically meaningful, and

existing proof procedures for modal logic are less efficient than proof procedures for classical logic in general and for deductive databases in particular.

The event calculus addresses these problems by representing both time and events explicitly. Database updates are given semantic structure by the use of general rules to derive information about relationships from descriptions of events. The event calculus borrows this structuring of updates from the situation calculus.

THE SITUATION CALCULUS

The situation calculus was introduced by John McCarthy [13] and developed by McCarthy and Hayes [14] as a general logical framework for reasoning about actions. It is still the subject of considerable theoretical investigation in artificial intelligence, and has recently been the focus of Hanks and McDermott's [6] analysis of problems with formalizing the notion of temporal projection. In this paper we investigate a version of the situation calculus, formulated within a logic-programming framework, based on that presented in [8], and oriented toward the treatment of database updates.

The situation calculus employs global states as explicit parameters of time-varying relationships. It treats events as state transitions. For example, the sequence of two events

E1:	Bob gives Book1 to John
E2:	John gives Book1 to Mary,

following the initial state

S0: Bob has Book1
John has Book2
Mary has Book3,

gives rise to the sequence of state transitions pictured in Figure 1.

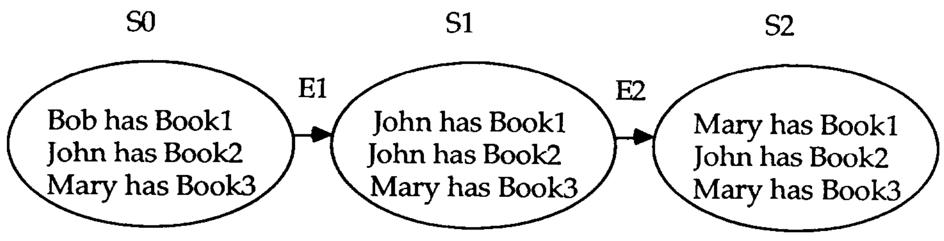
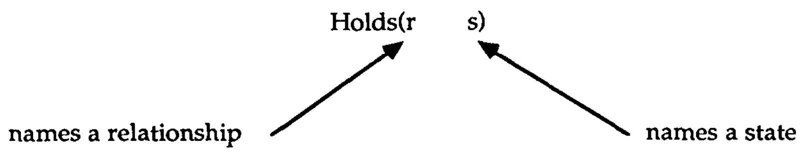


FIGURE 1.

As in [8], for the sake of generality it is convenient to employ a predicate which has parameters both for the name of the relationship and for the name of the global state in which the relationship holds:



Thus we write

Holds(possess(Bob Book1) S0)

instead of the weaker, but also adequate,

Possess(Bob Book1 S0).

In the first formulation, possess(Bob Book1) is a *term* which names a relationship. In the second, Possess(Bob Book1 S0) is an *atomic formula*. Both representations are expressed within the formalism of first-order, classical logic. However, the first allows variables to range over relationships, whereas the second does not.

If we identify relationships with atomic variable-free sentences, then we can regard a term such as possess(Bob Book1) as the *name* of a sentence. In this case "Holds" is a metalevel predicate, which can be viewed as a restricted form of the "Demo" predicate of [8]. Although the first argument of "Holds" is restricted to names of atomic sentences, we shall see later that this is not as restrictive as it might seem at first sight.

Whereas the initial state S0 is named by a constant symbol, noninitial states are named by means of a function symbol applied to the name of the previous state and the name of the event which constitutes the state transition, e.g.

S1 = result(give(Bob Book1 John) S0)
S2 = result(give(John Book1 Mary) result(give(Bob Book1 John) S0))

Notice that the first parameter *a*, of the term

results(*a s*)

actually names an event *type* rather than a concrete event *token*. For example, the term "give(Bob Book1 John)" names a type of event which can occur in different states and therefore can have several tokens. Only one event of a given type may occur in a given state. Thus the combination of an event type and a state constitutes a unique event token.

With this notation, the situation calculus enables us to describe the successive states S0, S1, S2 of our example:

Holds(R1 S0)	Holds(R1′ S1)	Holds(R1″ S2)
Holds(R2 S0)	Holds(R2 S1)	Holds(R2 S2)
Holds(R3 S0)	Holds(R3 S1)	Holds(R3 S2)

Here S1 and S2 are *abbreviations* for terms constructed by means of the function symbol "result", and

R1	abbreviates	possess(Bob Book1)
R2	abbreviates	possess(John Book2)
R3	abbreviates	possess(Mary Book3)
R1′	abbreviates	possess(John Book1)
R1″	abbreviates	possess(Mary Book1)

Thus in effect the situation calculus associates a global state as a kind of "context" or "time stamp" with every relationship that can be changed by an update. The intended interpretation of the time stamp is potentially ambiguous. Although we intend the state parameter to refer to states of the world, it can also refer to states of the database itself. Alternatively we can have two state parameters. For example

Holds*(*r s db*)

might express that

the database state *db* records the fact that relationship *r* holds in world state *s*.

In the remainder of this paper we shall assume that the state parameter, *s*, refers to "world states". Similarly, when we later discuss case semantics and the event calculus, we shall assume that event parameters refer to "world events" rather than "metaevents" of updating the database.

Like modal temporal logic, the situation calculus has the *advantage* that information about the past can be preserved when updates are performed. However, whereas the modal approach requires many sentences to be deleted and replaced by others, the situation calculus makes it possible to perform updates by adding sentences alone. Moreover, updates can be given semantic structure by using general rules to derive information about states from descriptions of events. For example, the relationship

Holds(possess(John Book1) result(give(Bob Book1 John) S0))

can be derived from the event description

Happens(give(Bob Book1 John) S0)

by means of the general situation-calculus rule

```
Holds(r result(a s))   if    Happens(a s)
                       and   Initiates(a r)
```

together with a domain-specific rule expressing that giving initiates possession:

Initiates(give(*x y z*) possess(*z y*)).

Explicit deletion of terminated relationships is replaced by implicit deletion of relationships which are not preserved. This is achieved by means of a situation-calculus rule called the *frame axiom*:

```
Holds(r result(a s))   if    Happens(a s)
                       and   Holds(r s)
                       and   not Terminates(a r)
```

together with domain-specific rules which describe the relationships terminated by events, e.g.

Terminates(give(x y z) possess(x y)).

The negative condition in the frame axiom can be interpreted by means of negation as failure.

Together the "Initiates" and "Terminates" relationships describe the *semantics* of events. Preconditions of events can be expressed as integrity constraints. A comparison between this treatment of preconditions, using integrity constraints, and the alternative treatment using conditions of general rules (as in [8]) is given in [10].

Because noninitial states are named by means of previous states, updates performed by adding event descriptions, e.g.

Add Happens(give(Bob Book1 John) S0)
Add Happens(give(John Book1 Mary) S1),

need to be made in the same order as events take place. Thus although the database preserves old information about the past, it cannot assimilate new information about the past. Moreover, there is an implied one-to-one correspondence between states of the world and states of the database, since the two are in synchrony.

Notice that, although the situation calculus only allows updates that explicitly change atomic sentences, derived relationships defined by means of general rules can be changed implicitly. For example if the database contains a rule which expresses that John always wants what he doesn't have, i.e.

Holds(wants(John x) s) if not Holds(possess(John x) s),

then explicit addition of the statement

Happens(give(John Book1 Mary) S1)

causes implicit addition of the conclusion

Holds(wants(John Book1) S2).

Notice that the first argument of the "Holds" predicate is restricted to a term naming an atomic formula. In a more general approach it would be possible to name a more general formula, as in the example

Holds(∀(X(wants(John X) $\leftarrow$ $\neg$possess(John X)))s).

In such a case, however, it would be necessary to have a metainterpreter. But then partial evaluation of the metainterpreter [22] would generate the situation-calculus clauses where the first argument of "Holds" names an atomic formula. For this reason, the situation-calculus restriction is not so great as it may seem at first sight.

The great disadvantage of the situation calculus is the so-called *frame problem.* The frame problem has two main aspects—one *epistemological*, the other *computational.* The epistemological aspect is the knowledge-representation problem of formalizing in a natural way that all relationships not terminated by an event are preserved. This problem is solved by our use of the single "Holds" predicate and negation as failure in the formulation of the frame axiom.

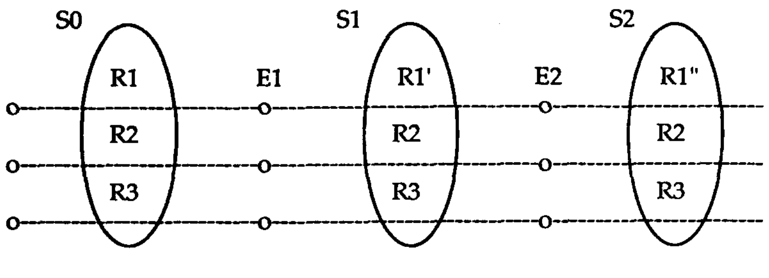


FIGURE 2.

Notice that the use of negation as failure also avoids the Hanks-McDermott problem [6], as discussed for example in [5] and [23].

The computational aspect of the frame problem is the excessive computational cost associated with using the frame axiom to reason that virtually all relationships are preserved from state to state. This overhead is illustrated in Figure 2, where the small circles represent the separate application of an axiom or inference step to conclude that a relationship holds in a given state. These inferences need to be performed whether the inference system reasons forward, deriving new states from old ones, or backward, logic-programming style, to determine whether a relationship holds in a given state by determining whether it held in the previous state.

A third aspect of the frame problem, the *ramification problem*, applies to our current formulation of the situation calculus: A derived relationship could persist even when the basis for its derivation is terminated. Thus, in our previous example, John would continue to want Book1 in state S1 even though he possesses it in state S1. This problem can be solved by restricting the frame axiom to nonderived relationships:

Holds(r result(a s)) if Holds(r s)
and Primitive(r)
and not Terminates(a r),

where "Primitive" only holds for nonderived relationships:

Primitive(r) if Holds(r S0)
Primitive(r) if Initiates(a r),

and no relationship is both primitive and derived.

Thus the two main *advantages* of the situation calculus are that it preserves information about the past and that updates have semantic structure. The initiation and termination of relationships is accomplished by adding event descriptions without performing explicit deletions. Moreover, because of the possibility of defining derived relationships by means of general rules, ramifications are dealt with automatically by the system rather than explicitly by the user.

The main *disadvantages* are that

the computational cost of the frame axiom is unacceptably high for an implementation of temporal reasoning,

because of the use of global states, events need to be totally ordered, and

it is not possible to assimilate new information about the past.

The event calculus addresses these disadvantages by associating *local* time periods rather than global states with relationships. These periods are a function of the event which initiates or terminates their associated relationships. Events themselves are given names, as is suggested by their representation in case semantics.

CASE SEMANTICS

Perhaps most natural-language-understanding systems developed in artificial intelligence are based upon some form of case semantics. Many of these are presented in a graphical, semantic-network notation. For example, the meaning of the sentence

"Bob gave Book1 to John"

might be represented by a collection of nodes denoting individuals and arcs denoting binary relationships as shown in Figure 3. The explicit treatment of the event (named E1 here) as an individual facilitates, among other things, the later addition of further information, e.g.

"It all happened in the park on (1 Jan 77)"

by adding extra nodes and arcs as shown in Figure 4. Thus updates are performed by adding information without deletion. Not only is past information preserved, but information can be added in any order, independently of the order of events.

Events can be temporally related to one another without having concrete times associated with them. For example,

"John gave Book1 to Mary after he read it"

might be represented by the network shown in Figure 5. Different event tokens of the same type can be distinguished without their needing to be associated with different times, e.g. Figure 6.

Events ordered by the "Earlier-than" relation need not be totally ordered and can be concurrent. A sequence of events which takes place in one part of the world need not be temporally related to a sequence of events taking place in another part. Thus the notion of event is fundmental, and different from the notion of time.

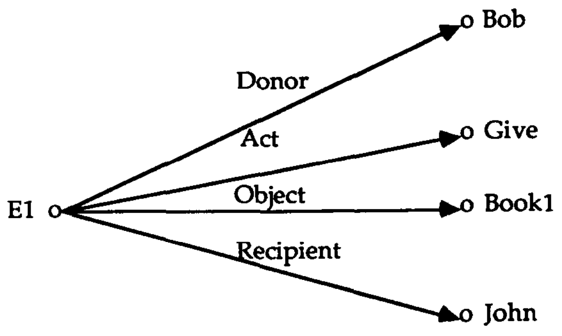


FIGURE 3.

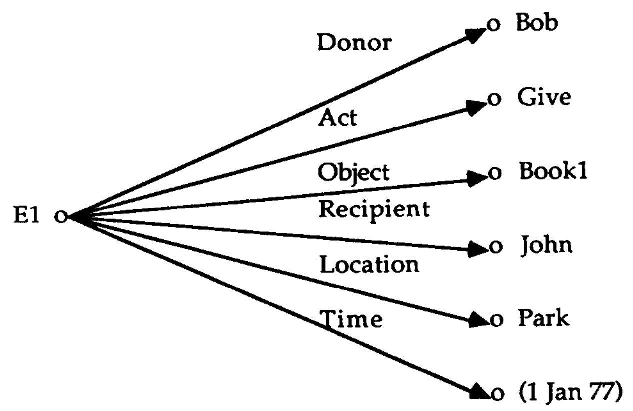


FIGURE 4.

The great *advantage* of the case-semantics approach is its ability to represent complex semantic situations. Its *disadvantages* result from the lack of a logical framework:

General rules cannot be conveniently represented.

General-purpose inference mechanisms are not well defined.

Because the connection between events and the relationships they initiate and terminate cannot be easily expressed, updates have no semantic structure.

These disadvantages can be overcome by reexpressing the concepts of semantic networks in a logical formalism. The semantic-network representations, in particular, can be translated into logic simply by using constant symbols in place of nodes and binary predicates in place of arcs. The network representing the semantics of

"Bob gave Book1 to John",

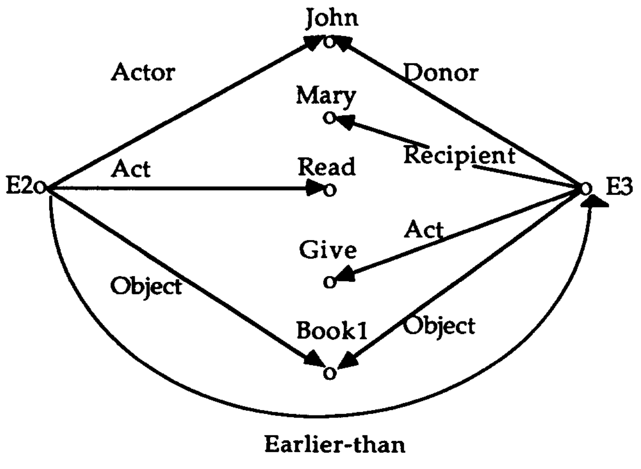


FIGURE 5.

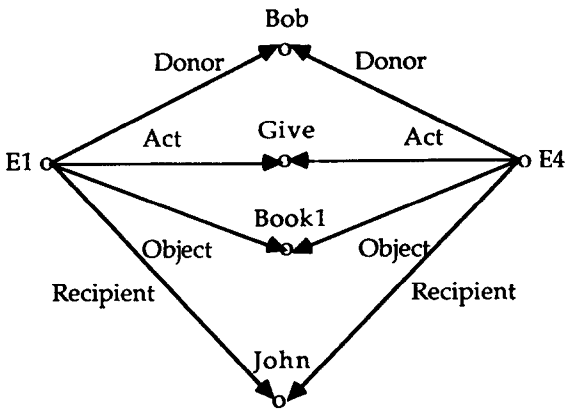


FIGURE 6.

for example, can be reexpressed by means of binary predicates:

Donor(E1 Bob)
Act(E1 Give)
Object(E1 Book1)
Recipient(E1 John).

A similar, but slightly less flexible, representation can be obtained by using predicates of greater arity, e.g.

Give(Bob Book1 John E1)

or

Happens(give(Bob Book1 John) E1).

Such a reformulation of case semantics in a logic-programming framework has been one of the major influences on the development of the event calculus. Other important influences have been the desire to associate time periods with relationships, as in Stamper's Legol [19] and Allen's temporal logic [1, 2], and the desire to structure updates by using general rules to connect events with the relationships they initiate and terminate, as in the situation calculus.

THE EVENT CALCULUS

Here we introduce the event calculus in the database-update case where events are used to derive initiated relationships only.

As in the situation calculus, the event calculus uses general rules to derive that a new relationship holds as the result of an event. However, as in historical databases, the event calculus associates time periods rather than global states with relationships. Thus, for example, the states of our situation-calculus example can be given the pictorial form in Figure 7. Here the small circles represent event tokens which give rise to time periods rather than global state transitions. The associated computational overheads will be discussed later.

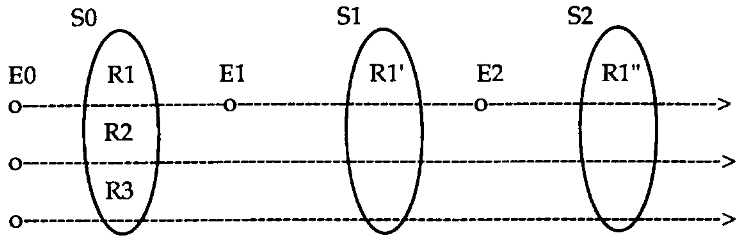


FIGURE 7.

Whereas in historical databases time periods are identified by giving their start and end times, in the event calculus they are *named* by terms of the form

after(*e r*),

where the first argument is the name of the event which starts the time period and the second argument in the name of the relationship itself. This second parameter is needed to distinguish different time periods that might be started by the same event, and that might be ended therefore by different events. For example, an event E of John's exchanging his possession of Book1 for Mary's possession of Apple1 initiates different periods:

after(E possess(Mary Book1))

after(E possess(John Apple1))

of possession by Mary and John respectively.

The use of a single term after(*e r*) in the event calculus to name time periods has similar functionality to the use of start and end times in historical databases. The event *e* which starts the period can be extracted directly from the name of the period. The event which ends the period can be derived by means of a general rule (similar to the persistence rule presented below) which expresses that an event *e** ends after(*e r*) if *e** terminates *r* and no event between *e* and *e** terminates *r*. The advantage of the single term over explicit start and end times is that unknown ends can be catered for without the need to introduce a fictitious infinite end time, ∞.

The event calculus uses a general, one-argument predicate to express that a relationship *r* holds for a period after(*e r*):

Holds(after(*e r*)).

Similar to the restriction that the first argument of "Holds" in the situation calculus names an atomic formula, the argument "*r*" in after(*e r*) is also restricted to the name of an atomic formula.

Using this notation, the states of our situation calculus example can be described by the assertions

Holds(after(E0 possess(Bob Book1))
Holds(after(E0 possess(John Book1))
Holds(after(E0 possess(Mary Book 3))
Holds(after(E1 possess(John Book1))
Holds(after(E2 possess(Mary Book1)).

As with the corresponding sentences of the situation-calculus formulation, the first three sentences describing the initial state would be given explicitly. The constant symbol E0 is used for convenience as the arbitrary name of an initializing event. The two remaining sentences can be derived from descriptions of E1 and E2 by means of general rules. (Similarly, the first three sentences could be derived from a description of E0.)

In general, the fact that a relationship holds for a period of time can be derived from an event description by means of a general rule (the *initiation rule*)

```
Holds(after(e r))   if    Happens(e)
                    and   Initiates(e r)
```

together with rules describing the relationships initiated by events. In this example,

```
Initiates(e possess(x y))   if    Act(e Give)
                            and   Recipient(e x)
                            and   Object(e y).
```

Thus, for example, the assertions

```
Holds(after(E1 possess(John Book1)),
Holds(after(E2 possess(Mary Book1))
```

can be derived from the event descriptions

```
Happens(E1)
Donor(E1 Bob)
Act(E1 Give)
Object(E1 Book1)
Recipient(E1 John),

Happens(E2)
Donor(E2 John)
Act(E2 Give)
Object(E2 Book1)
Recipient(E2 Mary).
```

That E1 occurred earlier than E2 can be represented by adding an assertion

```
Earlier-than(E1 E2)
```

or by associating explicit times with the events, e.g.

```
Time(E1 (1 Jan 77))
Time(E2 (25 Dec 89)).
```

Here we have employed a binary-predicate representation of events. This facilitates the treatment of incomplete event descriptions.

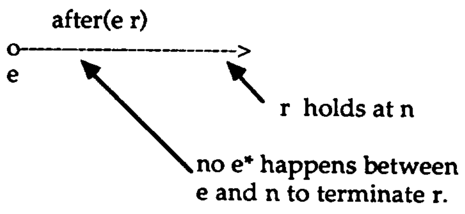
In the general form of the event calculus, the descriptions of E1 and E2 can be added to the database in any order. In particular, partial information about E1 can be intermingled with partial information about E2. Thus the event calculus incorporates the characteristics of semantic network representations of case semantics within a logic-programming framework. In addition it incorporates general rules connecting events with the relationships they initiate (and terminate) as in the situation calculus. The "Happens" predicate can be combined with other predicates over events, e.g.

"Possible", "Planned", "Permitted", "Obligated".

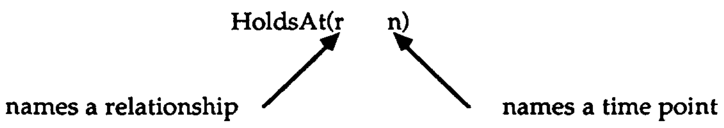
Thus modality can be incorporated without the use of modal logic. Moreover, as pointed out by Shanahan [17], general rules can be used to express that certain events are caused (implied) by other events.

THE FRAME PROBLEM

The computational aspect of the frame problem arises in the situation calculus when the frame axiom is used to reason that a given relationship holds in a given state because it held in the previous state and was not terminated by the state transition. In the event calculus we seek to avoid the computational inefficiencies of the frame problem by reasoning instead that a relationship holds at a given time point (which could be represented, for example, by a state, event, or date) if it holds for a period including that time point. This can be given a pictorial representation:



This reasoning can be formalized by employing a predicate



The "HoldsAt" predicate in the event calculus is meant to serve the same function as the "Holds" predicate in the situation calculus. We call the rule which defines it the *persistence axiom*:

```
HoldsAt(r n)  If Holds(after(e r))
              and e < n
              and not ∃e* [Happens(e*) and
                           Terminates(e* r) and
                           e < e* and
                           e* ≤ n].
```

Notice that the inequalities in this axiom have the effect that relationships hold for open periods which do not include their start and end times. Other conventions could also be employed.

We also need rules describing the relationships terminated by events. In this case

Terminates(*e* possess(*x y*)) if Act(*e* Give)
and Donor(*e x*)
and Object(*e y*).

Notice that, as in the situation calculus, we can also have general rules which express ramifications. For example:

HoldsAt(wants(John *x*) *n*)if not HoldsAt(possess(John *x*) *n*).

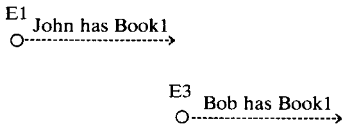
The ramification problem does not arise, because the persistence axiom, together with the initiation rule, only applies to explicitly initiated (i.e. primitive) relationships.

The persistence axiom states that once a relationship has been initiated, it persists indefinitely into the future until it is terminated. In practice we might want to add an extra condition to the axiom, limiting persistence to a period of time which is reasonable for the relationship concerned.

More importantly, the persistence axiom needs to be revised to deal correctly with incomplete information. For example, given only descriptions of the events

E1:	Bob gives Book1 to John
E3:	Mary gives Book1 to Bob
	E1 < E3,

the persistence axiom concludes that John and Bob have Book1 at the same time:

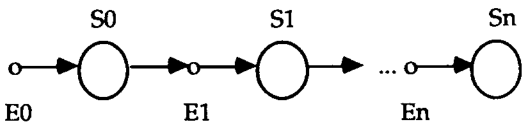


This is because E3 only terminates a previous state of possession of Book1 by Mary.

We shall discuss later how to modify the persistence axiom to deal correctly with incomplete information.

EXECUTION OF THE PERSISTENCE AXIOM

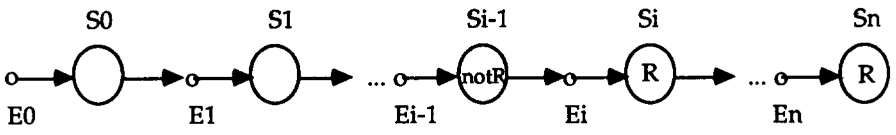
Here, to facilitate comparison with the situation calculus, we assume that events are totally ordered and therefore give rise to a sequence of global states:



The existence of this sequence implies that numerical time points can be associated with events. For example, we can arbitrarily assume that E*i* occurs at time *i*.

We shall also assume that, as in the situation-calculus case, we have complete, relevant information about the past. The present form of the persistence axiom deals correctly with this case.

We claim that under these assumption, the event calculus solves the epistemological and computational aspects of the frame problem. As in the situation calculus, the use of general "Holds" and "HoldsAt" predicates, together with negation as failure, solves the epistemological aspect of the frame problem. The solution of the computational aspect depends upon an optimization of the execution of the persistence axiom. In the case of the situation calculus, the problem is the excessive computation required to determine whether a relation R holds in a state Sn (equivalently, at a time point n). Pictorially:



The frame axiom requires us to find an event Ei earlier than Sn, which initiates R, such that no event between Ei and En terminates R. This requires $n-i$ applications of the frame axiom together with subsidiary inference steps to establish the conditions of the frame axiom. The inefficiency of these inferences, compared with the one inference required when destructive assignment is used to perform updates in relational databases, is the computational aspect of the frame problem, which the persistence axiom seeks to avoid.

To determine, under the same circumstances in the event calculus, that R holds at time n, requires, in contrast, only *one application* of the persistence axiom. This, however, subdivides into two subparts. The first subpart (consisting of the first two conditions) *finds an event* Ei *at a time before* n *which initiates* R. The second subpart (consisting of the four negated conditions) *shows that no event occurring after* Ei, *and before or at time* n, *terminates* R. Establishing these two subparts has a hidden computational overhead, which we now discuss.

The minimum overhead involved in applying the persistence axiom is the number of inference steps contained in a successful proof. In the case of the first subpart, therefore, we can ignore (for the time being) the work needed to find the right event Ei, and count only the number of steps in a proof. This is six steps for the first condition, plus one for the other, a total of seven. This number seven of steps in a proof is independent of the problem in general and of the size of n and i in particular. Of course, the size of n might well affect the complexity of finding the right solution. We shall come back to this point after we discuss the complexity of the second subpart.

The minimum overhead involved in the second subpart of the persistence axiom, because it involves negation as failure, is all the work of showing that the four conditions inside the negation have no solution. This requires an exhaustive search of the entire search space. However, different ways of solving the four conditions give rise to search spaces of dramatically different size. The possibility of solving the frame problem depends largely on the size of these search spaces, and therefore on the way the four conditions are solved.

At one extreme, executing the conditions in the order in which they have been written, with

Happens(e^*)

first, it would be necessary to investigate all recorded events

E0, E1, ..., En,

showing that none of them satisfies the remaining three conditions. This is computationally even worse than executing the frame axiom, which explores only the events between Ei and En.

It is possible to reduce the size of the search space significantly by executing conditions in a different order. For example, executing the condition

Terminates(e^* r)

first, followed by

Happens(e^*),

limits attention to those recorded events which are of the right type to terminate r. There is still a search required to determine whether any of these occur between Ei and En. In practice this might be almost efficient enough to be regarded as an acceptable solution of the frame problem. In fact, we can find a better solution by executing several conditions simultaneously.

The potential for simultaneous execution of conditions can be illustrated by considering the two conditions

$e < e^*$ and $e^* \leq n$

executed one at a time with e and n given, but e^* variable. In PROLOG-style execution, followed by the two remaining conditions, it is necessary to explore $e < e^*$ infinitely many times. This is clearly impossible. The situation is not much improved if the order of execution of the two conditions is reversed. The situation is dramatically improved, however, if the two conditions are transformed into one, say

Between(e e^* n),

so that given e and n, only times between e and n are generated as values for e^*. (Here, for the sake of simplicity, we assume that e, e^*, and n range over positive integers.) This transformation is easily accomplished and is an established logic-programming technique. In this case, if the new condition is executed before the two other remaining, we simulate execution of the frame axiom by exploring all events between Ei and En.

Such program transformation can be combined with more sophisticated "run-time" execution strategies. Among the simplest of these is the use of indexing for the associative retrieval of clauses. Our proposed solution of the frame problem is based upon such associative retrieval: We propose that event descriptions, when they are input, be executed one step forward to derive the conclusions of the form

Terminates(*e r*)

which they imply, and that these conclusions be stored and indexed on their second argument *r*. Moreover, the cluster of events all terminating the same relationship should be ordered according to the time of occurrence. Given such an indexed, ordered storage of "Terminates" relationships, to satisfy the second subpart of the persistence axiom, it suffices to

(1) access the cluster of events terminating the relationship *R*, and

(2) search of an event, stored in the cluster, which has occurred between *i* and *n*.

Both access to the cluster and search for one event occurring at an appropriate time can be accomplished efficiently using conventional database storage and accessing techniques.

Moreover, in the most frequently occurring case, where *n* is the "current time", i.e. after the latest event occurrence, it suffices to look at the latest-occurring event within the cluster of events terminating *R*. This can be done without any search, once the cluster has been found.

One further refinement of the execution strategy is necessary. To avoid the potential inefficiency involved in searching for the right event E*i* to solve the first subpart of the persistence axiom, we should similarly derive "Initiates" relationships by reasoning one step forward from event descriptions, to derive conclusions of the form Initiates(*e r*). These conclusions can then be stored, indexing them on their second argument *r*, and ordering them according to the time of occurrence of events.

The persistence axiom as a whole can then be implemented in the following way: To determine whether *R* holds at time *n*,

(1) (a) access the cluster of events initiating *R*;
    (b) find the latest one of them, *e*, occurring before *n*;
    (c) if there is none, then *R* does not hold at time *n*; otherwise

(2) (a) access the cluster of events terminating *R*;
    (b) find the latest one, e^*, occurring before or at *n*;
    (c) if $e < e^*$, then *R* does not hold at time *n*;
    otherwise, if $e^* < e$ or if there is no such e^*, then
    R holds at time *n*.

Accessing the cluster initiating *R* in (1)(a) and the cluster terminating *R* in (2)(a) can be performed using hashing and other indexing techniques, with the same efficiency as accessing *R* in a conventional relational database. Finding the latest time *e* before *n* in (1)(b) and e^* before or at *n* in (2)(b) can be accomplished in $\log k$ time, where *k* is the size of the given cluster. Moreover, in the most frequently occurring case, where *n* is the current time, finding *e* and e^* can be accomplished without any search, once the clusters have been found.

Furthermore, if there were never any reason to access historical (noncurrent) data, it would suffice to store only the latest initiating event for each relationship. Thus any nonempty cluster would "degenerately" consist of only a single event initiating a relationship which currently holds. This gives us an implementation very similar to that obtained with destructive assignment in relational databases.

One final inefficiency, however, remains in our implementation. Every relationship has to be stored redundantly for every event which initiates or terminates it. This overhead can be minimized by merging the two clusters of events, initiating and terminating a given relationship, into a single cluster ordered by the time of occurrence of events. The two parts of the implementation (1) and (2) of the persistence axiom can then be combined into one.

The various implementations of the persistence axiom which we have discussed do not exhaust the possibilities. The important point is that some of these rival the efficiency of destructive assignment in relational databases, and others incur a computational overhead which seems acceptable in comparison with that encountered in the situation calculus.

THE ELIMINATION OF TIME PERIODS

As a further optimization in the special case we have been considering, where events only initiate the persistence of relationships forward into time, time periods can be eliminated altogether. This can be done simply by using the initiation axiom to eliminate the Holds(after(*e r*)) condition of the persistence axiom, obtaining the rule

```
HoldsAt(r n)  if Happens(e)
              and Initiates(e r)
              and e < n
              and not ∃e* [Happens(e*) and
                          Terminates(e* r) and
                          e < e* and
                          e* < n]
```

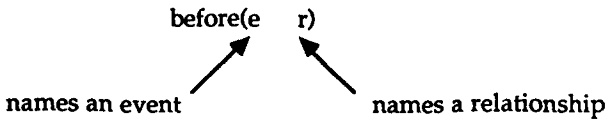
Time periods play a more important role in the general case where events also terminate relationships which persist into the past.

THE GENERAL CASE

In the general case an event description can give as much information about the past as it does about the future. Thus, for example,

"Bob gave Book1 to John"

implies both that Bob had the book before and that John had the book after the event. Both periods of time can be named as a function of the event and of the relationship. We use the notation

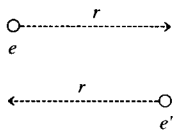


to name the period for which the relationship *r* terminated by *e* holds. We can also formulate axioms for persistence backward into time:

Holds(before(*e r*)) if Happens(*e*)
and Terminates(*e r*)

HoldsAt(*r n*) if Holds(before(*e r*))
and $n < e$
and not $\exists e^*$ [Happens(e^*) and
Initiates(e^* *r*) and
$n \leq e^*$ and
$e^* < e$].

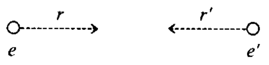
In cases such as



where no e^* terminating or initiating *r* occurs between *e* and *e'*, the two persistence axioms give us two ways of concluding *r* holds between *e* and *e'*. This redundancy can be avoided by approximately combining the two persistence axioms into one.

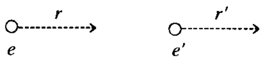
Cases of incomplete information that call for assimilating information about the past, such as those pictured in (a), (b), and (c) below, where information about events occurring between *e* and *e'* is missing, necessitate more dramatic revision of the rules:

(a)



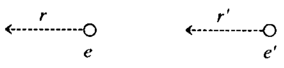
where *r* and *r'* are incompatible, e.g. $r = \text{possess}(xy)$, $r' = \text{possess}(x'y)$, and $x' \neq x$.

(b)



where *r* and *r'* are incompatible or identical.

(c)

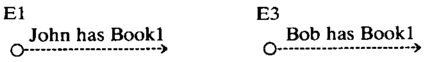


where *r* and *r'* are incompatible or identical.

The persistence rules as currently formulated allow us to conclude in such cases that incompatible relationships hold at the same time.

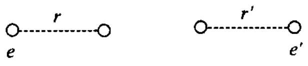
Such undesirable consequences can be avoided in several ways. The simplest solution in a case such as (b), for example, is to use integrity constraints to insist that, whenever an event description is added to the database, then all of the relationships terminated by the event are recorded in the database as holding just before the event. This solution is, however, very restrictive. For example, it would rule out the event description of E3 in the situation where

E1:	Bob gives Book1 to John
E3:	Mary gives Book1 to Bob
	E1 < E3.

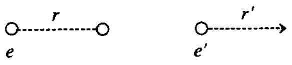


A much more powerful solution was presented in the original event-calculus paper [9], where general rules imply the existence of unreported events. Pictorially:

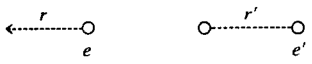
(a)



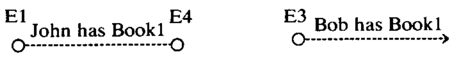
(b)



(c)



In the example of the events E1 and E3 above, this solution implies that there must be an event E4, between E1 and E3, which terminates John's possession of Book1:



In the database-update case, a third solution, which incorporates forward but not backward persistence, is also possible. In this approach, the condition

Terminates(e^* r)

in the persistence rule is replaced by the condition

Breaks(e^* r)

where Breaks is defined by

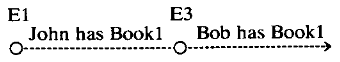
Breaks(e^* r) if Terminates(e^* r),

Breaks(e^* r) if Terminates(e^* r') and Incompatible(r r'),

Breaks(e^* r) if Initiates(e^* r),

Breaks(e^* r) if Initiates(e^* r') and Incompatible(r r').

In the example of the events E1 and E3 above, this approach implies that John's possession of Book1 persists until E3:



This loses the information that Mary possessed Book1 immediately before E3, but avoids concluding that incompatible relationships hold at the same time.

This problem with reasoning about persistence in the context of incomplete information also arises in the context of historical databases, where it is more difficult to deal with because there is no underlying semantics of events.

CONCLUSION

The event calculus seeks to combine the expressive power of case semantics with the deductive power of logic and the computational power of logic programming. It follows the situation calculus in its use of general rules to impose semantic structure on updates, while at the same time attempting to avoid the frame problem. In the course of doing so we have hoped to achieve, without explicitly incorporating deletion in updates, the efficiency obtainable with destructive assignment in relational-database updates. Thus we have aimed to obtain the advantages of the other approaches without incurring their disadvantages.

I am grateful to Marek Sergot and Murray Shanahan for discussions about the event calculus, to Fariba Sadri and Sury Sripada for helpful comments on earlier drafts of this paper, and to Lode Missiaen for pointing out the significance of the ramification problem. This research was supported by the Science and Engineering Research Council in the context of the Alvey Programme.

REFERENCES

1. Allen, J. F., Maintaining Knowledge about Temporal Intervals, TR-86, Computer Science Dept., Univ. of Rochester, Jan. 1981, *Comm. ACM* 26:832–843 (1983).
2. Allen, J. F., Towards a General Theory of Action and Time," *Artif. Intell.* 23:123–154 (1984).
3. Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum, New York, 1978, pp. 293–322.
4. Clifford, J. and Warren, D. S., Formal Semantics for Time in Databases, *ACM Trans. Database Systems* 8(2):214–254 (1983).
5. Eshghi, K. and Kowalski, R. A., Abduction Compared with Negation by Failure, in: *Fifth International Conference on Logic Programming*, MIT Press, 1989.
6. Hanks, S. and McDermott, D., Nonmonotonic Logic and Temporal Projection, *Artif. Intell.* 33(3):379–412 (1987).
7. Jones, S., Mason, P. J., and Stamper, R. K., Legol-2.0: A Relational Specification Language for Complex Rules, *Inform. Systems* 4, No. 4 (1979).
8. Kowalski, R. A., *Logic for Problem Solving*, North Holland, New York, 1979.
9. Kowalski, R. A. and Sergot, M. J., A Logic-Based Calculus of Events, *New Generation Comput.* 4:67–95 (1986).
10. Kowalski, R. A. and Sadri, F., Knowledge Representation without Integrity Constraints, Dept. of Computing, Imperial College, 1988.
11. Lee-Coelho-Cotta, Temporal Inferencing on Administrative Databases, *Inform. Systems* 10(2):197–206 (1985).
12. Lloyd, J. W. and Topor, R. W., Making Prolog More Expressive, *J. Logic Programming* 1(3):225–240 (1984).

13. McCarthy, J., Situation, Actions, and Causal Laws, Memo 2, Stanford Artificial Intelligence Project, 1963.
14. McCarthy, J. and Hayes, P. J., Some Philosophical Problems from the Standpoint of Artificial Intelligence, in: B. Meltzer and D. Michie, (eds.), *Machine Intelligence*, 4, Edinburgh U.P., Edinburgh, 1969, pp. 463–502.
15. Sadri, F., Three Recent Approaches to Temporal Reasoning, in: A. Galton (ed.), *Temporal Logics and Their Applications*, Academic, 1987, pp. 121–168.
16. Sernadas, A., Temporal Aspects of Logical Procedure Definition, *Inform. Systems* 5(3):167–187 (1980).
17. Shanahan, M., Representing Continuous Change in the Event Calculus, Dept. of Computing, Imperial College, London, 1989.
18. Snodgrass, R., The Temporal Query Language TQuel, *ACM Trans. Database Systems* 12(2):247–298.
19. Stamper, R. K., The Legol Project and Language, in: *Proceedings of the Datafair Conference*, British Computer Soc., *London*, 1973.
20. Stonebraker, M., The Design of the Postgress Storage System, in: *Proceedings of VLDB*, Morgan Kaufmann, 1987, pp. 289–300.
21. Sripada, S. M., A Logical Framework for Temporal Deductive Databases, in: *Proceedings of VLDB*, Morgan Kaufmann, 1988, pp. 171–182.
22. Takeuchi, A. and Furukawa, K., Partial Evaluation of Prolog Programs and Its Application to Metaprogramming, in: *Proceedings of IFIP 86*, North Holland, 1986, pp. 415–420.
23. Evans, C., Negation-as-Failure as an Approach to the Hanks and McDermott Problem, Dept. of Computing, Imperial College, 1988.