The Abductive Event Calculus as a General Framework for Temporal Databases

Kristof Van Belleghem Marc Denecker Danny De Schreye

Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium.
e-mail : {kristof, marcd, dannyd}@cs.kuleuven.ac.be

Abstract. In earlier work, we have shown that the formalism of abductive logic programs with FOL integrity constraints provides, under a completion semantics, the same declarative expressivity for representing incomplete information as full first order logic. We have shown how the combination of this formalism with a variant of the Event Calculus of Kowalski and Sergot results in a correct and very expressive framework for temporal reasoning and representation. In this paper we demonstrate how this Abductive Event Calculus formalism provides a general framework for the representation and use of temporal databases. On the declarative level, it is particularly convenient for the representation of incomplete knowledge. Complementary, on the procedural level, we are able to provide a number of simple algorithms using abduction and deduction to test the consistency of the base, answer queries, update the database, handle complex formulas and resolve inconsistency. Furthermore, the use of the database for general temporal problem solving is possible using the known Event Calculus and Logic Programming methods. In particular we show how planning is possible in this kind of temporal database.

1 Introduction

The Event Calculus (see [14]) is a well-known formalism for temporal representation and reasoning. The basic concepts of the formalism are events and properties: events initiate and terminate periods of time during which properties hold.

The Event Calculus has been modified in several ways, for example in [20], [9], [17] and [13], mainly to simplify the ontology and to eliminate problems occurring because of bidirectional persistence of properties (forward as well as backward in time). In the context of the Event Calculus, [8], [19] and [17] have introduced abduction to solve planning problems and [7] showed how abduction can be used to solve general temporal postdiction problems in the presence of incomplete information.

Though abduction is clearly a useful computational paradigm, it was not recognized earlier that the formalism of abductive logic programs with first order logic constraints provides the same declarative expressivity for representing

incomplete information as full first order logic (FOL). This was shown in [5], where it was exploited to provide an implementation of the A language of [10] in the Situation Calculus formulated as an abductive logic program.

In this paper we demonstrate how a temporal database with incomplete information can be formalized in the Abductive Event Calculus and how a suitable abductive procedure like SLDNFA ([6]), which satisfies sufficiently strong sound- and completeness results, can be used to implement the functionality of the database and to use the data for problem solving.

Kowalski ([13]) has argued earlier that the Event Calculus in Logic Programming can be used to formalize the evolution of a database system. The work presented in our paper addresses different issues than Kowalski's work, in particular the representation of incomplete information in Abductive Logic Programming and the use of an abductive procedure for implementing the functionality of the database.

The paper is organised as follows. In the next section, we briefly introduce the Event Calculus. Section 3 describes the abductive extension to Logic Programming, its semantics and the abductive procedure. In section 4 we specify the considered type of temporal database and in section 5 we show how it can be represented and used with the Abductive Event Calculus.

2 The Event Calculus

In the Event Calculus, information is represented in Horn clauses augmented with *negation as failure*. The following axioms define a simplified version of the Event Calculus, which we use as a basis for our framework:

$$\begin{array}{ll} holds\_at(P,T) & \leftarrow happens(E), E << T,\ initiates(E,P), \\ & \quad not\ clipped(E,P,T). \\ clipped(E,P,T) & \leftarrow happens(C),\ in(C,E,T), \\ & \quad terminates(C,P). \\ in(C,E,T) & \leftarrow E << C, C << T. \end{array}$$

$happens(E)$ holds if the event E occurs. Only one event is allowed to occur at any one time point, which makes it possible to represent events by their time of occurrence. In other words, we consider events to be special time points. The relation $<<$ defined on events and other time points is a strict linear order: our theory contains axioms ensuring irreflexivity, antisymmetry, transitivity and linearity, but they are left implicit.

In general, the actions associated with an event determine which properties are initiated or terminated by it. This is formulated in domain dependent axioms, for example

$$\begin{array}{ll} initiates(E, has(x,b)) & \leftarrow act(E, give(y,b,x)). \\ terminates(E, has(y,b)) & \leftarrow act(E, give(y,b,x)). \end{array}$$

3 Abduction

The first order logic theory represented by an Event Calculus program is usually defined as the one corresponding to its Clark completion semantics ([3]). The well-known SLDNF procedure can then be used to make deductive inferences. However, Clark completion semantics requires complete knowledge about the problem domain: incomplete data can not be represented. Furthermore, if we want to provide a wider functionality than the answering of queries (for example, use of the database for planning), deduction alone will not suffice.

These restrictions can be overcome in the following way: on the representational level, our theories are to be interpreted according to Console's completion semantics ([4]) for abductive logic programs, augmented with general first order logic constraints. This semantics allows for the occurrence of *undefined* predicates. Thus, incomplete knowledge can be represented.

On the level of problem solving, we will use *abduction* as well as deduction. This, of course, requires an abductive proof procedure: given a set of logic formulas F (facts and rules about the problem domain) and a number of conclusions G, an abductive procedure attempts to find a set of additional facts Δ such that

- $F + \Delta$ is consistent.
- $F + \Delta \models G$.
- Δ is minimal: no subset of Δ exists that satisfies the first two conditions.

The minimality condition is not always added. The facts allowed in Δ are usually constrained to obtain useful results (for example to avoid the solution $\Delta = \{G\}$).

In general, abduction can be used to deal with incomplete knowledge and to solve diagnosis and planning problems, by constraining the facts in Δ in the appropriate way. In general the predicates allowed in Δ are those we have incomplete knowledge about, in other words the undefined predicates. In the sequel we refer to these as *abducible* or *abductive*. For example, in planning problems we have incomplete knowledge about the actions occurring in the plan and the time relations between them, so actions and $<<$ will be abducible. We try to find the sequence of actions necessary to prove the goal, which is the desired end state. In a similar way postdiction (diagnosis) and indeterminism can be modeled, as demonstrated in [7]. We return to the topic of planning later, in the context of a temporal database.

3.1 The SLDNFA proof procedure

There have been several attempts to build an abductive proof procedure and to use it in the context of temporal reasoning. Examples can be found in [8], [19], [17] and [11].

In our proposal we use the SLDNFA procedure described in [6] and [7]. This procedure, an extension of the SLDNF resolution of Logic Programming ([12], [3]) that can deal with abductive predicates, can handle deduction as well as abduction and allows for a correct treatment of non-ground abducible atoms,

which is necessary in our applications. Its soundness and completeness with respect to Console completion semantics are proven in [6].

The implementation of SLDNFA we are using has a number of features making it more fit for planning in the Event Calculus and for general temporal reasoning. A necessary feature is of course the possibility to indicate which predicates are abducible, depending on the intended kind of problem solving and the available knowledge about the problem domain.

Another feature is the possibility to use constraints "$false \leftarrow A_1, \ldots A_n$". These constraints are handled by adding $A_1, \ldots A_n$ to a list of goals for which finite failure needs to be proven, thus ensuring $\neg false$.

Specifically related to the Event Calculus is the fact that iterative deepening is added, which allows us to find the shortest solution (the one including the smallest number of events) first. It is also possible to indicate a maximum number of events to limit the search. Finally, a special constraint module is implemented that makes sure the $<<$ relation on events constitutes a linear order (actually, a partial order will be returned if all of its possible linearizations are valid solutions).

4 Specification of the considered type of temporal database

4.1 Topology of time

When representing time, we have a choice of several topologies, for example point-based or interval-based time, and numerical or non-numerical time. We allow both time points and intervals in our database, considering intervals as periods of time started and ended by a time point. Currently SLDNFA does not support numerical constraints, so we will not use a numerical time line. The addition of these numerical constraints is one of our further research issues.

4.2 Contents of the database

The data we aim to represent in the database are formulas representing the truth value of properties during intervals and at time points, and the change of truth values at certain time points. Further, formulas representing the order on time points and the relations between intervals will be used. We choose the following set of basic formulas, with P an atom:

- $holds\_at(P, T)$: P is true at time point T.
- $holds(P, int(t_1, t_2))$: P is true throughout the interval $int(t_1, t_2)$. This interval does not need to be "maximal": P may remain true after t_2 or can be true already before t_1.
- $notholds(P, int(t_1, t_2))$: P is false throughout the interval $int(t_1, t_2)$.
- $on(P, T)$: P's value changes from false to true at time point T.
- $off(P, T)$: P changes from true to false at T.

The possible relations between time points and intervals are represented by the following formulas. (In the case of intervals, we distinguish thirteen possible relations, based on those defined in [2], though some names may differ.)

- $T_1 = T_2$: T_1 and T_2 are the same time point.
- $T_1 << T_2$: T_1 is chronologically before T_2.
- $equal(i_1, i_2)$: i_1 and i_2 are the same interval.
- $meets(i_1, i_2)$: the endpoint of i_1 is the starting point of i_2.
- $overlaps(i_1, i_2)$: i_1 starts before i_2, and ends during i_2.
- $starts(i_1, i_2)$: i_1 is an initiating subinterval of i_2.
- $ends(i_1, i_2)$: i_1 is a terminating subinterval of i_2.
- $during(i_1, i_2)$: i_1 is a subinterval of i_2 that is initiating nor terminating.
- $before(i_1, i_2)$: i_1 lies entirely before i_2.
- $after(i_1, i_2)$: inverse of *before*.
- $metby(i_1, i_2)$: inverse of *meets*.
- $overlapped(i_1, i_2)$: inverse of *overlaps*.
- $startby(i_1, i_2)$: inverse of *starts*.
- $endby(i_1, i_2)$: inverse of *ends*.
- $contains(i_1, i_2)$: inverse of *during*.

We can build more complex expressions by combining these basic formulas with logical connectives and quantifiers : if P and Q are valid expressions, then $\neg P$, $P \& Q$, $P \vee Q$, $P \oplus Q$ (exclusive or), $P \Rightarrow Q$, and $P \Leftrightarrow Q$ are valid as well, and if $P(x)$ is an expression, then so are $\forall x : P(x)$ and $\exists x : P(x)$.

4.3 Use of the database

The formulas defined above determine the possible contents of our temporal database. The functionality we require of such database is the following:

- testing whether a database D is consistent.
- answering normal Logic Programming queries as well as more complex ones. Since our data may be incomplete, we distinguish two types of query:
  1. "Is Q necessarily true in the database D ?" (does D $\models$ Q hold?)
  2. "Is Q possible in the database D ?" (is Q + D consistent ?).
- updating the database, with a consistency check of the new data.
- in the case of inconsistency, proposing solutions to restore consistency.
- finally, and maybe most importantly, extending the expressivity to make it possible to use the database for problem solving, in particular planning.

5 A general solution using the Abductive Event Calculus

5.1 Representation of data

We consider our data as a theory that consists of two parts. The first part is a logic program defining all basic formulas in terms of primitive Event Calculus predicates. The second part contains the real data in the base. These are considered integrity constraints on the possible states (models) of the database.

The basic formulas are defined by the following rules:

$$
\begin{array}{ll}
holds\_at(P,T) & \leftarrow happens(E), E << T,\ initiates(E,P), \\
 & \quad not\ clipped(E,P,T). \\
holds(P,int(T_1,T_2)) & \leftarrow interval(T_1,T_2),\ holds\_from(P,T_1), \\
 & \quad not\ clipped(T_1,P,T_2). \\
notholds(P,int(T_1,T_2)) & \leftarrow interval(T_1,T_2),\ notholds\_from(P,T_1), \\
 & \quad not\ started(T_1,P,T_2). \\
holds\_from(P,E) & \leftarrow initiates(E,P). \\
holds\_from(P,E) & \leftarrow holds\_at(P,E),\ not\ terminates(E,P). \\
notholds\_from(P,E) & \leftarrow terminates(E,P). \\
notholds\_from(P,E) & \leftarrow not\ holds\_at(P,E),\ not\ initiates(E,P). \\
started(E,P,T) & \leftarrow happens(C),\ in(C,E,T),\ initiates(C,P). \\
clipped(E,P,T) & \leftarrow happens(C),\ in(C,E,T),\ terminates(C,P). \\
on(P,E) & \leftarrow initiates(E,P),\ not\ holds\_at(P,E). \\
off(P,E) & \leftarrow holds\_at(P,E),\ terminates(E,P). \\
in(C,E,T) & \leftarrow E << C, C << T.
\end{array}
$$

It follows from our definitions that the interval $int(t_1,t_2)$ actually denotes the interval $]t_1,t_2]$, containing its endpoint but not its starting point. This choice is made because working with closed intervals can lead to inconsistencies (one time point can belong to two intervals with different values for the same property), while open intervals lead to time points where properties are undefined. A choice between the two types of halfopen intervals is easy: the definition of the Event Calculus naturally leads to the form $]t_1,t_2]$.

The chronological relations between intervals are expressed in terms of relations between their starting points and end points. This allows us to use the linear time constraint module of SLDNFA for reasoning on them.

$$
\begin{array}{ll}
equal(int(T_1,T_2),int(T_1,T_2)) & \leftarrow interval(T_1,T_2). \\
meets(int(T_1,T_2),int(T_2,T_3)) & \leftarrow interval(T_1,T_2),\ interval(T_2,T_3). \\
overlaps(int(T_1,T_2),int(T_3,T_4)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_4), \\
 & \quad T_1 << T_3,\ T_3 << T_2,\ T_2 << T_4. \\
starts(int(T_1,T_2),int(T_1,T_3)) & \leftarrow interval(T_1,T_2),\ interval(T_1,T_3), \\
 & \quad T_2 << T_3. \\
ends(int(T_1,T_2),int(T_3,T_2)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_2), \\
 & \quad T_3 << T_1. \\
during(int(T_1,T_2),int(T_3,T_4)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_4), \\
 & \quad T_3 << T_1,\ T_2 << T_4. \\
before(int(T_1,T_2),int(T_3,T_4)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_4), \\
 & \quad T_2 << T_3. \\
after(int(T_1,T_2),int(T_3,T_4)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_4), \\
 & \quad T_4 << T_1. \\
overlapped(int(T_1,T_2),int(T_3,T_4)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_4), \\
 & \quad T_3 << T_1,\ T_1 << T_4,\ T_4 << T_2.
\end{array}
$$

$$
\begin{array}{ll}
metby(int(T_1,T_2),int(T_3,T_1)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_1). \\
startby(int(T_1,T_2),int(T_1,T_3)) & \leftarrow interval(T_1,T_2),\ interval(T_1,T_3), \\
 & \quad T_3 << T_2. \\
endby(int(T_1,T_2),int(T_3,T_2)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_2), \\
 & \quad T_1 << T_3. \\
contains(int(T_1,T_2),int(T_3,T_4)) & \leftarrow interval(T_1,T_2),\ interval(T_3,T_4), \\
 & \quad T_1 << T_3,\ T_4 << T_2. \\
interval(T_1,T_2) & \leftarrow happens(T_1),\ happens(T_2), \\
 & \quad T_1 << T_2.
\end{array}
$$

These rules are interpreted under Console completion semantics ([4]) with happens/1, initiates/2, terminates/2 and $<<$ /2 as the undefined predicates. The definitions of other predicates are completed as in Clark completion semantics.

Another important remark is that the "free equality theory", which states that constants and terms with different names are unequal, holds for all terms and constants, except for time points. Time points with different names can be equal and must be treated as skolem constants.

Finally, as indicated earlier, $<<$ is a linear order on time points and events.

The rules defined so far determine the *meaning* of our database. The real data are considered integrity constraints on the possible models of this database. These can be basic formulas as well as complex expressions. Some examples:

$$
\begin{array}{l}
notholds(has(john,book_1),int(t_1,t_2))). \\
\forall(T) : holds\_at(p,T). \\
meets(int(T_1,T_2),int(T_3,T_4)) \oplus metby(int(T_1,T_2),int(T_3,T_4)). \\
holds\_at(has(X,O),T), holds\_at(has(Y,O),T) \Rightarrow X = Y.
\end{array}
$$

The data can be very incomplete, so possibly many different models exist. For example, consider the database containing only two simple constraints:

$$
\begin{array}{l}
holds(has(john,book_1),int(t_1,t_2)). \\
notholds(has(mary,book_2),int(t_3,t_4)).
\end{array}
$$

We do not know anything about John having his book outside of the interval $int(t_1,t_2)$. He can own it all the time, or only during the mentioned time period, or during a period that starts at t_1 but continues after t_2, and so on. Likewise for Mary's book we have many possible models. Finally we have no information at all concerning the temporal relation linking $int(t_1,t_2)$ and $int(t_3,t_4)$. These periods can overlap, be disjoint, be equal, etc. Many different models may correspond to different solutions for the undefined predicates.

5.2 Basic functionality of the database

Now that we have expressed the meaning of our data, we can determine how to use them. We first describe how the basic functionality of the database is provided in the special case where only basic formulas are allowed as constraints.

One very important task, useful for consistency testing as well as query answering, is the generation of a logical model for a set of data. To find such models,

we use abduction. As indicated earlier, our undefined predicates are happens, $<<$, initiates and terminates. These predicates are the abductive ones for the procedure. To find a model for a set of data, we collect these data in a goal, and try to build a proof for this goal using the definitions of the basic formulas and a number of abduced new facts. The resulting set of abduced facts (if it exists) forms, in a sense, a model for the data: $Comp(P + \Delta) \models F$, where P is the program consisting of defining rules, Δ is the set of abduced facts, F is the goal to be proven (the data), and $Comp(P)$ is the Clark completion of P. Every abduced solution is an assignment of truth values to the undefined predicates, and corresponds to one possible model for the data. If no solution can be abduced, the data are inconsistent.

Testing the consistency of a database is now straightforward: we check whether $P + F$ is consistent, where P is the program containing the basic definitions and F are the data in the database, by attempting to abduce a model in which F holds. The data are consistent if we find a model, inconsistent if we find failure.

Consider again the database containing the constraints

$$holds(has(john, book_1), int(t_1, t_2)).$$
$$notholds(has(mary, book_2), int(t_3, t_4)).$$

To check its consistency, we form the goal

$$\leftarrow holds(has(john, book_1), int(t_1, t_2)),\ notholds(has(mary, book_2), int(t_3, t_4)).$$

and find for example the abduced facts

$$happens(t_1). \quad happens(t_2). \quad happens(t_3). \quad happens(t_4).$$
$$t_1 << t_2 << t_3 << t_4.$$
$$initiates(t_1, has(john, book_1)).$$

which proves consistency of the data.

Answering queries can be done in a similar way. If we want to know whether Q is possible in the database F, we try to abduce a solution that (added to P) entails $F + Q$. For example, using the same data as above, the query "Is it possible that Mary owns book 2 at t_1" will be solved by attempting to prove

$$\leftarrow holds(has(john, book_1), int(t_1, t_2)),$$
$$notholds(has(mary, book_2), int(t_3, t_4)),$$
$$holds\_at(has(mary, book_2), t_1)).$$

which has as a model for example (omitting the *happens*-facts)

$$t_{new} << t_1 << t_2 << t_3 << t_4.$$
$$initiates(t_{new}, has(mary, book_2)).$$
$$initiates(t_1, has(john, book_1)).$$
$$terminates(t_3, has(mary, book_2)).$$

The answer to the query is therefore affirmative.

If the question is whether Q is *necessarily* true given F, we try to abduce a model for $F + \neg Q$. If we find no model ($P + F + \neg Q$ is inconsistent), we know

that $P + F \models Q$, which is what we were trying to find out. Using the same query as in the previous example, we would end up trying to prove

$$\begin{array}{l} \leftarrow holds(has(john, book_1), int(t_1, t_2)), \\ \quad notholds(has(mary, book_2), int(t_3, t_4)), \\ \quad not\ holds\_at(has(mary, book_2), t_1)). \end{array}$$

which has the solution

$$\begin{array}{l} t_1 << t_2 << t_3 << t_4. \\ initiates(t_1, has(john, book_1)). \end{array}$$

so we can conclude that Mary does not necessarily have book 2 at t_1.

Finally, to update the database, we check whether the resulting data would be consistent, and if so, add the new data item to the base. If inconsistency is detected, a warning results and the update can only be executed through user intervention. For example, if we want to add $holds\_at(has(mary, book_2), t_1))$ to the database, we try to compute a model for

$$\begin{array}{l} \leftarrow holds(has(john, book_1), int(t_1, t_2)), \\ \quad notholds(has(mary, book_2), int(t_3, t_4)), \\ \quad holds\_at(has(mary, book_2), t_1)). \end{array}$$

which succeeds as before. The new data item is then added to the base.

This functionality poses no problem for the SLDNFA procedure, except for the treatment of time constants. SLDNFA considers constants in the data to be normal constants, where they are intended to be skolems. We can solve this problem in the following way: we collect all data $(F_1, F_2, \ldots F_N)$ in the conjunction

$$F_1 \& F_2 \& \ldots \& F_N.$$

We write that conjunction in the form $F(t_1, \ldots t_n)$ where the t_i are our time point skolem constants. In short, we call this expression F. We can then "deskolemize" F: we replace all skolem constants by existentially quantified variables, which results in F':

$$\exists T_1, \ldots T_n : F(T_1, \ldots T_n).$$

Skolem's theorem states that for all P and F, with F' the deskolemization of F as defined above: $P + F$ is consistent $\Leftrightarrow$ $P + F'$ is consistent. Therefore, replacing skolem constants by existentially quantified variables does not change the result of a consistency check or a query.

We now do the following: before calling the SLDNFA procedure, we build a table in which we link every time constant to a variable. In the data we pass to the procedure, we replace every constant by its corresponding variable. As indicated, this does not change the consistency results.

To find the solution corresponding to this obtained consistency result, we combine the answer of the SLDNFA procedure with our table of time constants, where some of the variables may be unified by now. In that case the time constants corresponding to these variables are equal in this solution. We will discuss a detailed example later.

The search space can now be limited to solutions with a bounded number of events in a consistency/inconsistency preserving way: it can be shown that, if the data are consistent and contain only N different time points, there exists at least one model for these data containing 2N or less events. In general 2N even is a substantial overestimation of the needed number of events, and in most cases we find solutions with a number of events equal to or a little greater than N.

5.3 Introduction of complex data and queries

In the previous section we showed how SLDNFA can be used to abduce models for data consisting of only basic formulas. However, in the case of complex queries and data a preceding transformation step is required. This transformation step is based on the Lloyd-Topor transformation described in [16].

The transformation provides a method to transform a program containing non-Horn clauses and complex goals, an *extended program*, to a program containing only Horn clauses augmented with negation as failure. The soundness of the transformation under Clark completion semantics is proven in the article, and this soundness result holds for abductive logic programs under Console completion semantics as well. We recall the essence of the transformation here.

An extended logic program is a program consisting of "general clauses" ([15]). These are rules of the form

$$A \leftarrow W.$$

where A is an atom and W an arbitrary first order logic expression. Any variables in A and free variables in W are considered universally quantified at the beginning of the clause.

The transformation to a normal logic program is performed by replacing general clauses by others using a set of transformation rules, until only Horn clauses — possibly with negation in the body — are left. As an example, we include some of the basic transformation rules:

a) Replace $A \leftarrow W_1, W_2, \ldots, (\forall x_1 \ldots x_m : W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, \neg(\exists x_1 \ldots x_m : \neg W), \ldots W_n.$
b) Replace $A \leftarrow W_1, W_2, \ldots, \neg(V \Leftarrow W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, W, \neg V, \ldots W_n.$
c) Replace $A \leftarrow W_1, W_2, \ldots, \neg(\neg W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, W, \ldots W_n.$
d) Replace $A \leftarrow W_1, W_2, \ldots, \neg(\exists x_1 \ldots x_m : W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, \neg p(y_1 \ldots y_k), \ldots W_n.$
and $p(y_1 \ldots y_k) \leftarrow \exists x_1 \ldots x_m : W.$
where p is a new predicate symbol not occurring in the program, and
$y_1, \ldots y_k$ the free variables in $(\exists x_1 \ldots x_m : W)$.

Similar rules exist for each operator and its negation. A complete list can be found in [16].

The goal of the program can be transformed in the same way:

Replace $\leftarrow W.$
by $\leftarrow answer(x_1 \ldots x_n).$
and $answer(x_1 \ldots x_n) \leftarrow W.$
where $x_1 \ldots x_n$ are the free variables in W.

The resulting rule $answer(x_1 \ldots x_n) \leftarrow W.$ must be transformed further using the rules described above.

5.4 A detailed example

To illustrate how our system handles complex data and time skolems, we solve a small example query in detail. We have a database DB containing two data items, namely

$$holds(has(john, book_1), int(t_1, t_2)).$$
$$holds(has(mary, book_2), int(t_2, t_3)).$$

We want to know if it is possible that, for arbitrary time points a,b and c,

$$holds(has(john, book_1), int(a, b)) \vee notholds(has(mary, book_2), int(a, c))$$

is true. The following query is used:

$\leftarrow poss\_query(DB,$
$or(holds(has(john, book_1), int(a, b)), notholds(has(mary, book_2), int(a, c)))).$

The program collects the data from DB in a list and adds the query to it. All time constants are replaced by variables, and we obtain the following time table:

$t_1 - X_1$	$a - A$
$t_2 - X_2$	$b - B$
$t_3 - X_3$	$c - C$

The goal we want to prove becomes

$[holds(has(john, book_1), int(X_1, X_2)),\ holds(has(mary, book_2), int(X_2, X_3)),$
$or(holds(has(john, book_1), int(A, B)),\ notholds(has(mary, book_2), int(A, C))]$

but the data need to be transformed first. In this case only the disjunction is complex. New rules

$$q_0(A, B, C) \leftarrow holds(has(john, book_1), int(A, B)).$$
$$q_0(A, B, C) \leftarrow notholds(has(mary, book_2), int(A, C)).$$

are added to the program, and we try to solve the following query:

$\leftarrow holds(has(john, book_1), int(X_1, X_2)),$
$holds(has(mary, book_2), int(X_2, X_3)),\ q_0(A, B, C).$

If we ask for a solution with three events, the meta-interpreter replaces time variables by skolem constants, determines the order on these time constants, and abduces the necessary initiations and terminations to prove the goal (using the new rules for q_0 together with the general definitions of the database formulas).

The solution contains three events new_1, new_2 and new_3, where $X_1 = A = new\_1$, $X_2 = B = new\_2$ en $X_3 = C = new\_3$. The abduced order on these events is $new\_1 << new\_2 << new\_3$. The initiations are

$$initiates(new\_1, has(john, book_1)).$$
$$initiates(new\_2, has(mary, book_2)).$$

and terminations are not necessary. The time table now looks like this:

$t_1 - new\_1$	$a - new\_1$
$t_2 - new\_2$	$b - new\_2$
$t_3 - new\_3$	$c - new\_3$

and we can read the following solution

$$t_1 = a,\ t_2 = b,\ t_3 = c,$$
$$t_1 << t_2 << t_3,$$
$$initiates(t_1, has(john, book_1)),$$
$$initiates(t_2, has(mary, book_2)).$$

which is obviously correct. Of course it is not the only solution, and the program will find many more, for example solutions where g is initiated by t_1 or where f gets terminated by some event. One reason for the many solutions is the occurrence of a disjunction in the query.

5.5 Resolving inconsistency

We have extended this program further to help the user resolve inconsistency in the data. We use abduction to propose solutions for the inconsistency, as illustrated in the following example.

Supppose we have three formulas P, Q and R as data. The program collects these data in a list $[P, Q, R]$, which is given — after transformation — as a goal to the meta-interpreter. If the meta-interpreter returns with a solution, the data are consistent and there is no problem.

If no solution is found, and the user has chosen the "resolve inconsistency" option, control returns to the transformation program. This program will undo all changes it made to the data during transformation, and generates a new transformation, only this time not for $[P, Q, R]$, but for $[reject(P), Q, R]$.

The meta-interpreter then tries to explain $reject(P)$ instead of P. This is always possible, because we make reject/1 an abductive predicate. The result is then that an abduced fact $reject(P)$ is written in the solution, while the program continues trying to find a model for $[Q, R]$. The constraint P is dropped, which possibly resolves the inconsistency.

In further attempts every combination of formulas and rejected formulas is checked until a solution is found. In short then, for any fact P the program can either explain P, or abduce $reject(P)$ and ignore P. Looking at the abduced $reject(P)$ facts, the user sees which constraints have been dropped to restore consistency. Of course more than one solution can exist, and the user can choose the best one, whatever "best" means to him.

As an example, a database containing

$$holds(has(john, book_1), int(t_1, t_2))).$$
$$holds(has(mary, book_1), int(t_1, t_2)).$$
$$holds\_at(has(X, O), T),\ holds\_at(has(Y, O), T) \Rightarrow X = Y.$$

is inconsistent, and consistency can be restored by deleting any of the three constraints. One of the proposed solutions would be

$$reject(holds(has(john, book_1), int(t_1, t_2)))).$$
$$happens(t_1). \quad happens(t_2). \quad t_1 << t_2.$$
$$initiates(t_1, has(mary, book_1)).$$

This method is of course quite inefficient, since data are selected for rejection in a random way, without looking for the causes of the inconsistency. There exist, however, solutions to this problem. As mentioned earlier, the meta-interpreter maintains a list of constraints (negative goals) to be satisfied. If we keep track of the data items corresponding to a constraint, and determine which fact violates which constraint, we can use this information in a more intelligent method to resolve inconsistency. We will return to this issue briefly in the discussion.

5.6 Planning

If we want to use a database for planning, we of course need to introduce the concept of action. This is a well-known concept in the Event Calculus and we can use it without any modification. To use the database for planning, we define all possible actions and their effects. Then we make the actions abducible (instead of simply the initiation and termination of properties). Initiations and terminations now follow from the occurring actions. In this way, properties can not arbitrarily change their value as they could in the original model. Every change is now caused by an action.

A system for planning using the Abductive Event Calculus has been developed earlier. This system, based on abducible actions, can almost automatically be combined with our database system. We can then use our database for planning without a problem. Furthermore, the use of actions to explain every initiation and termination can be extended to non-planning problems. The actions then define every possible way in which properties can change. As a result, our models not only contain information about which properties changed value, but also about *why* this happened.

As an example, assume we want to build a very simple plan: John owns a certain book, and we want Mary to have it. The only possible action is giving the book to someone. We add the specification of this action's preconditions and effects to our basic definitions:

$$false \leftarrow act(E, give(Y, B, X)),\ not\ holds\_at(has(Y, B), E).$$
$$initiates(E, has(X, B)) \leftarrow act(E, give(Y, B, X)).$$
$$terminates(E, has(Y, B)) \leftarrow act(E, give(Y, B, X)).$$

We also introduce a special event *start* which occurs before all other events to take care of the first initiations. After this special event, only actions can change the world.

$$\begin{array}{ll} happens(start). & \\ false & \leftarrow happens(T),\ T << start. \\ initiates(start, P) & \leftarrow initially(P). \end{array}$$

and we make *happens*, $<<$, *initially* and *act* abducible.

The database *DB* would contain the formulas

$$holds\_at(has(john, book_1), t_1).$$
$$holds\_at(has(X, B), T),\ holds\_at(has(Y, B), T) \Rightarrow X = Y.$$

and we would try to solve the query

$$\leftarrow poss\_query(DB, [t_1 << t_2,\ holds\_at(has(mary, book_1), t_2)].$$

This gives for example the model

$$\begin{array}{l} happens(start).\quad happens(t_1).\quad happens(t_2).\quad happens(t_3). \\ start << t_1 << t_3 << t_2. \\ initially(has(john, book_1)). \\ act(t_3, give(john, book_1, mary)). \end{array}$$

which explicitly contains the plan.

As a final remark we can indicate that the original idea with abducible initiations and terminations is just a special case of the proposal using actions, in which every action corresponds to one initiation or termination.

6 Discussion

We have demonstrated how the Abductive Event Calculus provides a general framework for the representation and use of temporal databases. Both time points and intervals can be used. The representation of incomplete knowledge is perfectly possible.

Abduction provides a straightforward way to generate models for a set of data. This allows us to check consistency and to answer queries. Complex data can be handled using a preceding transformation step, and deskolemization allows us to represent time points that may be equal to each other.

Using an abductive predicate *reject/1* we introduced a simple method to help us resolve inconsistency by rejecting certain data. Finally, we have shown how the database can be used for planning by introducing actions and making them abducible.

In general, the proposed solutions are not very efficient. We mainly provide a theoretical framework for representing incomplete temporal databases, and give a number of simple algorithms to illustrate how this database can be used. These algorithms may be the basis of research on more efficient implementations.

One of our own further research goals, apart from the introduction of numerical time constraints, is the improvement of the efficiency of our procedures.

On one hand, we hope to obtain this greater efficiency by incorporating CLP techniques and tabulation.

On the other hand, we already find interesting ideas in the literature. For example, in [22] we find an algorithm for resolving inconsistency in a network of interval relations, based on the work in [1]. There, for each pair of intervals a list of possible relations between these intervals is maintained. If ever no possible relations are left between any two intervals, the data are inconsistent. Weigel and Bleisinger have modified and extended this procedure to efficiently derive solutions for the inconsistency.

Their solutions show some similarity to our approach, but work only on interval relations instead of general data. This allows for more efficient algorithms, especially if an incremental consistency checker is used.

Another approach to the representation of temporal databases can be found in [18]. A database is considered a collection of maximal intervals throughout which certain properties hold. For each property a list of such intervals is maintained. Incomplete knowledge can be represented by skolemizing the end points of an interval, and constraints on these end points can be expressed. The framework shows some similarity to ours, though no explicit events are used and only maximal intervals are represented. The system can be mapped to ours, however, and some of its proposed algorithms may be useful to us.

A possible extension for temporal databases is the introduction of a notion of belief. The representation of belief in a theory of time was addressed in [2]. One proposal to incorporate this notion in a temporal database is described in [21]. To incorporate a similar extension in our system, further research will be necessary.

The most important aspect of our framework is probably that it allows for the data in the base to be in the same language as the applications working with them. This is clearly illustrated by the straightforward extension for planning. Thus we hope to show that the Abductive Event Calculus is not only useful in several distinct temporal reasoning domains, but also provides a link between them.

Acknowledgements

Kristof Van Belleghem is partly supported by ESPRIT BR project Compulog II and partly by the Belgian IWONL. Marc Denecker is supported by Dienst Onderzoekscoordinatie, K.U.Leuven. Danny De Schreye is a senior research associate of the Belgian NFWO. We thank anonymous referees for valuable comments.

References

1. J. F. Allen. Maintaining Knowledge About Temporal Intervals. *CACM*, 26(11):832–843, 1983.
2. J. F. Allen. Towards a General Theory of Action and Time. *Artifical Intelligence*, 23(11):123, 1984.

3. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and databases*, pages 293–322. Plenum Press, 1978.
4. L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.
5. M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming*. PhD thesis, Department of Computer Science, K.U.Leuven, 1993.
6. M. Denecker and D. De Schreye. SLDNFA; an abductive procedure for normal abductive programs. In K. Apt, editor, *Proceedings of the International Joint Conference and Symposium on Logic Programming, Washington*, 1992.
7. M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proceedings of ECAI 92, Vienna*, 1992.
8. K. Eshghi. Abductive planning with event calculus. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th ICLP*, 1988.
9. C. Evans. The Macro-Event Calculus: Representing Temporal Granularity. In *Proceedings of PRICAI, Tokyo*, 1990.
10. M. Gelfond and V. Lifschitz. Describing Action and Change by Logic Programs. In *Proc. of the 9th Int. Joint Conf. and Symp. on Logic Programming*, 1992.
11. A. Kakas and P. Mancarella. Constructive abduction in logic programming. Technical report, Dipartimento di Informatica, University of Pisa, 1993.
12. R. A. Kowalski. *Logic for problem solving*. Elsevier Science Publisher, 1976.
13. R. A. Kowalski. Database updates in the event calculus. *Journal of Logic Programming, 1992*, 1992.
14. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(4):319–340, 1986.
15. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
16. J. Lloyd and R. Topor. Making prolog more expressive. *Journal of logic programming*, 1(3):225–240, 1984.
17. L. Missiaen. *Localized abductive planning with the event calculus*. PhD thesis, Department of Computer Science, K.U.Leuven, 1991.
18. A. Porto and C. Ribeiro. Temporal inference with a point-based interval algebra. In *Proceedings of ECAI 92, Vienna*, pages 374–378, 1992.
19. M. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of IJCAI 89*, page 1055, 1989.
20. M. Shanahan. Representing continuous change in the event calculus. In *Proceedings of the 9th ECAI*, page 598, 1990.
21. S. Sripada. A metalogical programming approach to reasoning about time in knowledge bases. In *Proceedings of IJCAI 93*, 1993.
22. A. Weigel and R. Bleisinger. Support for resolving Contradictions in Time Interval Networks. In *Proceedings of ECAI 92, Vienna*, pages 379–383, 1992.

This article was processed using the LaTeX macro package with LLNCS style