Answering Historical Queries in Databases using Relevance

Bernardo Siu and Leopoldo Bertossi
Pontificia Universidad Catolica de Chile
Escuela de Ingenieria
Depto. de Ciencia de la Computacion
Casilla 306, Santiago 22, Chile
E–mail: {bsiu,bertossi}@ing.puc.cl

Abstract

In this paper we present a syntactical class of historical queries in databases, and an algorithm for answering them. More precisely, we consider the problem of answering queries about the evolution of the database along a sequence of transactions. To make the algorithm efficient, we introduce procedural notions of relevant transactions and tuples, that allow us to know which transactions to execute and which tuples to update. We start from Ray Reiter's formalism [14, 16] based on the situation calculus [11] for specifying database updates. This paper extends previous work by Aris Zakinthinos on historical queries [19], which is also based on Reiter's work. Our algorithms have been implemented in SCDBR [3, 2, 1], a computational system that is able to reason from and about database specifications written in Reiter's formalism. We discuss some implementation issues.

Keywords: Database Updates, Historical Queries, Knowledge Representation, Situation Calculus, Relevance

1 Introduction

In this paper we present a syntactical class of historical queries on databases for which the process of answering them can be done automatically and efficiently. We follow Ray Reiter's proposal [14, 16] to specify the database evolution in the situation calculus, a first–order language for representing knowledge about dynamical worlds [11]. The database updates are specified in Reiter's formalism for solving the frame problem in the situation calculus [13]. This paper considerably extends previous work by Aris Zakinthinos on historical queries [19], which was also based on Reiter's proposal.

More specifically, we are interested in the following problem: Given the specification of the dynamics of a database, the initial (physical) relational database D_0, and a list of legal atomic transactions $T = [a_1, \ldots, a_n]$, to be executed in sequence on D_0, and then producing the (possibly virtual) databases $D_1, \ldots, D_n$, we want to ask queries like "*Is it always the case, i.e. in the whole evolution of the database, that a given property holds*"? or "*Has a given property been true at some state in that evolution*?". More complex queries are obtained by mixing basic queries of these forms. We want to answer such queries without having to physically update the whole database after executing each transaction, keeping all versions, and inspecting (possibly) all of them. We would rather like to answer the queries by doing a combination of a kind of list processing on T, minimal updates of the database, and satisfaction of temporal constraints.

In order to characterize the minimal updates to be performed, we introduce procedural notions of *relevant actions* and *relevant (entries in) fluents*. An entry in a fluent (or an action) will be relevant to a query if the answer to the query depends potentially on that fluent (or that action). We present an algorithm that computes the fluents and actions that are relevant to a query.

The work presented in this paper has been implemented in SCDBR [3, 2, 1], an automated system written in PROLOG, that is able to reason from and about database specifications written in Reiter's formalism.

The rest of this paper is organized as follows: In section 2, we recall Reiter's approach to database updates. We also introduce some notation and conventions we will use in the rest of the paper. In section 3, we show examples of historical queries in intuitive terms and how to answer them with Zakinthinos' approach. In section 4, we present a *canonical query format*. In section 5, we present a general algorithm for answering queries in the canonical query format, and some preliminary ideas for its optimization. Developing those ideas, in section 6, we give an algorithm that determines a (small) set of fluents and actions that are relevant to a given query. Finally, we draw some conclusions from our work.

2 Reiter's Specification of Database Updates

The situation calculus models a dynamic world as evolving from state to state, starting from an initial state S_0. If s is a state term and a is an action term, then the term $do(a, s)$ denotes the state resulting from executing a on s, and is called a successor state of s. The world changes its state only by executing an action. The world is described in terms of dynamic properties, called *fluents* (think of the tables in a database). Formally, fluents are predicates with an extra state argument. Their evolution can be specified by means of *successor state axioms* (SSAs), which characterize when each fluent becomes true at a state of the form $do(a, s)$.

In addition to the SSAs, the specification contains *action precondition axioms*, which are formulas of the form $Poss(A, s) \equiv \Pi_A$, where $Poss(A, s)$ stands for action A being possible at state s. A finite sequence of transactions is *legal* iff each transaction satisfies its precondition at its corresponding execution state. Reiter uses this idea to define the accessibility relation $<$ between states [14, 15]. Basically, $S_i < S_k$ holds iff state S_k can be reached from S_i by executing a legal transaction list.

As said before, for each fluent F there is a SSA that is usually of the form

$$Poss(a,s) \supset F(\bar{x})[do(a,s)] \equiv \gamma_F^+(\bar{x},a)[s] \vee (F(\bar{x})[s] \wedge \neg\, \gamma_F^-(x,a)[s]), \quad (1)$$

where the formulas γ_F^+, γ_F^- are the assumed to be the only conditions under which $F(\bar{x})$ can change its truth value, becoming true or false, respectively, in the state $do(a,s)$. They are obtained from positive effect axioms and negative effect axioms for fluent F, respectively [13]. Formula (1) can be read as: "*The fluent $F(\bar{x})$ holds in state $do(a,s)$ iff the conditions that make it true were satisfied in state s or $F(\bar{x})$ held in state s and the conditions that make it false were not satisfied*". Notice that if neither γ_F^+ nor γ_F^- hold, then the truth value of $F(\bar{x})$ is the same in s and $do(a,s)$. When the conditions γ_F^+, γ_F^- are completely known, Reiter [13] proposes to generate these SSAs. In this way he solves the *frame problem*, that is, the problem of specifying in a compact, succinct, way when a fluent does not change when an action is performed [11].

The formulas on the right–hand side of the SSAs are *simple* in s [13], that is, s appears free in them and no other state term is mentioned (but a tacit universal quantification on a and s is assumed in (1)) .

Formally, an according to [14, 16], a database is specified by a theory of the form $\Sigma = \Sigma_{bd} \cup \Sigma_{ap} \cup \Sigma_{una} \cup \Sigma_{ssa} \cup \Sigma_{S_0}$, where:
(1) Σ_{bd} contains (a) state independent axioms for the domain of discourse and (b) the foundational axioms [9, 12] of situation calculus, which describe the ontology of the situations and include an induction axiom over states.
(2) Σ_{ap} contains, for each named action A, a *precondition axiom* of the form $(\forall\, s) Poss(A,s) \equiv \Pi_A[s]$, stating when action A is possible. The formula $\Pi_A[s]$ is simple in state s and specifies the conditions that make A possible in state s.
(3) Σ_{una} contains unique name axioms for actions and states.

(4) Σ_{ssa} contains the SSAs. For each primitive table (fluent) F, there is one SSA of the form

$$(\forall\, a, s, \bar{x}) Poss(a,s) \supset F(\bar{x})[do(a,s)] \equiv \Phi_F(\bar{x}, a, s),$$

where Φ is a formula simple in s with its free variables among $\bar{x}, a, s$. One important and usual particular case of SSAs is (1).

Sometimes we will refer to formulas $\Phi_{F(\bar{c})}$, meaning that all variables in formula Φ_F must be renamed or instantiated accordingly. For example, if Φ_F is $F(x,y)[s] \vee (\exists\, x')R(x')$, then the formula $\Phi_{F(a,b)}$ will be $F(a,b)[s] \vee (\exists\, x')R(x')$.
(5) Σ_{S_0} is the initial database, a theory that describes the world in state S_0. We will assume that the initial database is relational. We also make the *closed world assumption*. With this assumption and the SSAs, we have complete knowledge in future (legal) states of the database.

3 Some Examples and Previous Work

In the following we give examples of historical queries taken from [14, 16] of the kind mentioned in section 1.

Example 1 We want to know if Sue has always been working in department 13. By "always" we mean in all states generated by a sequence T of transactions executed at S_0. If the term $do(T, S_0)$ denotes the state resulting from executing T at S_0, then the query can be expressed by the formula

$$(\forall s)(S_0 \leq s \leq do(T, S_0) \supset emp(sue, 13)[s]). \tag{2}$$

■

Example 2 Now, with the same T, we want to know if there is a state such that someone is enrolled in courses $c100$ and $m100$.

$$(\exists s)((S_0 \leq s \leq do(T, S_0)) \wedge (enrolled(st, c100) \wedge enrolled(st, m100))[s]). \tag{3}$$

■

The two previous examples have different syntactical forms. However, both of them can be reformulated in a common syntactical representation, as will be shown in section 4. This allows us to use a uniform mechanism for answering them.

In [19], Zakinthinos answers some examples of historical queries. His solution applies *list processing* on the list of transactions (as suggested by Reiter [14, 16]), when applicable, and tables updates. Zakinthinos mechanisms are very efficient in the cases he presents, but rather ad hoc. He does not say anything about the generality and extendibility of his approach. Now we present two examples taken from [19].

Example 3 Consider query (2). Suppose that we have a fluent $emp(x, d, s)$, meaning that the person x is employed in department d in state s. Assume that we have the following SSA for this fluent:

$$\begin{aligned} Poss(a, s) \supset emp(x, d)[do(a, s)] \equiv\ & a = hire(x, d) \vee \\ & emp(x, d)[s] \wedge \neg\,(a = fire(x) \vee a = quit(x)). \end{aligned} \tag{4}$$

As is clear from the SSA, the only actions that can change the truth value of $emp(sue, 13)$ are $hire(sue, 13)$, $fire(sue)$ and $quit(sue)$, so we can do list processing on T: Simply check if fluent $emp(sue, 13)[S_0]$ holds. Then, check that neither $fire(sue)$ nor $quit(sue)$ appear in the transaction list T. If both conditions are met, then answer *Yes*. Otherwise, answer *No*.

This possibility of doing list processing comes from two things: First, the formula $emp(sue, 13)$ we want to check is ground, so a simple bit is enough to "keep track" of it. Second, the syntactical form of the SSA of emp is so simple that, if an action is executed, one does not have to check any fluent to verify if the truth value of emp changes. One just needs to check if the action is mentioned in the SSA of emp. ■

Example 4 Zakinthinos also shows how to solve (some) queries involving variables other than state variables. His algorithm consists basically in the creation and minimal updating of auxiliary tables. For example, we may want to know if there is at least one employee in state $do(T, S_0)$. This is equivalent to checking if the formula

$$(\exists x, d)emp(x, d, do(T, S_0)) \tag{5}$$

holds. The algorithm proposed by Zakinthinos works as follows: Store in a table E all individuals that are employed in state S_0. Then process the list T in this way: Whenever you find an action $hire(x, d)$, add x to E. When you find an action $fire(x)$, remove the entry x from the table E. The answer to the query is given by counting the number of entries that are stored in the final E: If there are no entries in E, answer *No*. Otherwise, answer *Yes*. The algorithm performs a minimal update, since we are only considering the effects of the actions that can affect our query. Even if $hire(x, d)$ and $fire(x)$ have side effects, i.e. they affect fluents other than *emp*, they are not computed.

This algorithm does more than pure list processing, because it needs to keep track of every employee. So, a single bit is not enough and an auxiliary table is required. ■

4 The Canonical Query Format

Throughout this paper, T will be a given legal sequence of atomic transactions of length N, and $S_N = do(T, S_0)$. For technical reasons, we will extend this transaction list by the execution of a new action, *wait*, that does not cause any changes in the fluents and has a precondition axiom identically true[1]. This creates state $S_{N+1} = do(wait, S_N)$.

As we will see later in this section, every query will be represented in terms of universal sub-queries of the form

$$\forall s(s_L \leq s < s_U \supset F[s]), \tag{6}$$

with s_L, s and s_U being variables of sort state, and $F[s]$ being a formula in s that does not mention any term of sort action.

Notice that no terms of sort state other that s_L, s_U and s are admitted in (6). Notice also that the variables s_L, s_U are not quantified in (6). Nevertheless, as we will see soon, they will appear existentially quantified in more complex queries obtained by combining universal sub-queries.

Sometimes we will denote (6) by $UQ_F(s_L, s_U)$. States s_L, s_U are the *lower bound* and *upper bound*, respectively. The set of states that lie between them, including s_L and excluding s_U, will be referred to as the *range* of the universal (sub)query. This asymmetry between s_L and s_U wrt membership to the range, has the positive consequence of $F[s]$ not being forced to be true at any state s when s_L and s_U have the same value (i.e. when the range of the sub-query is empty). This might be desirable for posing some queries (see [18]). The negative consequence is having to add a new state $S = do(wait, do(T, S_0))$ if we want to consider all states along T. Hence the necessity of state S_{N+1}.

For the purpose of posing historical queries, we need to put constraints on the bounds of the sub-queries. This is achieved by means of a function of pairs of states in the list that returns the number of transactions required to reach one state from the other. If $S_0 \leq S_i \leq S_j$, then the distance from S_i to S_j, $dist(S_j, S_i)$, is the (finite) number of atomic transactions to be executed to reach S_j from S_i. $dist(S_i, S_j)$ is defined by $-dist(S_j, S_i)$. A formal definition of distance can be found in [18]. This definition and some arithmetical axioms should have to be included in the specification.

[1] In the case of SSAs generated from effect axioms as in [13], this is achieved by not mentioning *wait* in any of them.

The concept of distance is very general to express constraints: If we want to say that two states S_i, S_j are the same, we can use the constraint $dist(S_i, S_j) = 0$. If we want to say that S_j is reachable from S_i, we can use $dist(S_j, S_i) \geq 0$. If we want to say that S_j corresponds to the state resulting from executing exactly four actions on S_i, a constraint like $dist(S_j, S_i) = 4$ will do.

Definition 1 *The Canonical Query Format (CQF), is:*

$$\exists\, s_{L_1}, s_{U_1}, \ldots, s_{L_M}, s_{U_M} \left(\begin{array}{c} UQ_{F_1}(s_{L_1}, s_{U_1}) \wedge \cdots \wedge UQ_{F_M}(s_{L_M}, s_{U_M}) \wedge \\ (BC_1 \wedge BC_2 \wedge \cdots \wedge BC_r) \wedge \\ IR(S_{N+1}, s_{L_1}, s_{U_1}, \ldots, s_{L_M}, s_{U_M}, \bar{s}_1, \ldots, \bar{s}_c) \end{array} \right) \tag{7}$$

where:

1. $\exists\, s_{L_1}, s_{U_1}, \ldots, s_{L_M}, s_{U_M}$ *is a prefix of existential quantifiers over the bounds of the sub-queries* $UQ_{F_i}(s_{L_i}, s_{U_i})$.
2. $UQ_{F_i}(s_{L_i}, s_{U_i})$, *for* $i = 1, 2, \ldots, M$, *are universal queries.*
3. $(BC_1 \wedge BC_2 \wedge \cdots \wedge BC_r)$ *is a formula that restricts the bounds. Each* BC_i *is a comparison that can be of the form* $dist(s_i, s_j) \geq n$, $dist(s_i, s_j) \leq n$ *or* $dist(s_i, s_j) = n$, *where* s_i, s_j *are either bounds or ground terms of sort state, and* n *is an integer constant.*
4. $IR(S_{N+1}, s_{L_1}, s_{U_1}, \ldots, s_{L_M}, s_{U_M}, \bar{s}_1, \ldots, \bar{s}_c)$ *is a formula of the form:*

$$\bigwedge_{i=1}^{M} (s_{L_i} \leq s_{U_i}) \wedge (\bigwedge_{i=1}^{c} S_0 \leq \bar{s}_i \leq S_{N+1}),$$

where $\bar{s}_1, \ldots, \bar{s}_c$ *are all the terms of sort state mentioned in the query, including the bounds* s_{L_i}, s_{U_i}.[2]

The meaning of formula (7) is the following: Formulas $UQ_{F_i}(s_{L_i}, s_{U_i})$ say that there are bounds such that the formula F_i is true in each intermediate state. Besides, these bounds must satisfy certain constraints, represented by formulas BC_i (BC stands for "boundary constraint"). Formula $IR(S_{N+1}, \ldots)$ simply states that any state term mentioned lies between S_0 and S_{N+1}, and that each upper bound must be equal to or reachable from its respective lower bound (IR stands for "interval restriction"). In the rest of the paper we will assume that formula $IR(S_{N+1}, \ldots)$ is implicit and we will usually omit writing it.

Notice that without the restrictions BC_i, query (7) is always true, because if s_{L_i} and s_{U_i} take the same value for all i with $1 \leq i \leq M$, then all the sub-queries are true. However, there will be nontrivial constraints in any interesting query.

In the following we give some examples of formulas written in the CQF. Other examples can be found in [18]:

[2] If $S_N = do(T, S_0)$, then the foundational axioms of the situation calculus plus the unique names axioms ensure that any state s such that $S_0 \leq s \leq S_N$ must be one of the states generated along T.

Example 5 We want to know if Joe was hired as an employee of a lower rank before becoming a Chief Executive Officer (CEO).

$$\exists\, s_{L_1}, s_{U_1}, s_{L_2}, s_{U_2} \left(\begin{array}{c} \forall\, s_1(s_{L_1} \leq s_1 < s_{U_1} \supset (emp(Joe) \wedge \neg\ ceo(Joe))[s_1]) \ \wedge \\ \forall\, s_2(s_{L_2} \leq s_2 < s_{U_2} \supset (emp(Joe) \wedge ceo(Joe))[s_2]) \ \wedge \\ dist(s_{L_2}, s_{U_1}) \geq 0 \end{array} \right)$$

In this example, the only restriction RB_i that appears is $dist(s_{L_2}, s_{U_1}) \geq 0$. It requires that formula $emp(Joe) \wedge ceo(Joe)$ must be true in states that are reachable from the states that satisfy $emp(Joe) \wedge \neg\ ceo(Joe)$. We say that formula $emp(Joe) \wedge ceo(Joe)$ "follows" formula $emp(Joe) \wedge \neg\ ceo(Joe)$. The (omitted) formula $IR(S_{N+1}, s_{L_1}, s_{U_1}, s_{L_2}, s_{U_2})$ is

$$s_{L_1} \leq s_{U_1} \wedge s_{L_2} \leq s_{U_2} \wedge S_0 \leq s_{L_1}, s_{U_1}, s_{L_2}, s_{U_2} \leq S_{N+1}.$$

■

Example 6 Formula (2) can be rewritten as:

$$\exists\, s_{L_1}, s_{U_1} \left(\begin{array}{c} \forall\, s_1(s_{L_1} \leq s_1 < s_{U_1} \supset emp(sue, 13)[s_1]) \wedge \\ dist(s_{L_1}, S_0) = 0 \wedge dist(s_{U_1}, s_{N+1}) = 0 \end{array} \right) \tag{8}$$

The translation can be understood as follows: Formula $s_{L_1} \leq s_1 < s_{U_1} \supset emp(sue, 13)[s_1]$ says that formula $emp(sue, 13)[s]$ must be true in all intermediate states between s_{L_1} and s_{U_1} (including s_{L_1} and excluding s_{U_1}). Constraints $dist(s_{L_1}, S_0) = 0$, $dist(s_{U_1}, s_{N+1}) = 0$ specify that the lower bound must be equal to S_0 and the upper bound must be equal to $do(wait, do(T, S_0))$, respectively. Therefore, formula $emp(sue, 13)[s]$ must be true in all states s that lie between S_0 and $do(T, S_0)$, inclusively. ■

In the previous example, we put constraints on the bounds, to obtain $s_{L_1} = S_0$ and $s_{U_1} = S_{N+1}$. The reason for doing this, instead of leaving the query in the original form (2), is that restrictions RB_i now have all the information about the bounds values, so subqueries $UQ_{F_i}(s_{L_i}, s_{U_1})$ can (and will) be transformed without losing that information. The transformation, that will be described in section 5, can be done automatically, without further intervention of the user.

Example 7 Formula (3) can be rewritten as:

$$\exists\, s_L, s_U \left(\begin{array}{c} \forall\, s(s_L \leq s < s_U \supset \exists\, st\ \ (enrolled(st, c100) \wedge enrolled(st, m100))[s]\ \ ) \\ \wedge\ \ dist(s_U, s_L) = 1 \end{array} \right)$$

It is easy to see that this formula is equivalent to formula (3): Constraint $dist(s_U, s_L) = 1$ simply forces the range of the universal query to be non empty, so there must exist at least one state s such that the formula $(enrolled(st, c100) \wedge enrolled(st, m100))[s]$ is true in it. Remember that we are implicitly assuming that any state term st mentioned (including s_L and s_U) satisfies the condition $S_0 \leq st \leq S_{N+1}$.

Notice that we replaced a existential quantifier over a state by a universal quantifier. The trick here is that the universal quantification affects a variable that is bounded by two variables that are existentially quantified, so we can "simulate" the existential quantifier with a universal quantifier. ■

Next, we give other examples taken from [19], with their translations into the CQF. For clarity, all constraints of the form $dist(S_i, S_j) = 0$ have been rewritten as $S_i = S_j$.

Example 8 We want to know if there is a state such that John is enrolled in courses $c100$ and $m100$. This can be expressed by

$$(\exists\, s) S_0 \leq s \leq do(T, S_0) \wedge enrolled(john, c100, s) \wedge enrolled(john, m100, s),$$

and the corresponding query in the CQF is:

$$\exists\, s_{L_1}, s_{U_1} \left(\begin{array}{c} \forall\, s_1(s_{L_1} \leq s_1 < s_{U_1} \supset (enrolled(john, c100) \wedge enrolled(john, m100))[s_1]) \;\wedge \\ dist(s_{U_1}, s_{L_1}) = 1 \end{array} \right)$$

■

Example 9 *"Is someone enrolled in course* $c200$ *along with some other course in state* $do(T, S_0)$*?"*

$$\begin{array}{l} (\exists\, st, cs, g)(enrolled(st, cs, do(T, S_0)) \wedge enrolled(st, c200, do(T, S_0)) \wedge \\ grade(st, cs, g, do(T, S_0)) \wedge g \geq 50). \end{array}$$

The equivalent formula, with implicit existential quantification over s_{L_1}, s_{U_1}, is

$$\begin{array}{c} \forall\, s_1(s_{L_1} \leq s_1 < s_{U_1} \supset \exists\, st, cs, g \left(\begin{array}{c} enrolled(st, cs) \wedge enrolled(st, c200) \wedge \\ grade(st, c200, g) \wedge g \geq 50 \end{array} \right) [s_1]) \;\wedge \\ (s_{L_1} = do(T, S_0) \wedge s_{U_1} = S_{N+1}) \end{array}$$

■

Notice that the last query is not exactly a historical query, but it is rather a query on a future state $do(T, S_0)$. It is an instance of the *temporal projection problem*, that consists on checking if formula

$$\varphi[do(T, S_0)] \tag{9}$$

holds[3]. Any instance of this problem can be directly translated into the following formula in CQF:

$$\exists\, s_{L_1}, s_{U_1} \left(\begin{array}{c} \forall\, s_1(s_{L_1} \leq s_1 < s_{U_1} \supset \varphi[s_1]) \;\wedge \\ s_{L_1} = do(T, S_0) \wedge s_{U_1} = S_{N+1} \end{array} \right) \tag{10}$$

It is easy to see that formula (10) is equivalent to formula (9): Constraints $s_{L_1} = do(T, S_0)$ and $s_{U_1} = S_{N+1}$ force formula φ to be true in state $do(T, S_0)$. Therefore, answering queries in the CQF subsumes the *temporal projection problem*.

The CQF allows us to represent many kinds of queries. In each universal sub-query, we are expressing that a simple formula, i.e. a static property, holds in all intermediate

[3]According to [14], any instance of this problem can be answered using *regression*. See section 5 for an informal description of this method.

states between the two bounds. Taking this into account, there are some queries that cannot be expressed in the format. For example, a query like $\exists\, x, d\, \forall\, s(S_0 \leq s \leq do(T, S_0) \supset emp(x, d)[s])$ cannot be expressed. This is clear from its syntax. Intuitively, it requires to consider the persistence of an open formula, $emp(x, d)$, in different states. The other case, $\forall\, s(S_0 \leq s \leq do(T, S_0) \supset \exists\, x, d\ emp(x, d)[s])$ can be easily expressed. Notice that in this case, we need to consider if in each intermediate state there is at least one employee, without taking into account if that person is employed in other states.

Despite the previous remark, in sections 4 and 7 we present several useful syntactical classes of queries that can be easily translated into the CQF. In addition, as will be seen in section 7, answering those queries can be done efficiently.

5 A Naïve Algorithm for Answering Queries in the CQF

In this section we will give a general but naïve algorithm for answering historical queries. This algorithm is not efficient, but in the next sections we will show how to improve it. We are given a transaction list T and a query Q, to be answered, written in the form (7). For the rest of this section, the formulas F_i, BC_i, $IR(\ldots)$ mentioned below are supposed to be those of Q. The algorithm consists basically in the following steps (or modules):

1. For every formula F_i, obtain a list of states for which the formula is true. This requires evaluation in all states along T. The outcome of this module is what we call a "map" of the formulas F_i.

2. From the map and the original query, a new query is generated that only mentions equalities and inequalities between pairs of states and between pairs of integers. This new query can be seen as a Constraint Satisfaction Problem (CSP), for which some known algorithms can be applied. There is a "*Yes*" answer to the original query iff the CSP has a solution.

Now we describe the steps in detail. In the first step, we need to know which formulas F_i are true in which states. To do this, we need to evaluate each formula F_i in all states along T. This information is succinctly expressed in what we call a map. For example, suppose that $T = [T_1, \ldots, T_6]$. State S_{N+1} is $S_7 = do(wait, S_6)$ (for simplicity, we use the names $S_1 = do(T_1, S_0)$, $S_2 = do(T_2, do(T_1, S_0))$, …, $S_6 = do(T_6, do(\ldots, do(T_1, S_0))))$. Suppose that we evaluated the formula F in all states along T, finding out that F is true exactly in states S_0, S_1, S_4, S_5 and S_6. The map L of F would be

$$L = [[S_0, S_2), [S_4, S_7)]. \tag{11}$$

If F is not true in any state, then L would be the empty list, $[\,]$. Formally,

Definition 2 *A list $L = [[a_1, b_1), [a_2, b_2), \ldots, [a_k, b_k)]$, with a_j, b_j ground terms of sort state, is a map of a simple formula F, wrt a list of transactions T, iff the following conditions hold:*

1. *For all j, $1 \leq j < k$, the condition $S_0 \leq a_j < b_j < a_{j+1} \leq S_{N+1}$ holds.*

2. *The condition $S_0 \leq a_k < b_k \leq S_{N+1}$ holds.*

3. *$F[s]$ is true in every state s such that $a_j \leq s < b_j$ for some j, with $1 \leq j \leq k$.*

4. *$F[s]$ is not true in any state s such that $S_0 \leq b_j \leq s < a_{j+1} \leq do(T, S_0)$ for some j, $1 \leq j < k$.*

5. *If $b_k < S_{N+1}$, then $F[s]$ is not true in any state s such that $b_k \leq s \leq S_{N+1}$.*

6. *If $F[s]$ is not true in any state, then L is the empty list, $[\,]$.*

This definition readily extends to sets of simple formulas:

Definition 3 *A list $L = [L_1, L_2, \ldots, L_M]$ is a map of the list of formulas $F = [F_1, F_2, \ldots, F_M]$ iff for each i, L_i is a map of the formula F_i.*

The idea is to have in a single map all the information about which formulas F_i were true in which states along T. Assume that, given Q, we somehow obtained a map of its formulas F_i (later in this section we show how to obtain this map).

Notice that if the list $L = [[a_1, b_1), \ldots, [a_m, b_m)]$ is a map for the formula F_i, then the query $(\forall\, s_i) s_{L_i} \leq s_i < s_{U_i} \supset F_i[s_i]$ can be reduced to

$$s_{L_i} \geq s_{U_i} \vee (s_{L_i} < s_{U_i} \;\wedge\; s_{L_i} \geq a_1 \wedge s_{U_i} \leq b_1 \;\vee\; \cdots \;\vee\; s_{L_i} \geq a_m \wedge s_{U_i} \leq b_m). \qquad (12)$$

Formula (12) can be read as: *"There are two cases: either $s_{L_i} \geq s_{U_i}$ or $s_{L_i} < s_{U_i}$. In the second case, the bounds must be such that they are "contained" in some interval $[a_i, b_i)$ of the map L"*. Formula (12) is equivalent to the original query. A generalization of this will be made precise in Theorem 1, later on in this section.

A special case appears when the map of F_i is the empty list, $[\,]$. This means that the formula F_i was not true in any state along T, so the case $s_{L_i} < s_{U_i}$ is not permitted. In this case, the query $(\forall\, s_i) s_{L_i} \leq s_i < s_{U_i} \supset F_i[s_i]$ is reduced to

$$s_{L_i} \geq s_{U_i}. \qquad (13)$$

Notice that the only state variables mentioned in (12) and (13) are s_{L_i} and s_{U_i}. The algorithm that takes the formula $UQ_{F_i}(s_{L_i}, s_{U_i})$ and the map L and yields a formula in the form (12) or (13) is straightforward: If L is the empty list, stop: The resulting formula is (13). If L is not empty, replace the values a_i, b_i of (12) by the values a_i, b_i that appear in L.

For example, the formula $(\forall\, s_1) s_{L_1} \leq s_1 < s_{U_1} \supset F[s_1]$, with the map shown in formula (11), would be reduced to: $s_{L_1} \geq s_{U_1} \vee (s_{L_1} < s_{U_1} \wedge s_{L_1} \geq S_0 \wedge s_{U_1} \leq S_2 \;\vee\; s_{L_1} \geq S_4 \wedge s_{U_1} \leq S_7)$. If the map L were the empty list, then the formula $(\forall\, s_1) s_{L_1} \leq s_1 < s_{U_1} \supset F[s_1]$ would be reduced to $s_{L_1} \geq s_{U_1}$.

Notice that formulas (12) and (13) by themselves can be trivially made true, since the variables s_{L_i}, s_{U_i} are not quantified, so we can always make $s_{L_i} \geq s_{U_i}$. However, we cannot simply replace them by *true*, since other constraints on the bounds may appear later. Recall that the sub-queries $UQ_{F_i}(s_{L_i}, s_{U_i})$ are part of a bigger query that includes additional constraints on bounds.

Now we will present this reduction in formal terms, extending it to a general query in CQF.

Definition 4 *Given a universal query $UQ_{F_i}(s_{L_i}, s_{U_i})$ and a map L_i of the formula F_i, the associated new query $UQ'_{L_i}(s_{L_i}, s_{U_i})$ is defined as follows:*

- *If L_i is the (nonempty) list $[[a_1, b_1), \ldots, [a_m, b_m)]$, then $UQ'_{L_i}(s_{L_i}, s_{U_i})$ is the formula (12).*
- *If L_i is the empty list, then $UQ'_{L_i}(s_{L_i}, s_{U_i})$ is the formula (13).*

Definition 5 *If Q is a query of the form (7) and for each $i = 1, 2, \ldots, M$, the list L_i (possibly empty) is a map for the formula F_i, then the new query Q' is as follows:*

$$\exists\, s_{L_i}, s_{U_1}, \ldots, s_{L_M}, s_{U_M} \left(\begin{array}{c} UQ'_{L_1}(s_{L_1}, s_{U_1}) \wedge \cdots \wedge UQ'_{L_M}(s_{L_M}, s_{U_M}) \wedge \\ (BC_1 \wedge BC_2 \wedge \cdots \wedge BC_r) \wedge \\ IR(S_{N+1}, s_{L_1}, s_{U_1}, \ldots, s_{L_M}, s_{U_M}, \bar{s}_1, \ldots, \bar{s}_c) \end{array} \right) \tag{14}$$

With this definition, we have simply generated a new query $UQ'_{L_i}(s_{L_i}, s_{U_i})$ for each universal query $UQ_{F_i}(s_{L_i}, s_{U_i})$, preserving the constraints. Since in the new query only equalities and inequalities between pairs of states appear, it can be regarded as a Constraint Satisfaction Problem, so that the original query is true iff there are values for the bounds that satisfy the new query.

The correctness of this algorithm is stated in the following theorem[4]:

Theorem 1 *Given an action theory Σ, a query Q of the form (7) to be checked against a legal transaction list T, and a map $L = [L_1, \ldots, L_M]$ of the formulas F_i and the new query Q' of the form (14), it holds $\Sigma \models (Q \equiv Q')$*

Theorem 1 states that answering the original query (considering the entire database specification) is equivalent to checking the new query against Σ.

5.1 On implementation

In the mapping module, we need to obtain a map of the formulas F_i mentioned in the query Q. There are at least two ways to achieve this:

- One way of obtaining a map consists in applying *regression* [13, 14]: If we want to check a formula φ that is simple in the state $do(T, S_0)$, then we can obtain a query φ', simple in the state S_0, that is equivalent to the original query, in the sense that φ holds iff φ' holds [13]. Formula φ' can be checked against the initial database. This

[4]The proofs can be found in [18].

is very useful for checking queries about a single future state, such as in the *temporal projection problem*, shown in section 4. However, the resulting queries can be very long, specially if T is long. This method has been implemented in the system SCDBR, which also allows us to prune the resulting query according to the unique name axioms.

- Another way of getting a map consists in applying *progression* [8, 10], that is, in physically updating the initial database according to the effects of a legal transaction list. A given formula can then be checked against the current (extensional) database. This can be much more efficient than regression [10]. We have implemented progression in SCDBR on the basis of the algorithms described in [10] and the "roll–forward" module of the programming language GOLOG [7], developed by the Cognitive Robotics Group in Toronto.

In [19], Zakinthinos uses progression, because database updating, for his examples, is straightforward and can be done efficiently (some examples were shown in section 3).

We will assume that this module has been implemented using progression, since it allows us to apply some optimization techniques to the algorithm for computing the map (see section 7). Besides, we will assume that we are given the initial database and a legal transaction list to be executed on it. Another approach is to have the current database and the list of transactions in the past. However, it is not always possible, in this case, to generate the tables associated to each state. For example, we may have a transaction that deletes all the entries in a table. If we have the original table, we can perform the deletion, obtaining an empty table. However, if we had the resulting table, we cannot roll–back the transaction, since we do not know which entries were deleted.

Another approach consists in having the current database and the differences of the tables between consecutive states. However, this approach is not in the spirit of this paper, in which we analyze the transaction list, trying to execute as few actions and update as few entries in tables as possible.

This kind of optimization has to do with considering *relevant* entries in tables and *relevant* instances of actions [18]. Usually, an action affects only few entries in few tables. Then, we may assume that only a few actions in a transaction list will be able to change the truth value of the (few) formulas F_i involved in a query.

Having obtained a map of the formulas F_i and generated a new query of the form (14), a constraint checker can be used to answer the new query. For this purpose, we have used *constraint logic programming* (CLP). If we assume that ground state terms are represented by numbers rather than by a list of actions, that is, the state S_0 is represented by the number 0 and the state $do([T_1, \ldots, T_k], S_0)$ is represented by the integer k, then we can obtain a new query that involves only equalities and inequalities between integers.

Given a comparison between $dist(s, s')$ and an integer c, we introduce a new comparison between $s_* - s'_*$ and c, where s_*, s'_* are the integer terms that represent the state terms s, s', respectively. They will be numbers if the original state terms are ground, and will be integer variables if the original state terms are variables. Constraints like $s < s'$, $s \leq s'$ generate the new constraints $s_* < s'_*$, $s_* \leq s'_*$, respectively. For example, the constraint $dist(s, S_0) \geq 1$ yields the new constraint $s_* - 0 \geq 1$. Constraint $s > do([A_1, A_2, A_3], S_0)$ yields the new constraint $s_* > 3$.

In SCDBR we have implemented this checking module by calling clp(fd) [6]. This checking can be done without having to access the physical database. Since the new query only mentions the $=$, $<$, $\leq$, $>$, $\geq$ predicates between integers, all we need is arithmetic. This is consequence of the following theorem, which states the correctness of the algorithm that generates the new query:

Theorem 2 *If $\mathcal{Z}$ is the ordered structure of integer numbers, then, given an action theory Σ, a query Q' of the form (14) and the new, arithmetical query Q'' obtained as described above, then $\Sigma \models Q'$ iff $\mathcal{Z} \models Q''$.*

From Theorems 1 and 2, we can see that the problem of checking a query Q in the CQF can be reduced to checking constraints on integers. Given the map and the numerical constraints, this checking can be very fast. However, obtaining the map can be very expensive, since it requires checking all the formulas F_i in all states along T.

5.2 Can we improve the algorithm?

Now we will outline several sources of optimization of the naïve algorithm we presented before. They aim to minimize the cost of the first step of the naïve algorithm, namely, the construction of the map. In following sections we will apply most of the ideas described in the rest of this section.

(1) We should execute only those actions that can affect the truth value of the fluents involved in the query and only update some entries in some tables. In sections 6 and 7 we present a general algorithm that determines which entries in which tables must be updated, and what actions must be executed. There we will use the concept of *relevant* actions and fluents.
(2) A smart algorithm should stop as soon as the query can be satisfied: It should not be necessary to evaluate formulas in all states along T. For example, query (3) is satisfied as soon as we find a state s such that formula $(enrolled(st, c100) \land enrolled(st, m100))[s]$ is true in s, so no further evaluation is required in states which are reachable from s.
(3) A smart algorithm should stop as soon as the query cannot be satisfied. For example, evaluation in query (2) should stop as soon as a state that does not satisfy $emp(Sue, 13)$ is found.
(4) There are cases where sub-queries have to be evaluated only at a part of T, for example, in the *temporal projection problem*, so, truth evaluations of formulas at other states should not be performed. The database could still be updated in other states, but skipping the evaluation part.

All these ideas are incorporated in the algorithms presented in section 7.

6 Computing Relevance

In section 5 we presented a naïve algorithm for answering historical queries of the form (7) against a transaction list T. In the first step of that algorithm, all actions in T are executed, computing all of the effects. However, an action affects a few entries in a few tables in general.

We might assume that, given T, only a few actions will be able to change the truth value of the formulas F_i involved in the query, since only a few tuples appear in each formula F_i.

The question that naturally arises is "How to characterize those *relevant* actions?". A related question asks for the minimum set of tuples that must be updated to answer the query. Notice that the interest is in determining what actions and tuples in the database are *irrelevant*, since we will not have to compute their effects and store them, respectively.

Intuitively, a tuple $F'(\bar{a})$ will be relevant to a tuple $F(\bar{b})$ if a change in the truth value of $F'(\bar{a})$ (at some state s) can possibly change the truth value of $F(\bar{b})$ at some state equal to or reachable from s. (Notice that we are calling *tuple* an entry in a table. By extension, we will call $F(x)$ a tuple, even when x is a variable.) A tuple will always be relevant to itself, for the obvious reason that a change of the truth value of $F(\bar{b})$ in a state changes the truth value of $F(\bar{b})$ in that same state.

An action $A(\bar{a})$ will be relevant to a tuple $F(\bar{b})$ if there is a state s such that the truth value of $F'(\bar{c})[s]$ differs from the truth value of $F'(\bar{c})[do(A(\bar{a}, s))]$, with $F'(\bar{c})$ being a tuple that is relevant to $F(\bar{b})$. Therefore, to check the truth value of $F(\bar{b})$ along T, we must compute the effects of that relevant action $A(\bar{a})$, if $A(\bar{a})$ appears in T. On the other hand, if $A(\bar{a})$ is irrelevant to $F(\bar{b})$ and $A(\bar{a})$ appears in T, its effects can be ignored.

In [17], we have given a precise *semantical* characterization of this intuitive notion of relevance. In this paper we will provide a *procedural* notion of relevance, more precisely, we will introduce an algorithm that, given a tuple $F(\bar{c})$, yields a set of tuples and a set of parameterized actions that are relevant to $F(\bar{c})$. Having such algorithm, in order to answer an historical query Q in the CQF, we follow these steps:

1. Keep in L all the tuples appearing in Q.
2. Compute R, the set of relevant tuples to the tuples in L.
3. Having the relevant tuples, compute the set of relevant actions RA.
4. Finally, evaluate the query, executing only the actions that are in RA and updating only the tuples that are in R.

We will illustrate the algorithm for obtaining the relevant tuples and actions by means of an example:

Example 10 Suppose that we have the following specification, which is a more complex version of the employee database presented in section 3. We do not include the precondition axioms, since they will be not important for our purposes of computing relevance (it is also possible to think of stating $Poss(a, s) \equiv true$). We have the following SSAs:

$$\begin{aligned} Poss(a,s) \supset emp(x,d)[do(a,s)] \equiv\ & a = hire(x,d) \vee emp(x,d)[s] \wedge \\ & \neg\, \exists y (a = fire(y,x) \wedge emp(y,d) \wedge boss(y)[s]). \\ Poss(a,s) \supset boss(x)[do(a,s)] \equiv\ & a = promote(x) \vee boss(x)[s] \wedge \neg\, a = quit(x). \\ Poss(a,s) \supset salary(x,\$)[do(a,s)] \equiv\ & a = change\_salary(x,\$) \vee salary(x,\$)[s] \wedge \\ & \neg\, \exists \$' (a = change\_salary(x,\$') \wedge \$' \neq \$). \end{aligned}$$

Action $\mathit{fire}(y, x)$ is understood as y firing x. This would make x unemployed iff y is a boss employed in the same department of x.

We want to know if the formula (8), asking if Sue has always been employed in dept. 13, holds. The only tuple appearing in the query is $emp(sue, 13)$, so we must obtain the tuples and actions that are relevant to $emp(sue, 13)$. We will obtain the relevant tuples and actions for a generic tuple $emp(x, d)$, then we will obtain the relevant tuples and actions for $emp(sue, 13)$ by instantiation of variables.

From the SSAs we can see that, in order to check if $emp(x, d)[s]$ holds in any state s, we have to check $emp(y, d)$ and $boss(y)$ (for any value of y, in particular for $y = x$), since the truth value of $emp(x, d)$ depends on the truth values of these tuples: Anyone employed in department d can be promoted to a boss and any boss can fire x. The way we obtain these *relevant tuples* is by recursively looking at the formulas $\Phi_{F(\bar{c},\bar{x})}$ appearing on the right–hand sides of the SSAs[5]. We will not look at the tuples appearing in the (omitted) action precondition axioms. Remember that we are checking the query against a legal transaction list T.

If we want to keep updated the tuple $emp(x, d)$, we start by adding it to the list of relevant tuples. We will regard x and d as constants, since they stand for a specific entry in table *emp*.

Next, we look at the tuples appearing in $\Phi_{emp(x,d)}$ (see section 2 for a description of formulas $\Phi_{F(\bar{c},\bar{x})}$), that is, to the tuples in

$$a = hire(x, d) \vee emp(x, d)[s] \wedge \neg \exists y(a = \mathit{fire}(y, x) \wedge emp(y, d) \wedge boss(y)[s]),$$

obtaining the tuples $emp(y, d)$ and $boss(y)$ (notice that we are not considering state arguments in the tuples). We will replace the variable y by an *anonymous variable*, $\_$. Later on in this section we will discuss this renaming. We delete $emp(x, d)$ from the set of relevant tuples, but add $emp(\_, d)$ to the list, since $emp(x, d)$ represents a single entry in *emp*, while $emp(\_, d)$ represents several entries, which include $emp(x, d)$. We also add $boss(\_)$.

Now, we look at the formula $\Phi_{boss(\_)}$, that is, $a = promote(\_) \vee boss(\_)[s] \wedge \neg\, a = quit(\_)$, obtaining the tuple $boss(\_)$. We already have it in the set of relevant tuples, so we do not add it again. Now we look at the SSA of $emp(\_, d)$, that is, at

$$a = hire(\_, d) \vee emp(\_, d)[s] \wedge \neg \exists y(a = \mathit{fire}(y, \_) \wedge emp(y, d) \wedge boss(y)[s]).$$

The tuples appearing there are $emp(\_, d)$, $emp(y, d)$ and $boss(y)$. We rename y to $\_$, finding out that all of these tuples are already in the set of relevant tuples, so we do not add them. Since there are no more SSAs we stop, obtaining that the set of tuples that are relevant to $emp(x, d)$ is $\{emp(\_, d), boss(\_)\}$, which is the intuitively expected result.

The set of tuples relevant to a specific entry in table *emp* is obtained by variable instantiation. For example, the set of relevant tuples to $emp(Sue, 13)$ is obtained by the substitution $\sigma = \{x/Sue, d/13\}$. This gives us the set of relevant tuples $\{emp(\_, 13), boss(\_)\}$. In general, an auxiliary table may be necessary, actually only partially instantiated.

[5]We use the notation $F(\bar{c}, \bar{x})$ to refer to an entry in table F with some arguments instantiated (the $\bar{c}$ part) and some not instantiated (the $\bar{x}$ part). They are not necessarily in any order and any part can omitted.

The actions relevant to an arbitrary tuple are obtained by applying a similar method: Since $emp(\_, d)$ and $boss(\_)$ are the only tuples we have to update, we look at the actions appearing in the SSAs of $emp(\_, d)$ and $boss(\_)$. This gives us the set $\{\mathit{fire}(\_, \_), \mathit{hire}(\_, d), \mathit{quit}(\_), \mathit{promote}(\_)\}$. Therefore, in order to check the truth value of $emp(Sue, 13)$, we have to compute the effects of the actions $\{\mathit{fire}(\_, \_), \mathit{hire}(\_, 13), \mathit{quit}(\_), \mathit{promote}(\_)\}$, if they appear in T.

For example, if $T = [\mathit{hire}(sue, 13), \mathit{promote}(john), \mathit{fire}(john, sue), \mathit{change\_salary}(john, 15.000), \mathit{hire}(peter, 9)]$, the only relevant actions are $\mathit{hire}(sue, 13)$, $\mathit{promote}(john)$ and $\mathit{fire}(john, sue)$, since they are the ones that are *particular cases* of elements of the set $\{\mathit{fire}(\_, \_), \mathit{hire}(Sue, 13), \mathit{quit}(\_), \mathit{promote}(\_)\}$. The effects of the other actions can be discarded. This concept of particular case will be made precise below. ■

Notice that there can exist circularity in the SSAs: The SSA of $F(\bar{x})$ can mention $F'(\bar{x}')$ and vice-versa. In fact, if we use Reiter's specification, with formulas of the form (1), then $\Phi_{F(\bar{x})}$ always mentions $F(\bar{x})$. This could make the recursive search go into an infinite loop. However, our algorithm will always stop, because it keeps only one version of the "most general" tuples, without state arguments. To define precisely this idea of generality, we need the notion of *subsumption*:

Definition 6 *1. An expression E' is subsumed by another expression E if there is a variable substitution σ such that $E\sigma = E'$, i.e. E is* more general *than E'.*

2. An expression E' is subsumed by a set of expressions L if there is a expression E in L and variable substitution σ such that $E\sigma = E'$.

3. A set of expressions L' is subsumed by another set of expressions L if each element of L' is subsumed by L.

Notice that an infinite set of expressions can be subsumed by a finite set of expressions. Indeed, the infinite set $L' = \{f(1), f(2), f(3), \ldots\}$ is subsumed by the set $L = \{f(x)\}$, with x variable, since each element of L' is a particular case of $f(x)$.

For example, $\mathit{promote}(john)$ is subsumed by $\mathit{promote}(y)$, since there is a variable substitution, namely $\sigma = \{y/john\}$, such that $\mathit{promote}(john) = \mathit{promote}(y)\sigma$. Therefore, the set of relevant actions for $emp(sue, 13)$ subsumes $\mathit{promote}(john)$, since it contains $\mathit{promote}(y)$.

In the algorithm for computing relevance, we will consider a special variable substitution, σ^*, that changes every variable by a new variable. This is like standardizing variables apart, except for the fact that two occurrences of the same variable are renamed to different variables. We will rename all the variables to a single special variable, denoted by an underscore $(\_)$, that can take different values in the same expression. This way, the expression $E(x, y)$ is subsumed by the expression $E(\_, \_)$. This special variable is similar to the *anonymous variable* used in PROLOG. This renaming will guarantee that the algorithm for computing relevance will stop.

The substitution σ^* makes variables loose their identity: For example, if we have a set of expressions $\{E(x), E'(x)\}$ and we apply σ^* to each expression in the set, we obtain $\{E(\_), E'(\_)\}$, which does not convey the information that the variables in both expressions are the same. However, this does not matter for determining relevant tuples and actions.

Prior to the application of the algorithm, all of the variables appearing in $F(\bar{c}, \bar{x})$ must be replaced by new constants, so that they are not affected by the variable substitution σ^*. After the algorithm has stopped, the new constants introduced are replaced by the original variables.

Now, we present in precise terms the general algorithm for computing CRT, the set of relevant tuples (CRT stands for *computationally relevant tuples*)[6]. In this algorithm we use an operator O. Given a set of tuples S, $O(S)$ is obtained in a two–step process: First, the auxiliary set $AuxO(S)$ is generated by collecting in it all the tuples mentioned in the SSAs of the tuples in S that are not subsumed by S.[7] Then, the set $O(S)$ is generated by collecting in it all the tuples in $AuxO(S)$ and all the tuples in S that are not subsumed by $AuxO(S)$. Formally,

Definition 7 *Given a set of tuples S, $O(S)$ is defined by:*

$$\begin{aligned} AuxO(S) = \ & \{F'(\bar{c}', \bar{x}')\sigma^* /\ \textit{there is } F''(\bar{c}'', \bar{x}'') \in S \textit{ and } F'(\bar{c}', \bar{x}') \textit{ is mentioned in } \Phi_{F''(\bar{c}'', \bar{x}'')} \\ & \textit{and } F'(\bar{c}', \bar{x}') \textit{ is not subsumed by } S\}. \\ O(S) = AuxO(S) \cup \ & \{F'(\bar{c}', \bar{x}') / F'(\bar{c}', \bar{x}') \in S \textit{ and} \\ & F'(\bar{c}', \bar{x}') \textit{ is not subsumed by } AuxO(S)\}. \end{aligned}$$

Example 11 Consider the specification shown in Example 10 in this section. Suppose that $S = \{emp(sue, 13)\}$. The tuples appearing in the SSA of $emp(sue, 13)$ are $emp(y, 13)$ and $boss(y)$. Notice that the tuples do not contain the state argument. Since y is a variable, it is changed to $\_$. Therefore,

$$AuxO(\{emp(sue, 13)\}) = \{emp(\_, 13), boss(\_)\}.$$

Both tuples are collected, since they are not subsumed by $emp(sue, 13)$. Now, we have that

$$O(\{emp(sue, 13)\}) = \{emp(\_, 13), boss(\_)\}.$$

The tuple $emp(sue, 13)$ has not been collected, because it is subsumed by $emp(\_, 13)$. ■

The previous example shows that $O(S)$ always subsumes S. This is clear, since we are keeping the most general versions of the tuples.

We will need to apply the operator O several times to a given set of tuples. We will denote by $O^i(S)$ the result of applying the operator i times on the set of tuples S. In particular, we will compute $O^i(\{F(\bar{c}, \bar{x})\})$, for a tuple $F(\bar{c}, \bar{x})$, until all the tuples are collected, that is, until the operator O reaches a fix-point.

An important property of the operator O is that it reaches a fix-point. This is guaranteed by the following theorem

Theorem 3 *Given a specification without function symbols on non–state variables appearing in the tuples in the SSAs, for any tuple $F(\bar{c}, \bar{x})$, there is an integer number P such that $O^P(\{F(\bar{c}, \bar{x})\}) = O^{P-1}(\{F(\bar{c}, \bar{x})\})$.*

[6]We do not consider here the optimization based on relevance to the algorithm for answering queries; this will be done in section 7.

[7]The fluents can be linearly collected: If the SSA is $F(x) \wedge x = a$, we collect $F(x)$, not $F(a)$.

Definition 8 *$CRT(F(\bar{c}, \bar{x}))$, the set of tuples that are relevant to the tuple $F(\bar{c}, \bar{x})$, is $O^P(\{F(\bar{c}, \bar{x})\})$, the fix-point of operator O applied to $\{F(\bar{c}, \bar{x})\}$.*

Example 12 We will compute the fix-point of $O(\{emp(sue, 13)\})$. In Example 11, we saw that the first iteration yields $\{emp(\_, 13), boss(\_)\}$. For the second iteration, the tuples appearing on the right–hand sides of the SSA for $emp(\_, 13)$ are $emp(\_, 13)$, $emp(y, 13)$ and $boss(y)$. Since y is a variable, it is changed to $\_$. The only tuple appearing in the SSA of $boss(\_)$ is $boss(\_)$. All these tuples are subsumed by the result obtained in first iteration. Thus, we have:

$$AuxO(\{emp(\_, 13), boss(\_)\}) = \varnothing,$$
$$O(\{emp(\_, 13), boss(\_)\}) = \{emp(\_, 13), boss(\_)\}.$$

Since we obtained the same as in the first iteration, the fix-point of $O(\{emp(sue, 13)\})$ is the set $\{emp(\_, 13), boss(\_)\}$. ■

Since the operator O always reaches a fix-point after a finite number of steps, we can always compute CRT in finite time. However, this is not necessarily true if we allow function symbols in the tuples. Consider the following example:

Example 13 Suppose we have the following SSA, in which there is a function symbol in a fluent:

$$Poss(a, s) \supset F(n)[do(a, s)] \equiv F(n + 2)[s] \wedge a = A \vee F(n)[s]. \tag{15}$$

If we we want to compute the tuples that are relevant to $F(0)$. We obtain

$$\begin{aligned} &O^0(\{F(0)\}) = \{F(0)\}, \\ &O^1(\{F(0)\}) = \{F(0), F(2)\}, \\ &\dots \\ &O^n(\{F(0)\}) = \{F(0), F(2), \dots, F(2 * n)\}, \\ &\dots \end{aligned}$$

The operator does not reach a finite fix-point. The problem comes from the combination of two things: First, the specification is circular, in the sense that an instance of F is relevant to another instance of F. Second, the operator always contains ground tuples and new constants are introduced at each step, which are not subsumed by the result of the previous step. This makes the algorithm go into an infinite loop.

This problem can be satisfactorily solved with a slight modification of (15), consisting in replacing the function symbols in the fluents by introducing new variables: $F(f(x))$ is equivalent to $(\exists\, x')F(x') \wedge x' = f(x)$. With the following SSA, that is equivalent to (15):

$$Poss(a, s) \supset F(n)[do(a, s)] \equiv a = A \wedge (\exists\, n')(F(n')[s] \wedge n' = n + 2) \vee F(n)[s], \tag{16}$$

the algorithm yields the fix-point $\{F(\_)\}$:

$$\begin{aligned} &O^0(\{F(0)\}) = \{F(0)\}, \\ &O^1(\{F(0)\}) = \{F(\_)\}, \\ &O^2(\{F(0)\}) = \{F(\_)\} = O^1(\{F(0)\}). \end{aligned}$$

Notice that (16) does satisfy the hypothesis of Theorem 3, of not allowing function symbols over non–state terms in the fluents. ■

We have to pay a price for the guaranteed existence of the finite fix-point of O: We do not always obtain a precise characterization of the intuitively relevant tuples. In the previous example, the only intuitively relevant tuples to $F(0)$ are of the form $F(2*k)$, with k positive integer, but the fix-point $\{F(\_)\}$ subsumes any entry in table F, in particular, the "irrelevant" tuples $F(1)$, $F(3)$, $F(5)$, $F(7)$, etc.

In [18, 17], we model the intuitively relevant tuples by means of a semantic notion of relevance, and we show that a perfect computational characterization of the (semantically) relevant tuples is not always possible. We also prove there that the output of our algorithm for computing relevant tuples subsumes all the semantically relevant tuples. Therefore, any tuple that cannot be obtained by instantiating the variables of some element of $CRT(F(\bar{c},\bar{x}))$ is (semantically) irrelevant to $F(\bar{c},\bar{x})$. We will be (possibly) updating tuples that are not relevant, but no relevant tuple will be missed in the database update.

Having defined the computational notion of relevance of tuples, let us define the set of computationally *relevant actions* for a tuple $F(\bar{c},\bar{x})$. Basically, having obtained $CRT(F(\bar{c},\bar{x}))$, we directly collect all the actions appearing in the SSAs of the tuples in $CRT(F(\bar{c},\bar{x}))$ (compare with Example 10). So, no recursion is necessary.

Definition 9 *$CRA(F(\bar{c},\bar{x}))$, the set of actions that are computationally relevant to the tuple $F(\bar{c},\bar{x})$, is defined by:*

$$\begin{array}{rl} CRA(F(\bar{c},\bar{x})) = & \{A(\bar{c}',\bar{x}')/ \textit{ there is } F''(\bar{c}'',\bar{x}'') \in CRT(F(\bar{c},\bar{x})) \textit{ and} \\ & A(\bar{c}',\bar{x}') \textit{ is mentioned in } \Phi_{F''(\bar{c}'',\bar{x}'')} \textit{ and} \\ & A(\bar{c}',\bar{x}') \textit{ is not a variable}\}. \end{array}$$

It is not always the case that the intuitively relevant actions to $F(\bar{c},\bar{x})$ (see the beginning of this section) are instances of elements of $CRA(F(\bar{c},\bar{x}))$. In [18] we present an example of SSAs for which this happens. However, it is also shown in [18], that there is one important syntactical class of SSAs for which $CRA(F(\bar{c},\bar{x}))$ does subsume all the (semantically) relevant actions, namely, all the specifications derived from effect axioms.

Now, we will give a complexity analysis. The cost of computing relevance can be as high as $O(N_F \times ((N_C + N_A + 1)^{N_A}))$, with N_F the (small) number of fluents (tables), N_A the (small) maximum arity of a fluent and N_C the number of constants in the specification. This happens when we compute the tuples that are relevant to a tuple that contains only variables (thus adding N_A new constants to the specification) and we recursively add all the combinations of fluents, constants and positions in the arguments of the fluents. Notice that this recursive process can be done only once, offline, because we can compute the relevant tuples and actions for generic tuples, and then obtain the relevant things for specific tuples by instantiation of variables.

In many cases, including all those in [19, 13] and most of the examples processed with SCDBR, the algorithm needs only one iteration. In other examples, like the ones presented in [2] and Example 10, the algorithm needs two iterations.

7 Optimizing the Naïve Algorithm

In this section we present some optimizations of the naïve algorithm described in section 5 for answering historical queries in the CQF. First, we show an improvement that uses the set of relevant tuples and actions, presented in section 6. Next, we present some optimizations that use the idea of not processing the entire transaction list, but a minimum initial segment (see section 5). These last algorithms are very optimized, but they apply to some particular kinds of queries.

7.1 Optimizing by relevance

Given a historical query and a transaction list T of length N, the following algorithm uses relevance for computing a map of the queries:

Algorithm 1

1. Generate the set L of tuples mentioned in the query. This is done by collecting the tuples appearing in the formulas F_i of the query.

2. Compute sets CRT_L, CRA_L: For each tuple $F(\bar{c}, \bar{x}) \in L$, compute $CRT(F(\bar{c}, \bar{x}))$ and the set $CRA(F(\bar{c}, \bar{x}))$ as described in section 6. Then

$$\begin{array}{l} CRT_L = \bigcup_{F(\bar{c},\bar{x}) \in L} CRT(F(\bar{c}, \bar{x})). \\ CRA_L = \bigcup_{F(\bar{c},\bar{x}) \in L} CRA(F(\bar{c}, \bar{x})). \end{array}$$

3. Get the map M of the queries in the following way:

   3.0. Make $j = 0$. For all i make $L_i = []$, $status_i = false$.

   3.1. For all i do: Check F_i in S_0. If it is true, make $status_i = true$ and open an interval in list L_i at state S_0.

   3.2. Make $j = j + 1$. If $j = N + 1$, go to step 3.6.

   3.3. If action a_j is not subsumed by CRA_L, go to step 3.2. Else, update the tuples subsumed by CRT_L according to a_j.

   3.4. For all i do: Check formula F_i against the current database. If $status_i = true$ and F_i is *false* in current state, close the interval in list L_i at state S_j and make $status_i = false$. Otherwise, if $status_i = false$ and F_i is *true* in current state, open an interval in list L_i at state S_j and make $status_i = true$.

   3.5. Go to step 3.2.

   3.6. Close all the intervals at state S_j. Each list L_i is a map of the formula F_i. Go to step 4.

4. Given the map $M = [L_1, \ldots, L_M]$, obtain a equivalent CSP as described in section 5 and check if it can be satisfied. If it is the case, answer *Yes*. Otherwise, answer *No*.

■

This algorithm can be applied to any query in the CQF and with any specification given in terms of SSAs. This is a kind of *semantic* query optimization, since it uses the information contained in the SSAs, which describe the unique semantics of the database updates.

Example 14 Consider the specification of Example 10. Suppose that the initial database is $emp(john, 13)$ and we want to obtain a map of formula $F_1 = emp(Sue, 13)$ against the transaction list $T = [hire(sue, 13),\ promote(john),\ fire(john, sue),\ change\_salary(john, 15.000), hire(peter, 9)]$.

We will generate the map of $emp(sue, 13)$ with the algorithm given above: First, the only relevant tuples to $emp(sue, 13)$ are $emp(\_, 13)$ and $boss(\_)$. The only relevant actions to $emp(Sue, 13)$ are $hire(sue, 13)$, $promote(john)$ and $fire(john, sue)$. We make $j = 0$, $status_1 = false$, $L_1 = [\,]$. We make $j = 1$ and execute $hire(sue, 13)$, adding $emp(sue, 13)$ to the current database (which is subsumed by $emp(\_, 13)$). Since F_1 is true in this state and $status_1 = false$, we open an interval in L_1, which becomes $[[S_1,\quad)]$. We make $status_1 = true$. We make $j = 2$ and execute $promote(john)$, thus adding $boss(john)$ (which is subsumed by $boss(\_)$) to the current database. In step 4. we do not change L_1, since both $status_1$ and F_1 are true. We make $j = 3$ and execute $fire(john, sue)$, deleting the relevant tuple $emp(sue, 13)$. Since $status_1 = true$ and F_1 is false, we close the interval of L_1 at state S_3, so the map becomes $[[S_1, S_3)]$. We make $status_1 = false$. We make $j = 4$ and skip the irrelevant action $change\_salary(john, 15.000)$. We make $j = 5$ and skip the irrelevant action $hire(peter, 9)$. We make $j = 6$, so we go to step 6. Since the last interval of L_1 has a second component, closing the intervals has no effect, so we stop: The map of F_1 is $[[S_1, S_3)]$, the same appearing in Example 10. ■

The algorithms presented in this paper are implemented in the system SCDBR: It can physically update the current database according to the effects of a legal action. Given a tuple, it can determine what tuples and actions are relevant to it. SCDBR determines what are the relevant tuples and actions for generic tuples only once, right after the user loads the specification. The tuples and actions that are relevant to an instantiated tuple are obtained by variable instantiation, what is very fast. These "subroutines" are used to answer historical queries written in the CQF.

To express the historical queries in the CQF, the user must introduce a list of universal queries and a list representing the constraints BC_is. Each universal query is represented by a list of three elements: The lower bound, the upper bound and the formula F_i associated to the sub-query. Each constraint BC_i is of the form $dist(S_k, S_i)\ op\ C$, where S_k, S_i are either integer constants or bounds appearing in the universal sub-queries.

The mapping module is an implementation of the Algorithm 1. Irrelevant actions are replaced by the action `wait`, prior to the execution of this module, which has no effects.

Now we give a simple worst–case complexity analysis for answering queries in the CQF. We will only consider the cost of obtaining the map of the formulas F_i. Stating and solving the CSP does not access the physical database.

The algorithm presented in this section executes all the transactions and update all the tuples in the worst case (when every action and every tuple is relevant to the sub-queries). Therefore, if a query has M sub-queries to be checked against a transaction list T of length N, it will make $O(N \times M \times P(S))$ operations in the worst case, where $P(S)$ is the cost of progressing the database according to a given action, being S a measure of the size of the database. As the number of sub-queries is small, the complexity of this algorithm is practically $O(N \times P(S))$.

7.2 Some special cases

Now, we present three specializations of the previous algorithm that introduce further optimizations for processing queries in special syntactical forms. None of them needs to apply the last step of the naïve algorithm, namely, the constraint satisfaction. This is because it is clear, early in the execution, if the constraint satisfaction problem will have a solution or not, and at that point an answer to the query can be given right away.

1. We want to know if a given formula φ holds in all intermediate states between S_0 and $do(T, S_0)$, inclusively. This can be represented in the CQF by

$$\exists\, s_{L_1}, s_{U_1} \left(\begin{array}{c} \forall\, s_1\ (s_{L_1} \leq s_1 < s_{U_1} \supset \varphi[s_1]) \wedge \\ dist(s_{L_1}, S_0) = 0 \wedge dist(s_{U_1}, s_{N+1}) = 0 \end{array} \right)$$

   The mapping module can stop as soon as it finds a state s such that $\varphi[s]$ is not true. This is accomplished with the following algorithm[8]:

```
IF φ holds in S0 THEN
  FOR each action a in T DO
    IF a is relevant to the query THEN
      update the relevant tuples according to a
      IF φ does not hold in current state THEN
        RETURN false;
  RETURN true;
ELSE
  RETURN false;
```

2. Another interesting case is to check if a formula φ holds in some intermediate state between S_0 and $do(T, S_0)$, inclusively. This is represented in the CQF by

$$\exists\, s_{L_1}, s_{U_1}\, \forall\, s_1\ \ s_{L_1} \leq s_1 < s_{U_1} \supset \varphi[s_1] \wedge dist(s_{U_1}, s_{L_1}) = 1\ .$$

   The idea here is to stop as soon as we find a state that satisfies φ:

```
IF φ holds in S0 THEN
  RETURN true;
ELSE
  FOR each action a in T DO
    IF a is relevant to the query THEN
      update the relevant tuples according to a
```

[8]In the following, the scope of the instructions `IF` and `FOR` is determined by the indentation level: All instructions that are "inside" these instructions are indented wrt the `IF` or `FOR` statement.

```
    IF φ holds in current state THEN
      RETURN true;
RETURN false;
```

3. If $\varphi[do(T, S_0)]$ is an instance of the *temporal projection problem* (see section 5), then it can be answered using the following algorithm:

```
FOR each action a in T
  IF a is relevant to the query THEN
    update the relevant tuples according to a
IF φ holds in current state THEN
  RETURN true;
ELSE
  RETURN false;
```

   The optimization in this algorithm comes from not checking the formula φ in intermediate states.

The examples shown above are interesting special cases of historical queries. However, the algorithms for answering them are not easily extendible to more complex queries.

8 Conclusions and Discussion

(1) Using the situation calculus as underlying the formalism, we have presented a Canonical Query Format that allows us expressing historical queries. We presented a naïve algorithm for answering any query in the CQF. Next we optimized the algorithm by introducing computational notions of relevance of tuples and actions. We also presented some special cases of historical queries that can be easily answered.
(2) In [19], Zakinthinos relies on the syntactical form of the SSAs, which are *context free* [10]. Formula (4) is context free: The truth value of $emp(x, d)$ does not depend on other tuples, but only on the actions to be executed. In the cases he presents, progression can be done in $O(1)$, so the algorithms are linear in the length of T in the worst case. His underlying idea is to make as much list processing as possible, while we always make database updating. However, the special algorithms presented in section 7.2 coincide with his algorithms in the examples he shows ([18]). We do not rely on the syntactical form of the specification, so our specializations are more general.
(3) In [18, 17], *semantical* notions of relevance of tuples and actions are introduced, and the soundness of the algorithm based on the computational notion of relevance is established, by proving the completeness of computational relevance wrt semantical relevance. More precisely, the set of semantically relevant tuples is subsumed by the set of computationally relevant tuples. The same holds for actions when the SSAs are derived from effect axioms. So, the algorithm does not miss any (semantically) relevant update. It might be the case that it declares as relevant tuples and/or actions that are not, but this is not going to produce incorrect results. There are cases where both notions coincide, e.g. for context-free SSAs [10].

(4) A work very much in the spirit of ours is presented by Chomicki in [5]. In the framework of *temporal logic*, he addresses the problem of checking if an integrity constraint (IC) is satisfied in an interval of the evolution of the database. He tries to check the IC updating as little information as possible (this is called "bounded history encoding"), which is a similar basic idea to the one we developed in this paper. Having the database in a current state that satisfies an IC and an update leading to a new state, he checks if the new state satisfies the IC. If this is the case, the update is accepted and the new state becomes the current state. Otherwise, the database remains untouched and the update is rejected.

In our case, having the database in a state s such that a formula φ is true in it and an action A, we wish to check if φ is true in the state $do(A, s)$. In contrast with Chomicki, we can tell in advance if A will affect formula φ. If it does not affect φ, that is, if A is irrelevant to φ, then we can safely skip it, without computing its effects, and without having to check φ in the "new" state. Only if A is relevant to φ, we execute it and we check φ in the new state. In Chomicki's case, updates always have to be performed. Another difference with our work presented is that [5] does not introduce any explicit notion of relevance.

(5) In [4], a notion of *irrelevance* is introduced in the context of updates of relational databases. An update in a relational base table is irrelevant to a view (obtained by applying select, project and join operations from relational algebra to a set of base relations), if it does not produce a change in that view. They give necessary and sufficient conditions for deciding if an update is irrelevant to a view, which can be efficiently checked in some cases. Although we have not worked out the details, we believe that, reconstructing their approach in a situation calculus based setting, their relevant updates would be particular cases of our relevant actions.

Acknowledgments

We wish to thank Marcelo Arenas, Pablo Sáez, Cristian Ferretti, Mauricio Strello and Richard Scherl for stimulating conversations, advice and help. Leopoldo Bertossi visited Toronto on a couple of occasions and is grateful to Ray Reiter, Daniel Marcu, Fangzhen Lin and Aris Zakinthinos for stimulating conversations and relevant information. This research has been supported partly by DIUC, and FONDECYT (#193–0554) grants.

References

[1] L. Bertossi, M. Arenas, and C. Ferretti. "SCDBR: An Automated Reasoner for Database Updates (Extended Abstract)". In *Proc. VXI International Conference of the Chilean Computer Science Society (SCCC'96), Valdivia, Nov. 13–15*, 1996.

[2] L. Bertossi, M. Arenas, and C. Ferretti. "SCDBR: An Automated Reasoner for Specifications of Database Updates". *To be submitted for publication*, 1996.

[3] L. Bertossi and C. Ferretti. "SCDBR: A Reasoner for Specifications in the Situation Calculus of Database Updates". In *Temporal Logic. Proceedings First International Conference, ICTL '94, Bonn, Germany, July 1994, LNAI 827*. Springer.

[4] J. Blakeley, N. Coburn, and P. Larson. "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates". *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[5] J. Chomicki. "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding". *ACM Transactions on Database Systems*, 20(2):149–186, June 1995.

[6] D. Díaz. *"clp(FD) 2.21 User's Manual"*. INRIA–Rocquencourt, France, 1994.

[7] Y. Lésperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. "A Logical Approach to High–Level Robot Programming – A Progress Report". In *Proc. AAAI Fall Symposium of Control of the Physical World by Intelligent Systems*, New Orleans, LA, November 1994.

[8] F. Lin and R. Reiter. "How to Progress a Database (and Why) I: Logical Foundations". In E. Sandewall J.Doyle and P. Torasso, editors, *Proc. Fourth Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 425–436. Morgan Kaufmann, 1994.

[9] F. Lin and R. Reiter. "State Constraints Revisited". *Journal of Logic and Computation. Special Issue on Actions and Processes*, 4(5):655–678, 1994.

[10] F. Lin and R. Reiter. "How to Progress a Database II: The STRIPS Connection". In *Proc. IJCAI'95*, pages 2001–2007, 1995.

[11] J. McCarthy and P. Hayes. "Some Philosophical Problems from the Standpoint of Artificial Intelligence". In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502, Edinburgh, Scotland, 1969. Edinburgh University Press.

[12] Javier Pinto. *"Temporal Reasoning in the Situational Calculus"*. PhD thesis, Department of Computer Science, University of Toronto, 1994.

[13] R. Reiter. "The Frame Problem in the Situation Calculus: a Simple Solution (Sometimes) and a Completeness Result for Goal Regression". In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[14] R. Reiter. "On Specifying Database Updates". Technical Report KRR-TR-92-3, University of Toronto, Department of Computer Science, Toronto, Canada, 1992.

[15] R. Reiter. "Proving Properties of States in the Situation Calculus". *Artificial Intelligence*, 64(2):337–351, 1993.

[16] R. Reiter. "On Specifying Database Updates". *Journal of Logic Programming*, 25(1):53–91, 1995.

[17] B. Siu and L. Bertossi. "A Semantical Notion of Relevance in Specifications of Database Updates". *To be submitted for publication*, 1996.

[18] B. Siu and L. Bertossi. "Answering Historical Queries in Databases". Technical Report RT-PUC-DCC-2-96, Pontificia Universidad Catolica de Chile, Escuela de Ingenieria, Departamento de Ciencia de Computacion, Casilla 306, Santiago 22, Chile. ftp://ing.puc.cl/dcc/techReports, 1996.

[19] A. Zakinthinos. "Improved Runtime Complexity for Historical Queries with Aplication to Secure Systems". Department of Computer Science, University of Toronto, 1993. Term Proyect for CSC2532.

Bernardo Siu is a Computer Engineer and a Master on Engineering Sciences (1996). He obtained his degrees from the Catholic University of Chile. In the last two years, he has been doing research at the Logic and Computational Knowledge Representation Group of the School of Engineering of the Catholic University of Chile. Currently he is working at the Applied Research Division of the School of Engineering.

Leopoldo Bertossi is associate professor of the Department of Computer Science (School of Engineering, Catholic University of Chile). He has PhD degree in mathematics from the Catholic University of Chile (1988). He has been visiting professor at the Dept. of Computer Science of the University of Toronto, and at the Dept. of Electrical Engineering and Computer Science of the University of Wisconsin(Milwaukee). He has been a DAAD scholar at the University of Freiburg (Germany). His areas of research are: knowledge representation in AI, computational logic, and databases.