Answering Historical Queries in Databases

(Extended Abstract)

Bernardo Siu and Leopoldo Bertossi
Pontificia Universidad Catolica de Chile
Escuela de Ingenieria
Depto. de Ciencia de Computacion
Casilla 306, Santiago 22, Chile
Email: {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 automatically and efficiently. We adopt Ray Reiter's formalism [17, 19] based on the situation calculus [13] for specifying database updates. For this purpose, we introduce a procedural notion of relevant transactions and tuples, and also a semantical notion of relevance against which the computational notion is compared and established as complete. In this way we are able to answer historical queries, i.e. queries about the entire evolution of the database. This paper extends previous work by Aris Zakinthinos on historical queries [21], which is also based on Reiter's work. Our algorithms have been implemented in SCDBR [3, 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: Specification of Database Updates, Historical Queries, Knowledge Representation, Situation Calculus, Relevance

In [17, 19], Reiter presented a formalism for specifying database updates, using the *situation calculus* [13] as logical language. In this paper we address the problem of posing and answering historical queries on databases, when their dynamics are expressed in that formalism: Given a 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 databases $D_1, \ldots, D_n$, we want to ask queries like "*Is it always the case, 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 can be obtained by mixing this kind of basic queries. 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. Rather, we would like to answer the queries by doing a combination of a kind of list processing on T and a minimal update of the database. This paper describes this approach. Details, proofs, additional examples and results can be found in [20].

The situation calculus regards a dynamic world as evolving from state to state, starting from an initial state S_0. The world is described in terms of dynamic properties, called *fluents* (think of the tables of 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 successor state $do(a, s)$, resulting from executing action a at state s [16]. 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 [17, 18]. 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 of the form

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

The formulas on the right–hand side of the SSAs are *simple* in s [16], that is, s appears free in them and no other state term is mentioned.
Here, the formulas γ_F^+, γ_F^- are the conditions that make $F(\bar{x})$ true or false, respectively, in the state $do(a,s)$. 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 $do(a,s)$. Reiter [16] introduced these SSAs in order to solve the *frame problem* [13] when the conditions γ_F^+, γ_F^- are completely known.

Now we will give an example of historical query: We want to know if Sue has always been working in department 13. By "always" we understand all states along 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

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

Previous work on answering historical queries, when databases are specified as above, was shown in [21] by Zakinthinos. He applies *list processing* on the list of transactions (as suggested by Reiter in [17]), when applicable, and auxiliary 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.

In the rest of this paper, T will be a given, legal transaction list. The most basic kind of query we can think of is to check if, given a formula F and two states s_L, s_U along a transaction list, F holds in all intermediate states. This is expressed by the formula

$$\forall\, s(s_L \leq s < s_U \supset F[s]). \tag{3}$$

This will be called an *universal query.* 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 query. This asymmetry in the range has the good consequence that if s_L and s_U have the same value, i.e. the range of the sub-query is empty, then $F[s]$ does not have to be true at any state s. This might be desirable for posing some queries. The bad consequence is that if we want to consider all states along T, we need to add a new state $S_{N+1} = do(anything, do(T,S_0))$, where N is the length of T.

For purposes of posing historical queries, we need to put restrictions on the bounds of the sub–queries. To do this, we require a function between states 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. $dist(S_i,S_j)$ turns out to be $-dist(S_j,S_i)$.
Having presented the universal queries and the *dist* function we can pose more complex queries by introducing a *canonical query format* (CQF). Basically, an historical query in the CQF is an existential quantification on the bounds of a conjunction of universal queries and constraints on the

bounds. Each boundary constraint is of the form $dist(S_k, S_i)$ op C, where S_k, S_i are either ground state terms or bounds appearing in the query, op is one of the (integer) operators $\leq$, $=$, $\geq$ and C is an integer constant.

For example, formula (2) can be rewritten as:

$$\exists\, s_{L_1}, s_{U_1}\, \forall\, s_1 \left(\begin{array}{c} s_{L_1} \leq s_1 < s_{U_1} \supset emp(sue, 13)[s_1] \wedge \\ dist(s_{L_1}, S_0) = 0 \wedge dist(s_{U_1}, s_{N+1}) = 0 \end{array} \right) \tag{4}$$

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 the 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}). Restrictions $dist(s_{L_1}, S_0) = 0$, $dist(s_{U_1}, s_{N+1}) = 0$ specify that the lower bound must be equal to S_0 and the upper bound must be equal to $do(wait, do(T, S_0))$. Therefore, formula $emp(sue, 13)[s]$ must be true in all states s that lie between S_0 and $do(T, S_0)$, inclusively.

Now, we want to know if there is a state such that John is enrolled in courses $c100$ and $m100$. This can be expressed as

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

and the equivalent query in CQF is:

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

The only restriction over the bounds is $dist(s_{U_1}, s_{L_1}) = 1$, which forces the bounds to be different. This implies that the formula $enrolled(john, c100) \wedge enrolled(john, m100)$ must be true at least in one state.

Having presented the CQF, we can present an algorithm for answering queries in the CQF. It consists of two parts:

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 for path T.

2. From the map and the original query, a new query is generated that only mentions equalities and inequalities between 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.

In the first module, we need to obtain a map of the formulas F_i mentioned in the query. There are at least two ways to achieve this: (1) Using *regression* [16, 17]. The basic idea is the following: 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' that is simple in state S_0 and equivalent to the original query. Formula φ' can be checked against the initial database. (2) Using *progression* [10, 12], that consists 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 database. This can be far more efficient than regression [12]. Progression has been implemented in the programming language GOLOG [8], and in SCDBR [1, 3, 2], applying the algorithms described in [12].

We will assume that this module has been implemented using progression, since it allows us to apply some optimization techniques. These optimizations are required, because the algorithm we have presented here so far is inefficient: Obtaining the map can be very expensive, because it requires

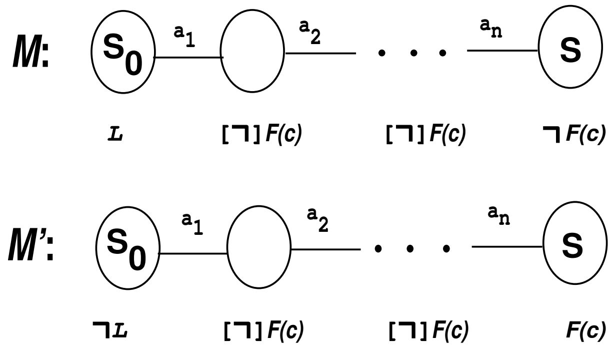

Figure 1: The Semantic Notion of Relevance of Tuples

checking all the formulas F_i in all states along T, updating the entire database with each executed action.

In the rest of this paper 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 executed 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 is not in the spirit of this paper, in which we are interested in analyzing 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 *relevant* entries in tables and *relevant* instances of actions [20]. Usually, an instance of 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.

The question that naturally arises is "How to characterize those relevant actions?". Another related question asks for the minimum number of tuples that must be updated to answer the query. Tuples outside this set and irrelevant actions are not considered when answering the query, because there are no effects on the query associated to them.

Now we will introduce a semantical notion of relevance of sets of tuples. Intuitively, a set of tuples L will be relevant to a tuple $F(\bar{b})$ if a change in the truth value of the elements of L (at some state s[1]) can possibly change the truth value of $F(\bar{b})$ at some state equal to or reachable from s. L must satisfy minimality, i.e. there is no proper subset of L that is relevant to $F(\bar{b})$. We will say that a ground tuple $F'(\bar{c}')$ is relevant to tuple $F(\bar{c})$ if there is a set L such that $F'(\bar{c}') \in L$ and L is relevant to $F(\bar{c})$. Obviously, a tuple will be always relevant to itself.

Figure 1 illustrates the notion of the relevance of a set L of ground tuples to a ground tuple $F(\bar{c})$. M and M' are models of a theory that includes only the SSAs of the specification, dropping the initial database and the precondition axioms, and then making every action possible in every state (so the transaction list $[a_1, \ldots, a_n]$ is always legal according to both M and M'). In the figure, the notation $L, \neg\, L$ at S_0 means that any tuple in L is interpreted differently by both models M and M' at that state, though none of them is forced to interpret as identically true or false all of the tuples in L. These models give the same truth value to the tuple $F(\bar{c})$ in every pair of corresponding

[1]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.

intermediate states $do([a_1, \ldots, a_k], S_0)$, with $k < n$, but each can give $F(\bar{c})$ different truth values along the transaction list. This is expressed with the notation $[\neg]F(\bar{c})$, which means that the tuple $F(\bar{c})$ can either hold or not hold at the corresponding state. M and M' have to give different truth values to $F(\bar{c})$ in state $do([a_1, \ldots, a_n], S_0)$.
An action $A(\bar{a})$ will be semantically 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 $F'(\bar{c})[do(A(\bar{a}, s))]$, being $F'(\bar{c})$ a tuple relevant to $F(\bar{b})$. Therefore, to check the truth value of $F(\bar{b})$ along T, we must compute the effects of $A(\bar{a})$, in case $A(\bar{a})$ 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, saving time.
We are interested in finding algorithms for computing relevant tuples and actions. However, as shown in [20], it is not always possible to obtain a perfect characterization of the (semantically) relevant tuples and actions, because those problems are not recursively enumerable. What we will present is an algorithm that computes a small superset of the relevant actions and tuples. This "unsoundness" might only cause extra computational work, but it does not produce incorrect answers in the algorithms for answering historical queries[2].
The idea of the algorithm is this: Given a 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 tuples that are relevant to the tuples in L. (3) Having the relevant tuples, we compute the set of relevant actions RA. (4) Finally, we evaluate the query, executing only the actions that are in RA and updating only those tuples in R.

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

Example 1 Suppose that we have the following SSAs[3]:

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

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

We want to know if the formula (4) holds. The only tuple appearing in the query is $emp(sue, 13)$, so we must obtain the relevant tuples and actions to $emp(sue, 13)$. For the sake of the example, we will obtain the relevant tuples and actions for a generic tuple $emp(x,d)$ (the names of the variables used are not important), then we will obtain the relevant tuples and actions for $emp(sue, 13)$ by instantiating variables.
We can see from the SSAs that, in order to check if $emp(x,d)[s]$ holds in any state s, we have to inspect $emp(y,d)$ and $boss(y)$ (for any value of y, in particular $y = x$), since the truth value of $emp(x,d)$ depends on the truth values of those tuples: Anyone employed in department d can be promoted to boss and any boss can fire x. The way we obtain these *relevant tuples* is by looking recursively at the formulas $\Phi_{F(\bar{c},\bar{x})}$ appearing in the SSAs[4]. We do not need to look at the tuples

[2] In many cases, including all those of Zakinthinos and all the cases implemented in SCDBR [1, 3, 2], the computationally relevant tuples and actions are exactly the semantically relevant ones.

[3] Predicate $Poss(a,s)$ was dropped and the precondition axioms were eliminated from the specification for not being important for our purpose of computing relevance.

[4] 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 ordered and any part can omitted.

appearing in the (omitted) action precondition axioms, because we will check the query against a transaction list T, known or assumed to be legal: Since the action preconditions are satisfied, we do not need to keep the tuples appearing in them.

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 $\Phi_{emp(x,d)}$, that is, in the formula

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

obtaining $emp(y, d)$ and $boss(y)$ (notice that the fluents do not have the state argument). We will replace the variable y by an *anonymous variable*, _. We delete $emp(x, d)$ from the set of relevant tuples and add $emp(\_, d)$ to it, 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 in

$$a = promote(\_) \vee boss(\_)[s] \wedge \neg\, a = quit(\_),$$

obtaining $boss(\_)$. 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 in

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

The tuples appearing there are $emp(\_, d)$, $emp(y, d)$ and $boss(y)$. We rename y to _, finding out that all of those tuples are already in the set of relevant tuples already, 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, as was shown before.

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 $\{emp(\_, 13), boss(\_)\}$, which are the relevant tuples. In general, an auxiliary table may be necessary, e.g. partially instantiated.

The actions relevant to an arbitrary tuple are obtained by a very 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 $\{fire(\_, \_), hire(\_, d), quit(\_), promote(\_)\}$. Therefore, in order to check the truth value of $emp(Sue, 13)$, we have to compute the effects of the actions $\{fire(\_, \_),\ hire(\_, 13),\ quit(\_),\ promote(\_)\}$, if they appear in T.

For example, if $T = [hire(sue, 13),\ promote(john),\ fire(john, sue),\ change\_salary(john, 15.000),$ $hire(peter, 9)]$, the only relevant actions are $hire(sue, 13)$, $promote(john)$ and $fire(john, sue)$, since they are the ones that are *particular cases* of elements of the set $\{fire(\_, \_),\ hire(sue, 13),\ quit(\_),$ $promote(\_)\}$. 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. This could make the recursive search go into an infinite loop. However, our algorithm will always stop, because it keeps only one version the "most general" tuples, without state arguments. A tuple $F(\bar{c}, \bar{x})$ is more general than the tuple $F'(\bar{c}', \bar{x}')$ if there is a variable substitution σ such that $F(\bar{c}, \bar{x})\sigma = F'(\bar{c}', \bar{x}')$, that is, $F'(\bar{c}', \bar{x}')$ is a particular case of $F(\bar{c}, \bar{x})$.

The set of tuples which are computationally relevant to a tuple $F(\bar{c}, \bar{x})$ is denoted by $CRT(F(\bar{c}, \bar{x}))$. The set of actions computationally relevant to a tuple $F(\bar{c}, \bar{x})$ is denoted by $CRA(F(\bar{c}, \bar{x}))$. In the example, we have that

$$CRT(emp(sue, 13) = \{emp(\_, 13), boss(\_)\},$$
$$CRA(emp(sue, 13) = \{fire(\_, \_), hire(\_, 13), quit(\_), promote(\_)\}.$$

One of our main results tells us that all the tuples that are semantically relevant to $F(\bar{c},\bar{x})$ are particular cases of elements in $CRT(F(\bar{c},\bar{x}))$. The same result holds for the actions that are semantically relevant to $F(\bar{c},\bar{x})$ wrt $CRA(F(\bar{c},\bar{x}))$ when SSAs are obtained from effect axioms [16]. Notice that, since the relevant tuples and actions are obtained from the SSAs, which are static, we have to compute relevance just once, prior to checking any query.

Having shown how to compute relevance, we can now improve the previous algorithm for answering queries in the CQF. Basically, the new one skips the irrelevant actions, i.e. the actions that are not computationally relevant to the query. The truth values of formulas F_i are stored in a *map*.

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[5].

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

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

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

   3.0. Make $j = 0$. This is a counter that keeps the correlative number of the current action in the transaction list T, of length N.

   3.1. For all i do: Check F_i in S_0. Store the truth value of F_i in the map.

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

   3.3. If action a_j is not a particular case of some element in 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. Store the truth value of F_i in the map.

   3.5. Go to step 3.2.

   3.6. We have just obtained the map. Go to step 4.

4. Given the map, obtain a CSP and check if it can be satisfied.

Now we will briefly describe the implementation of the algorithm we have presented in this paper. The implementation has been done as a module of our more general computational system SCDBR, for (*Situation-Calculus-Based Database Reasoner*). Preliminary reports about the system can be found in [1, 3, 2]. The system is implemented in SICStus Prolog.
The system can derive a set of SSAs from a set of effect axioms. Having obtained the final specification, SCDBR can perform, among others, the following tasks: Decide if a transaction list is legal, regress a query to the initial state and check it against the initial extensional database. For this checking, SCDBR can call mechanical theorem provers and database management systems, such as OTTER [14], PROLOG, ORACLE. SCDBR can also prove *integrity constraints* (ICs) from the specification [4]. In some cases, given an IC that is not entailed by the specification, it can change the specification in order to subsume the IC, making it true in every legal state in the evolution of the database [11, 15].

[5] It is good enough to process the formulas F_i linearly: If F_i is $F(x) \wedge x = a$, we collect $F(x)$, and not $F(a)$.

We have implemented in SCDBR all of the work presented in here: the system 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 a given specification. The relevant tuples and actions to an instantiated tuple are obtained by variable instantiation, which 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 boundary constraints. 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.

The mapping module is an implementation of the algorithm that uses relevance, presented above. Irrelevant actions are replaced by an "idle" operation `wait` prior to the execution of this module, so their effects are not computed.

The checking module is currently implemented using *constraint logic programming*, specifically, with clp(fd) [7], which is called once the map has been obtained. Basically, we automatically generate a program written in clp(fd) that represents the CSP. This clp program gives the final answer to the original query.

Now we will compare our work with related approaches by other people. In order to answer historical queries, Zakinthinos [21] considers only *context free* SSAs [12]. For example, the SSA of $boss(x)$ in Example 1 is context free: The truth value of $boss(x)$ does not depend on other tuples, but only on the executed actions. His underlying idea is to make as much list processing as possible, while we always do database updating. However, some special cases of our algorithm coincide, in esence and complexity, with Zakinthinos' algorithms. However, we do not rely on any special syntactical form of the specification as he does. In that sense, our work is more general.

In [9], Lin and Reiter formalized the notion of *forgetting*. Given a theory Σ and S, a set of ground instances of predicates, they denote by $forget(\Sigma, S)$ the theory resulting from forgetting the elements of S in Σ. The new theory can be obtained by syntactical manipulation if Σ and the cardinality of S are finite. The cost of this manipulation is exponential in the cardinality of S. In their semantical counterpart, a structure M is model of $forget(\Sigma, S)$, iff it agrees with a model of Σ on everything except the truth values of the elements in S.

They say that a set S of ground instances of predicates is *irrelevant* to a query q (wrt Σ) whenever it holds

$$\Sigma \models q \textit{ iff } forget(\Sigma, S) \models q \text{ and } \Sigma \models \neg\, q \textit{ iff } forget(\Sigma, S) \models \neg\, q$$

This means that a set S is irrelevant to q iff the entailment of q from Σ can be obtained regardless of the truth value of the elements in S. Notice that in order to check if S is irrelevant to q, we must have $forget(\Sigma, S)$, i.e., we need S prior to checking relevance.

Our notion of relevance of tuples can be expressed in terms of Lin and Reiter's forgetting. However, expressing the relevance of even a single tuple may require forgetting many other tuples, so the cost of computing the theory *forget* can be very high.

Lin and Reiter also show how to forget about an entire relation. However, we usually want to forget only part of a relation, though that part can be of infinite cardinality (this happens when the irrelevant tuples are partially instantiated, and the rest of the variables can take an infinite number of values). In consequence, it is not very practical, for our purposes, to think of relevance in terms of forgetting.

From a computational point of view, there is a fundamental difference between their approach and ours: They do not compute a list of tuples that are relevant to a query, but they can guess a set

S, then compute *forget*(Σ, S) and finally check irrelevance. By exhaustive checking of irrelevance they could obtain relevance. Our computational approach allows us to obtain relevance directly. Another difference is that they do not consider the relevance of actions, but only irrelevance of (instances of) predicates. Our personal view is that not only the relevant information, as the one stored in the fluents, is important, but also the *mechanisms* that make relevant information change, i.e. the actions. Handling relevance in the situation calculus, with specifications à la Reiter, make this last task easier, because the effects of the actions are explicitly shown.

Blakeley et al. [5] introduce a notion of irrelevance for updates of relational databases. An update in a base table will be irrelevant to a view (obtained by applying operations from relational algebra to a set of base relations), if it does not produce a change in that view. They gave necessary and sufficient conditions for deciding if an update is irrelevant to a view. In some cases, those conditions can be checked efficiently. Although we have not worked out the details, we believe that after translating their setting into Reiter like specifications, their relevant updates would be a particular case of our relevant actions. Proving this would provide an interesting class of specifications such that the problem of determining relevance of actions is decidable and efficiently computable, namely, the one identified in [5].

A work very much in the spirit of ours is presented by Chomicki in [6]. 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 is that [6] does not introduce any explicit notion of relevance.

In this paper, we have shown how to use the notions of relevance for optimizing the process of answering historical queries. In [20], we describe in detail all the ideas and algorithms presented here. In particular, a general algorithm for computing relevance of tuples and actions can be found. Also precise statements and proofs of all the claims made in this paper are provided there. In particular, the results about the non recursive enumerability of semantical relevance and about the subsumption of the semantical notion of relevance by its computational counterpart. Other uses of relevance are also sketched.

Acknowledgments

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

grants.

References

[1] Leopoldo Bertossi, Marcelo Arenas, and Cristian Ferretti. SCDBR: An Automated Reasoner for Specifications of Database Updates. *To be submitted for publication*, 1996.

[2] Leopoldo Bertossi, Marcelo Arenas, and Cristian 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.

[3] Leopoldo Bertossi and J. Cristian 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*, pages 543–545, Bonn, Germany, 1994. Springer.

[4] Leopoldo Bertossi, Javier Pinto, Pablo Saez, Deepak Kapur, and Mahadevan Subramaniam. Automating Proofs of Integrity Constraints in the Situation Calculus. In *Foundations of Intelligent Systems. Proc. Ninth International Symposium on Methodologies for Intelligent Systems (ISMIS'96), Zakopane, Poland.* Springer, LNAI 1079., 1996.

[5] 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.

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

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

[8] 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.

[9] F. Lin and R. Reiter. Forget It! In *Presented at the AAAI Fall Symposium on Relevance, New Orleans*, 1994.

[10] 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.

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

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

[13] 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 University Press, 1969.

[14] W.W. McCune. *OTTER 3.0 Reference Manual and Guide.* Argonne National Laboratory, Technical Report ANL-94/6, 1994.

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

[16] 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.

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

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

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

[20] Bernardo Siu and Leopoldo 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.

[21] 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.