Specification and Implementation of Temporal Databases in a Bitemporal Event Calculus *

Carlos A. Mareco and Leopoldo Bertossi

Departamento de Ciencia de la Computación
Escuela de Ingeniería
Pontificia Universidad Católica de Chile**

Abstract. In this paper we show how temporal databases can be specified and implemented using the bitemporal event calculus, an extension of the event calculus that includes both valid and transaction time, and the possibility to perform temporal updates. A caching mechanism that maintains the current historical state and is updated after each transaction has also been incorporated. We also consider the problem of checking integrity constraints in this kind of temporal databases. The methodology for consistency checking presented here is an extension of other approaches found in the literature that exploit the assumption that the database satisfies its integrity constraints prior to the update transaction. A prototype of the formalism and the checking mechanism, implemented in Prolog, has also been developed.

1 Introduction

Time often shows up in information systems, as these are, after all, models of some part of our reality. However, despite a great deal of work in the area of temporal databases[1], most database management system implementations do not have built-in time support. The reason for this is the complexity of the semantics and representation of time. Having a logical specification of a temporal database and being able to study its evolution through a sequence of transactions would be useful to shed some light on the topic.

In this paper we briefly show how an extended version of the event calculus [8], a logic programming formalism for specifying event occurrences and change, can be used to both specify and implement temporal databases. An extended account can be found in [10]. This new formalism will allow us to have a logical specification of a temporal database, with a logic programming semantics and the possibility of computing directly from the specification. Our specification has, among others, the following features:

- Handle *two* (orthogonal) *time lines*: valid and transaction time.

* This research was supported by FONDECYT (Grant #1980945) and ECOS/CONICYT (Grant C97E05.)

** Casilla 360, Santiago 22, Chile. {camareco, bertossi}@ing.puc.cl

[1] We use the terminology of Snodgrass and Ahn [16] regarding time and databases.

– Support *temporal updates* similar to those found in conventional temporal databases.
– Allow queries involving both time lines.
– Allow *corrections* to be made to the information contained in the Database.
– Efficient query answering and integrity constraint checking.

The rest of this paper is organized as follows. First, we briefly introduce the event calculus. Sect. 2 summarizes some aspects of the bitemporal event calculus, BtEC, presented in [10], in particular, it shows how to add transaction time and temporal updates to the simplified event calculus. Section 3 explains how a caching mechanism can improve the efficiency of the underlying computational mechanisms. Integrity constraint checking in the bitemporal event calculus is analyzed in Sect. 4. Finally, in Sect. 5 we discuss some related work.

1.1 The Event Calculus

The event calculus was introduced by Kowalski and Sergot in [8] as a formalism for reasoning about events and change in a logic programming framework. However, after the original paper was published, most of the work on the event calculus centered on a simplified version [6, 13], called the simplified event calculus.

In the simplified event calculus, events initiate and terminate properties of the real world. Updates are of additive nature only, and they are specified by adding facts of the form: $happens\_at(Event, Time)$, where $Time$ is the valid (or real world) time of occurrence of the event. Events are related to the properties they initiate/terminate by predicates of the form:

$$initiates\_at(Event, Property, Time), \quad terminates\_at(Event, Property, Time).$$

The simplified event calculus is composed of the following general rules:

$$\begin{aligned} holds\_at(p, v) &\leftarrow happens\_at(e, v_1),\ v_1 < v,\ initiates\_at(e, p, v_1), \\ &\qquad \text{not } broken(p, [v_1, v]). \\ broken(p, [v_1, v_2]) &\leftarrow happens\_at(e, v),\ v_1 < v < v_2,\ terminates\_at(e, p, v). \end{aligned}$$

The $holds\_at$ rule is used to ask about the validity of some property at a particular time point. It states that a property p is valid at time v if there exists an event e which occurs at a time v_1 less than v, e initiates p and the interval $[v_1, v]$ for p is not broken. An interval for a property is broken if there exists an event which terminates the property and occurs within the interval. Negation-as-failure is used to express that, by default, the effects of an event which initiates a property extend until it is interrupted by another event which terminates the property. Since the simplified event calculus allows us to deduce the (valid) time periods during which properties hold, it can be used, as pointed out in [18], to formalize a historical database, albeit without a facility for correcting errors.

2 The Bitemporal Event Calculus

As each event is added, the database goes through different states (a state is a set of properties with their valid time intervals), each state being a historical database in itself. However, unlike a temporal database, when an event is added, it destroys the previous state of the database, making it impossible to derive the valid time intervals which held in previous states.

In order to access a certain state s, only events which were introduced up to that state s must be considered by the rules. A *transaction time* can be associated to every event, and events are specified as facts: $happens\_at(event, valid\ time, transaction\ time)$.

In the simplified event calculus, events specify only one time point, namely, the time at which the event occurs in the modeled reality. Therefore, an event may only determine the start or end of a property's valid time interval.

However, several temporal data models and databases, such as TQuel [14], ATSQL2 [20], BCDM [5] and Chronolog [1], provide updates that specify valid time intervals rather than time points. This is so because it is quite common to refer to time intervals, for example, to say that an employee's salary will be x from September to December. In addition, it is still possible to simulate the behavior of the simplified event calculus using intervals, simply by specifying that the interval has an infinite upper bound.

Conversely, it is also possible to specify intervals with time points by using two events per interval [18]. This, however, quickly becomes complex for simple and natural cases. For these reasons, we chose to use valid time intervals instead of time points directly in the events, which are now specified as: $happens\_at(event, [v_1, v_2], tx)$, where $[v_1, v_2]$ is the valid time interval, and tx is the time when this occurrence is introduced in the database. The semantics of this transaction is: the event occurs at valid time point v_1 and its effects extend up to v_2. It is *not* an event with duration; events are still considered to be instantaneous, the only difference is that now their effects do not extend until infinite as before, only up to v_2.

The first modification required is to have the rules filter out events according to the transaction time specified in the query. Other changes are necessary to account for the time intervals in the events. The basic rules of our formalism are the following:

$$holds\_at(p, vt, tx) \leftarrow happens\_at(e, [v_s, v_e], t_{in}),\ t_{in} \leq tx,\ v_s < vt \leq v_e,\ initiates\_at(e, p, v_s, tx), \text{not } broken(p, vt, [t_{in}, tx]). \quad \text{(BtEC1)}$$

$$broken(p, v, [t_s, t_e]) \leftarrow happens\_at(e, [v_s, v_e], t_{in}),\ t_s < t_{in} \leq t_e,\ v_s < v \leq v_e,\ terminates\_at(e, p, v_s, t_e). \quad \text{(BtEC2)}$$

The *holds_at* rule states that a property p holds in a valid time point vt and transaction time tx if there exists an event e which initiates the property and is introduced in the database in a transaction time $t_{in} \leq tx$ with a valid time interval $[v_s, v_e]$ that contains vt, and the property is not terminated at that valid time point from the moment (t_{in}) when the event was introduced in the database until the time (tx) specified in the query. Predicate *holds_at* is like the "timeslice" operator offered by some temporal data models.

The *broken* rule states that a property p is interrupted at a valid time point v if there is an event which occurs in a transaction time between t_s and t_e, has a valid time interval containing v and terminates the property. This means that, unlike the simplified event calculus, where the order (transaction time) of the events is not considered, if a property is initiated by an event at a transaction time t, events that occurred at transaction times less than t cannot affect the property.

There is also a rule that defines the predicate *mholds_for*(*Prop*, *ValidInterval*, *Trans.Time*), which derives the valid time intervals of the properties for a given transaction time directly from the event occurrences.[2]

On the basis of the formalism presented so far it is possible to support four kinds of update on a temporal database [14, 1]: insertion, deletion, replacement and modification. Notice, however, that since a single event can initiate or terminate several properties, one event can produce effects that are a combination of one or more of these types of update.

Insertion and Deletion: are normally achieved by means of events that initiate (terminate) a property as in Sect. 1.1. Another way of accomplishing this type of update, that is similar to conventional databases, is by defining two domain independent events, **ins** and **del**, that simply initiate and terminate respectively, the specified property. They can be specified by $\mathit{initiates\_at}(\mathbf{ins}(p), p, v, t)$ and $\mathit{terminates\_at}(\mathbf{del}(p), p, v, t)$; meaning that any event $\mathbf{ins}(p)$ ($\mathbf{del}(p)$) occurring at valid time v and transaction time t initiates (terminates) property p. They can be used for error correction. We are neither deleting nor revising the previous events, but only overriding their effects on the property.[3]

Replacement: In a replacement, properties in given intervals are replaced by other properties in possibly different intervals. A way of specifying a replacement, e.g. an error correction, is by means of an event, **replace**, of the form: $\mathit{happens\_at}(\mathbf{replace}(p_1, [vs_1, ve_1], p_2), [vs_2, ve_2], tx)$, meaning that p_1, in the interval $[vs_1, ve_1]$ will be replaced by p_2, in the interval $[vs_2, ve_2]$. This event can be expressed in terms of **ins** and **del** events and vice versa.

Modification: similar to the `UPDATE` sentence in SQL. It means updating the value of a property to another at a given valid time interval. It could be seen as a particular case of a replacement, where the valid time intervals coincide. Most modifications are realized by domain-dependent events, but it is also possible to represent the operation explicitly, using a new event, **update**, specified by $\mathit{happens\_at}(\mathbf{update}(p_1, p_2), [v_s, v_e], tx)$.

[2] Intervals determined by this and the previous rules are open to the left and closed to the right. The time point to the right may be infinity. Coalescing of intervals is not done automatically by these rules, but there is another one in the BtEC for this purpose.

[3] These events are ordinary events, at the same object level as the domain dependent ones, and are properly handled by the general rules of the BtEC presented before, in particular, they share the same logic programming semantics of the others.

Table 1. Transactions entered in the database

Event	Valid Int.	T.Time
hire('therese', 5873, 'bahnhofstrasse', 'zurich')	[02.01, *inf*]	1.1
ins($salary(5873, 3630)$)	[02.05, *inf*]	2.1
hire('franziska', 6542, 'rennweg', 'zurich')	[02.01, *inf*]	3.1
ins($salary(6542, 3200)$)	[02.01, *inf*]	4.1
hire('lilian', 3463, 'speedway', 'tucson')	[02.02, *inf*]	5.1
ins($salary(3463, 3400)$)	[02.02, *inf*]	6.1
$raise(6542, 3800)$	[06.01, *inf*]	7.1
replace($salary(3463, 3400), [02.02, inf], salary(3463, 3500)$)	[02.02, *inf*]	8.1
update(*employee*('franziska', 6542, 'rennweg', 'zurich'), *employee*('franziska', 6542, 'nieder', 'zurich'))	[08.01, *inf*]	9.1

Example 1: (Based on [17]) There are two tables: *employee(ename,eno,street,-city)* and *salary(eno,amount)* and two events, *hire* and *raise*, defined by:

$$initiates\_at(raise(eno, amt), salary(eno, amt), v, t).$$
$$terminates\_at(raise(eno, amt), salary(eno, amt'), v, t) \leftarrow amt \neq amt'.$$
$$initiates\_at(hire(eno, name, st, cty), employee(eno, name, st, cty), v, t).$$

Also, it is assumed that the transactions[4] in Table 1 have been entered in the database. Next, we show some queries on this database. These are expressed as Prolog rules.

1. *List the salary history of the employees.*

```
q1(E,S,I) :- mholds_for(employee(E,N,_,_), I1, 10.1),
        mholds_for(salary(N,S),I2,10.1), intersection([I1,I2],I).
```

   In this query the employee and her salary are extracted, along with their respective valid time intervals using the *mholds_for* rule. Then the intersection of the valid time intervals is computed. This is equivalent to performing a temporal join. The results of this query are:

```
franziska 3800 [8.01,inf]  | franziska 3800 [6.01,8.01]
therese   3630 [2.05,inf]  | lilian    3500 [2.02,inf]
franziska 3200 [2.01,6.01] |
```

2. *List the employees for which no one makes a higher salary in a different city.* This query can be expressed as:

[4] The times shown (e.g. 02.01) represent dates in (month.day) format.

```
q2(E,I) :- mholds_for(employee(E,N,_,C), I1, 10.1),
        mholds_for(salary(N,S),I2,10.1), intersection([I2,I1],In),
        (auxq2([N,S,C],I3) -> diff(In,I3,I); I = In).

auxq2([N,S,C],I) :- mholds_for(salary(N1,S1),Iv2,10.1),
        N \== N1, mholds_for(employee(_,N1,_,C1),Iv1,10.1),
        intersection([Iv1,Iv2],I), S1 > S, C1 \== C.
```

The results are:

```
franziska   [2.01,2.02] | lilian      [2.02,2.05]
franziska   [6.01,8.01] | therese     [2.05,inf]
franziska   [8.01,inf]
```

3 A Caching Mechanism for the BtEC

The event calculus performs a lot of redundant computation when answering queries, since every time a query about a property is posed, the whole computation is repeated from the beginning, even though the property might have not changed [2]. One way to improve its efficiency is to add a caching mechanism, as in the cached event calculus (CEC) [2], to store the valid time intervals of the properties and update them efficiently whenever an event is added to the database (the undesirable alternative would be to compute everything from scratch after every transaction.) Thus, when a query is formulated, the intervals are already available in memory and they do not have to be computed by the rules.

We extended and adapted the CEC to the case of interval-based updates and two time lines, i.e. the formalism of Sect. 2 [10]. The idea is to keep in memory the set of valid time intervals corresponding to the current (relative to transaction time) historical state, and update it with every transaction. That is, the cache keeps an historical database at the current transaction time. The cache is computed by means of rules in the BtEC specification. It consists of a collection of facts of the form *mholds_for*(*Prop*, *ValidInterval*), without transaction time.

Since most of the queries posed to a temporal database refer to the current transaction time, these can be answered just by looking up in memory the relevant intervals. The rules given in Sect. 2 are executed only when a query refers to some past state. There is a price to be paid, of course, the time required to perform the update does increase, since the cache needs to be modified to reflect the effect of the event that was introduced, but the performance gained in answering queries makes the combined cost lower than in the non-cached version [2].

Integrity constraint checking is also improved by using the cache, as will be seen in Sect. 4.

4 Integrity Constraint Checking

We consider only temporal integrity constraints which refer to the current historical state, that is, to properties with arbitrary valid time intervals as time stamps, but only with a fixed transaction time (the current one).

A way to check the consistency of a database is simply to transform the integrity constraints into their denial forms, as queries about their violating tuples, and pose them after each transaction. This is extremely expensive in the simplified event calculus, where the valid time intervals must be derived each time they are required. Then, an important improvement can be achieved by using the caching mechanism described in Sect. 3. Since most integrity constraints refer only to the current historical state (which is kept in the cache), the verification process takes place entirely in the cache, what is much more efficient[5].

Further improvements on integrity constraint checking can be made by exploiting the assumption that the database satisfies its integrity constraints prior to a transaction. We present here such a methodology. It can be seen as a specialization of the one proposed in [7] for checking integrity constraints in a deductive database. In [7], an extension of the SLDNF resolution based proof procedure with negation as failure [9] is used to reason forwards from the update, focusing on the relevant parts of the database and constraints. It achieves the same effect of the simplification algorithms of Nicolas [11].

The idea behind our methodology is to take advantage of the fact that, when the caching mechanism updates the intervals, it gathers information regarding which properties were modified as a direct or indirect[6] result of the transaction. After the cache has been updated, only those integrity constraints which contain properties that were modified are considered for verification. Out of these, only some, depending on the modified property and the integrity constraint, are effectively considered for checking. The modified properties are used to instantiate variables in the integrity constraints, which reduces the search space. Also, in many cases the integrity constraints can be simplified, so only "residues" of them are actually checked.

As explained before, we consider only temporal integrity constraints which refer to the current transaction time (the time of the cache) and we assume that they are written as denials. Thus, they have the form:

$$\begin{aligned}
&\leftarrow \; \mathit{mholds_for}(p_1(\bar{x}_1), I_1), \ldots, \mathit{mholds_for}(p_n(\bar{x}_n), I_n), \\
&\qquad f(\bar{x}_1, I_1, \ldots, \bar{x_n}, I_n), [\text{ not } \mathit{aux}(\bar{x}_1, I_1, \ldots \bar{x_n}, I_n).] \\
&\mathit{aux}(\bar{x}_1, I_1, \ldots \bar{x_n}, I_n) \leftarrow \mathit{mholds_for}(q_1(\bar{y}_1), I'_1), \ldots \mathit{mholds_for}(q_m(\bar{y}_m), I'_m), \\
&\qquad\qquad \phi(\bar{x}_1, I_1, \ldots, \bar{x_n}, I_n, \bar{y}_1, I'_1, \ldots).
\end{aligned}$$

[5] Integrity constraints which refer to different transaction times in addition to valid time do not benefit from this approach.

[6] In case a precondition for a property being initiated is terminated by a new event, the property ceases to hold.

That is, some properties (p_i) appear in non-negated *mholds_for* clauses and some others (q_i) within a negation (*aux* rule).[7] Notice also that we omitted the transaction time from the *mholds_for* predicate here because that is the time of the cache by default.

Example 2: *Temporal Referential Integrity Constraint.*
We can express this integrity constraint (for a concrete case) as:

$$\begin{aligned} &\leftarrow mholds\_for(salary(e,s),I), \text{not } aux(e,I). \qquad \text{(TRI)}\\ aux(e,I) &\leftarrow mholds\_for(employee(e),I_1), contains(I_1,I)., \end{aligned}$$

which says that the integrity constraint is violated if there is a tuple (e, s) with a valid time interval I in the *salary* table, such that e does not exist in the *employee* table with an interval that contains I. In this case it is necessary to use the auxiliary rule for *aux* to avoid floundering; otherwise we would have had the free variable, I_1, inside a negation. □

Before describing our procedure for integrity constraint checking, let us notice that no matter how complex the events are (either domain dependent or updates), at the end, wrt the current transaction time, everything reduces to the introduction into the cache of certain properties valid at certain intervals, or the removal of them. The procedure, summarized in Fig. 1, is based on this observation.

Notice that an integrity constraint is checked only when a property which appears non-negated (respectively, within a negation) in it is introduced (respectively, removed)[8]. Under the assumption that the database satisfies its integrity constraints prior to the transaction, that is, they are true in the cache we had before the execution of the new event or update, the introduction (removal) of a property which occurs negated (non-negated) in an integrity constraint cannot falsify it.

Nicolas, in [12], has shown that integrity constraints in prenex conjunctive normal form[9] which do not contain a relation R in a negated atomic formula (respectively, non-negated) are unaffected when a tuple is introduced (respectively removed) into (from) R. The cases in which Nicolas says it is not necessary to check the integrity constraint are exactly the cases we are not considering[10].

The way the integrity constraints are checked (unifying the occurrence of the modified property in the integrity constraint and, possibly, deleting that occurrence) resembles the refutation that takes place in [7]. However, since our

[7] The reason for introducing the negations in this form is to avoid floundering, that is, a call to negation as failure with uninstantiated variables [9].

[8] We use the words "introduced" and "removed" for the operations on the cache, instead of "inserted" and "deleted" that are reserved for operations on the database.

[9] That is, formulas that consist of a prefix of quantifiers followed by a quantifier free formula that is a conjunction of disjunctions.

[10] Remember that we are dealing with integrity constraints in denial form. Integrity constraints that are not written as denials can be transformed into this form by using the well-known Lloyd-Topor transformation rules [9].

For each modified property $r(\bar{a})$*:*
For each integrity constraint IC*:*
If ($r(\bar{a})$ *was* introduced *in the interval [c,d] and* r *appears non-negated in* IC*):*
- *Unify the clause* $mholds\_for(r(\bar{x}),[y,z])$ *in* IC *with* $mholds\_for(r(\bar{a}),[c,d])$*. Call the resulting (instantiated) rule* IC_a*.*
- *Delete* $mholds\_for(r(\bar{a}),[c,d])$ *from* IC_a*. Call it* IC'_a*.*
- *Check* IC'_a *against the updated cache.*

If ($r(\bar{a})$ *was* removed *and* r *appears within a negation in* IC*):*
- *Unify the clause* $mholds\_for(r(\bar{x}),[y,z])$ *in* IC *with* $mholds\_for(r(\bar{a}),[c,d])$*. Call the resulting (instantiated) rule* IC_a*.*
- *Check* IC_a *against the updated cache.*

Fig. 1. Integrity constraint checking procedure

database consists only of *mholds_for* facts (there are no intensional rules), we do not need to consider the cases in which deductive rules are inserted or deleted nor the implicit changes in the intensional knowledge which could occur as a result of a database update. Thus we are left with a simplified methodology well suited to checking a large class of integrity constraints, those that only refer to the current historical state, in the bitemporal event calculus.

Example 3: Consider the integrity constraint (TRI) of the previous example. Suppose the database is empty and the fact that John is hired in the interval [5,20] is then recorded. Since the fact *employee*(john) was inserted and *employee* appears within a negation, the integrity constraint cannot be violated by this transaction.

Now we issue a transaction that states that John's salary will be 40k during the interval [10,20]. In this case, since the inserted property appears non-negated in the integrity constraint, it must be checked. After performing the unification and deleting the (unified) occurrence of *salary* in the integrity constraint, we must try to show that: not $aux(\text{john},[10,20])$, which is not possible. This means the integrity constraint was not violated by the transaction.

Alternatively, if we had entered that John's salary would be 40k during the interval [10,30], then: not $aux(\text{john},[10,30])$ would have been true, showing that the integrity constraint was indeed falsified. □

Every time an event or update is executed on the database, the integrity constraints are checked against the cache on which corresponding introductions and removals are performed (using the methodology already described)[11]. Now, to update the cache, a **replace** update on the database is transformed into a **del** followed by an **ins** on the database, and they have corresponding effects on the cache. Our specification of the continuous checking of the integrity constraints

[11] This continuous checking is necessary for being sure that the key assumption that the integrity constraints are satisfied before every event or update execution holds.

would not allow to check the integrity constraints after the first, **del**, part of the **replace**, but only after the second, **ins**, part has been completed.

The checking procedure introduced above can be easily implemented right into the caching mechanism. We used PROLOG for this. The mechanism must collect all the properties that were inserted (along with their intervals) and deleted as a result of a transaction into two lists: *added* and *deleted*. The integrity constraints are specified as rules whose heads contain two lists: the list of properties that appear non-negated (and their intervals) and the list of properties that appear negated (in this case, no intervals are needed) in the integrity constraint, as follows (n is simply the number of the integrity constraint):

$$\begin{aligned} ic(n,[\ &\overbrace{(p_1(\bar{x}_1)-I_1),\ldots}^{\text{non-negated properties}}\ ],[\overbrace{q_1(\bar{y}_1),\ldots,q_m(\bar{y}_m)}^{\text{negated properties}}]) \leftarrow \\ &mholds\_for(p_1(\bar{x}_1),I_1),\ldots,f(p_1(\bar{x}_1),I_1,\ldots),\ \text{not}\ aux(p_1(\bar{x}_1),I_1,\ldots). \end{aligned}$$

To perform the checking, each integrity constraint is considered separately. If the list of non-negated properties has an element in common with the *added* list, the occurrence of the element in the integrity constraint is unified with the modified property, then deleted from the integrity constraint and the remaining integrity constraint is checked. If the list of negated properties has an element in common with the *deleted* list, the occurrence of the element is unified with the property and the instantiated integrity constraint is checked.

Example 4: *The salary of an employee cannot decrease.*
This integrity constraint can be formulated as:

```
ic(1,[(salary(E,S)-I),(salary(E,S1)-I1)],[]) :-
    mholds_for(salary(E,S),I), mholds_for(salary(E,S1),I1),
    S < S1, lessEq(I1,I).
```

Which means that there cannot be an employee with salaries S (in the interval I) and S1 (in the interval I1) such that the the lowest salary (S) occurs in an interval I $\geq$ I1.

The head of the rule contains the integrity constraint number, 1, the list of properties that appear non-negated in the integrity constraint (in this case, both occurrences of *salary*) and the list of negated properties (which is empty in this case). Suppose that the cache contains `mholds_for(salary(john,40000), [5,20])`. If a transaction stating that John's salary will be 30000 in [20,30] (which violates the integrity constraint) is entered in the database, then the *added* list will contain `[salary(john,30000)-[20,30]]`. Since the the list of non-negated properties does have elements in common with the *added* list, we unify the first element of the non-negated list with the sole element of *added* (this instantiates the variables `E, S, I`). Then we delete the first occurrence of *salary* in the right hand side of the rule and try to prove

```
mholds_for(salary(john,S1),I1), 30000 < S1, lessEq(I1,[20,30]).,
```

which is true, since the cache contains `mholds_for(salary(john,40000), [5,20])`. This indicates that the transaction violated the integrity constraint.

5 Further and Related Work

Our BtEC based specification of a temporal database supports the essential features of available and well-known temporal database specifications, like TSQL2 [15]. Some important differences with TSQL2 have to do with the fact that the BtEC has a full logical specification. In particular, automated reasoning is possible from the specification. BtEC also supports a much wider class of updates, the ones that are domain specific and correspond to the primitive events of the event calculus. Also more general integrity constraints are supported by BtEC. On the other side, implementation and optimization issues, already addressed in TSQL2, are still and open subject in BtEC[12].

S. M. Sripada, in [18], extends the event calculus to support transaction time. His framework is based on the original event calculus [8], and is capable of handling both proactive and retroactive updates, as well as correction of errors (by means of a *revise* event, which eliminates all the effects of a given event). In [19], an efficient implementation of the event calculus for temporal database applications is presented. This approach, called Digest, stores all the information about the time periods than can be deduced from the database, and, whenever a new event is added, the Digest is updated incrementally. The problem of integrity constraint checking is not addressed.

Our formalism, being based on time points instead of named intervals, is much simpler than Sripada's. BtEC (without cache) consists of only 9 rules, as opposed to Sripada's 24 rules. Updates in BtEC can be done by specifying time intervals rather than pairs of (possibly artificial) events, which is easier to specify and more natural and similar to traditional database updates. Also, all types of update that appear in temporal databases are available in BtEC. Corrections in our model are made with a **replace** event, which allows the replacement of a single property as opposed to revising an event (which alters all properties affected by it).

The caching mechanism of Sect. 3 serves a similar purpose to the Digest approach, improving the query answering response time. However, we also take advantage of the caching mechanism to enhance integrity constraint verification.

With respect to integrity constraints involving both valid and transaction time, in particular, temporal specialization dependencies [4], it is possible, as shown in [10], to optimize the BtEC specification on the basis of a compilation of the integrity constraints into the BtEC rules.

In a temporal database, multiple values for a single data item may appear, when different values are assigned to it at different transaction times and in the same valid time point. Thus, a problem of *simultaneous value semantics*, SVS, appears [3]. As shown in [10], it is possible to specify and implement SVS in an extension of the bitemporal event calculus with relatively few changes, including *decision time* [3], that is used to determine the ordering of events.

[12] Nevertheless, a prototype implemented in PROLOG, that incorporates all the features mentioned in this paper (and others) has been developed [10]. Wrt integrity constraint checking, a rollback predicate has been implemented.

References

1. M. Böhlen. *Managing Temporal Knowledge in Deductive Databases.* PhD thesis, Swiss Federal Institute of Technology Zürich, 1994.
2. L. Chittaro and A. Montanari. Efficient Temporal Reasoning in the Cached Event Calculus. *Computational Intelligence*, 12(3):359–382, 1996.
3. O. Etzion, A. Gal, and A. Segev. Extended Update Functionality in Temporal Databases. In *Temporal Databases - Research and Practice*, LNCS, pages 56–95. Springer-Verlag, 1998.
4. C. Jensen and R. Snodgrass. Temporal Specialization and Generalization. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):954–975, December 1994.
5. C. Jensen, M. Soo, and R. Snodgrass. Unifying Temporal Data Models via a Conceptual Model. Technical Report TR 93-31, Dept. of Computer Science, University of Arizona, September 1993.
6. R. Kowalski. Database Updates in the Event Calculus. *Journal of Logic Programming*, 12(162):121–146, 1992.
7. R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In *Proceedings of the 13th International Conference on Very Large Databases*, 1987.
8. R. Kowalski and M. Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4:67–95, 1986.
9. J. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.
10. C. A. Mareco. Especificación e Implementación de Bases de Datos Temporales en un Cálculo de Eventos Extendido. Master's thesis, Departamento de Ciencia de la Computación. Pontificia Universidad Católica de Chile, 1999.
11. J.-M. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18:227–253, 1982.
12. J.-M. Nicolas and K. Yazdanian. Integrity Checking in Deductive Databases. In H. Gallaire and J. Minker, editors, *Logic and Databases.* Plenum Publishing Corp., 1978.
13. M. Shanahan. Prediction is Deduction but Explanation is Abduction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1055–1060, 1989.
14. R. Snodgrass. An Overview of TQuel. In Tansel et al., editors, *Temporal Databases.* Benjamin/Cummings, 1993.
15. R. Snodgrass, editor. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995.
16. R. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer*, pages 35–42, September 1986.
17. R. Snodgrass, M. Böhlen, C. Jensen, and A. Steiner. Transitioning Temporal Support in TSQL2 to SQL3. In *Temporal Databases - Research and Practice*, LNCS, pages 150–194. Springer-Verlag, 1998.
18. S. M. Sripada. A Logical Framework for Temporal Deductive Databases. In F. Bancilhon and D. J. DeWitt, editors, *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 171–182. Morgan Kaufmann, 1988.
19. S. M. Sripada. Efficient Implementation of the Event Calculus for Temporal Database Applications. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 99–113. MIT Press, 1995.
20. A. Steiner. *A Generalisation Approach to Temporal Data Models and their Implementation.* PhD thesis, Swiss Federal Institute of Technology, Zürich, 1998.