

Plan and Schedule Generation over Temporal Databases

Antonio L. Furtado^{1,2} and Marco A. Casanova²

¹Departamento de Informática, Pontificia Universidade Católica do R.J.
22.453, Rio de Janeiro - Brasil

²Rio Scientific Center, IBM Brazil,
P.O. Box 4624, 20.001 Rio de Janeiro RJ, Brazil

Abstract

A clausal specification of operations is shown to be adequate to both plan generation and temporal databases recorded by way of update-oriented structures. By combining these two concepts, it is then shown how databases can support in a uniform way queries involving past, present and future states, as well as both immediate and future (revokable) updates, expected to take place at a specified time. These databases also permit the generation of plans coupled with time schedules.

1. INTRODUCTION

The contribution of this paper is twofold. We first describe a declarative way of specifying both the structure *and the operations* of an entity-relationship schema. We also show that, after formalizing certain common assumptions, the specification is executable.

We then describe a plan generation algorithm and a method to introduce the time dimension, whereby the facts that hold at a certain instant can be inferred from the record of the operations executed. By combining these features, we show how to extend temporal databases so as to cover past, present and future states, as well as to draw plans coupled with time schedules.

The ideas discussed in the paper have been tested through a prototype implementation. In the prototype, the entity-relationship concepts are defined on top of a logic programming / relational environment handled by a Prolog / SQL system [WM].

The paper is organized as follows. Section 2 introduces the specification of operations by clauses. Section 3 discusses plan generation, whereas section 4 introduces the temporal dimension. The full combination of all these features, which is the main thrust of the paper, is described in section 5. Section 6 contains the conclusions.

2. DECLARATIVE SPECIFICATION OF ER CONCEPTS

It is not surprising that the entity and relationship classes of a particular database application can be specified in a declarative style by way of clauses, with the syntax below. Two remarks on attributes are in order: (1) keys of an entity or relationship class are implicitly understood to be attributes of the class and, thus, *attribute* clauses are only necessary for non-key attributes; (2) for convenience, we require that there must be a single *domain* clause for each attribute, noting that attributes can be shared by several entity or relationship classes.

```
entity(<entity class>,<key>)
relationship(<relationship class>,<participant classes>)
attribute(<entity class>,<attribute>)
attribute(<relationship class>,<attribute>)
domain(<attribute>,<description of possible values>)
```

What is less obvious is that operations can also be specified by clauses. For an operation O, the clauses should indicate its effects, in terms of facts added and/or deleted by O, and the pre-conditions for its application. Pre-conditions may in turn involve positive and negative (i.e. negated) facts, which may be added or deleted by other operations [Ni,FN]. The syntax of the clauses to specify operations is:

```
added(<fact>,<operation>)
deleted(<fact>,<operation>)
precond(<operation>,<expression involving facts>)
```

In turn, a fact may consist of either the simple declaration that an entity or relationship instance exists or that the instance has a given value for some attribute. The syntax for facts is:

```
<entity class>ϕ<key>
<entity class>ϕ<key>\<attribute>(<value>)
<relationship class>#[<participants list>]
<relationship class>#[<participants list>]\<attribute>(<value>)
```

where <participants list> is a list of pairs of the form <entity class>ϕ<key>. A *state* is then a set of facts with the above syntax.

The clauses that form the specification of an operation O may contain (logical) variables which, at a first glance, may look similar to the input/output variables of a procedure. However, these variables play a more general role since we let them be replaced by arbitrary terms, in particular by other variables, whereas the input/output variables of a procedure can only be substituted by values.

An *instance* O' of an operation O is then defined as a set of clauses obtained by uniformly substituting the variables occurring in the specification of O by terms. As we shall see, the algorithms we describe in this paper will generate operation instances mostly through a unification process.

In the spirit of abstract data types specifications, we decided that, in our original prototype, most of the behavior of entity and relationship instances would be fixed by the clauses related to operations. However it is certainly possible to expand the repertoire to include clauses that separately declare what options to use in view of constraints inherent in the model (existence of instances of entities participating in relationships, "weak" entities which only exist as long as they participate in a relationship and which we do not allow to have non-key attributes, etc.). The present prototype only fixes that deleting an instance of an entity or relationship class implicitly deletes all its attributes, and thus exhaustive lists of delete clauses for each attribute are unnecessary for operations declared to delete such instances. A forthcoming version of the prototype will also automatically generate the pre-conditions necessary to guarantee that the operations preserve the semantic constraints associated with relationship classes.

The specification of operations will remain incomplete unless we provide answers to the following questions:

- (a) what happens if an operation instance is applied to a state where its pre-conditions do not hold?
- (b) what happens if an operation instance is applied to a state where some of its intended effects already hold?
- (c) if an operation instance is applied to a valid state (where the problems above do not occur), how will the old state be transformed into the new one?

One possible strategy [VF] - that we shall adopt here - is to establish in cases (a) and (b) that nothing happens, i.e., that no state change is produced. In case (a) the operation instance is said to be non-applicable, whereas for (b) it is said to be non-productive (vacuous). Case (c) corresponds to the so-called frame problem [Ko]. As expected, we establish that facts in *added* and *deleted* clauses will be changed as these clauses stipulate; any other fact will hold in the new state if and only if it held in the old state.

Having settled for this strategy, we can express it under the form of algorithms, whereby we end up with an executable specification. The two algorithms below show in outline how to perform queries and updates driven by the *added*, *deleted* and *precond* clauses.

Algorithm 2.1: query a recorded fact

query(F)

input:

 F - fact

steps:

- 1) simply look for the record of F in the current state
-

Algorithm 2.2: execute an operation instance

execute(O)

input:

O - operation instance

steps:

- 1) check if the pre-conditions to the execution of O indicated in the respective *precond* clause hold in the current state
 - 2) obtain the sets Sa and Sd of facts to be added and deleted by O, as indicated in the respective *added* and *deleted* clauses
 - 3) check if no fact in Sa holds in the current state and if all facts in Sd do hold
 - 4) record all facts in Sa and remove the records of all facts in Sd from the current state
-

It should be clear that, if we handle a database with these algorithms, we can only answer queries about the present (i.e. the current state).

3. PLAN GENERATION

The plan generation process consists of, given a goal, creating a sequence $O_1 \Rightarrow O_2 \Rightarrow \dots \Rightarrow O_n$ of operations - the *plan* - able to lead from the current state to a state where the goal is true.

In the simplest situation, a goal consists of a single fact that we want to become true, and the pre-condition to each operation can involve one fact at most. For example, we want that student John be taking the Math course. The algorithm below shows how to generate plans of this kind, using the same *added*, *deleted* and *precond* clauses. Note that the algorithm is recursive.

Algorithm 3.1: generate a plan for a simple positive goal

plans(G,P)

input:

G - goal consisting of a single positive fact

output:

P - plan

steps:

- 1) if G already holds in the current state, P consists of the trivial sequence *start*
- 2) otherwise, search for an operation declaration that contains an *added* clause able to add G, that is, whose first term unifies with G, and which is productive, as tested in steps 2 and 3 of algorithm 2.2. Let O' be the instance of O resulting from such unification

- 3) check if the pre-condition C' to apply O' , indicated in the respective *precond* clause, already holds; the test is made over the partial plan P' thus far generated:
 - if P' is *start*, simply check if C' holds at the current state
 - if P' is non-trivial, the test is recursive: C' holds at the state reached by P' , where P' is $P'' \Rightarrow O''$, if either an *added* clause declares that O'' adds C' and the pre-condition to apply O'' holds at the previous state P'' , or C' already held at P'' and there is no *deleted* clause declaring that O'' deletes C' ; if the test succeeds, P is $P' \Rightarrow O'$
 - 4) otherwise, take C' as (sub-)goal and apply the algorithm recursively to generate a partial plan Q' to reach an intermediate state where C' holds; P will then be $Q' \Rightarrow O'$
-

Limiting a goal to a single positive fact is too restrictive. Also the pre-condition for an operation instance may involve more than one fact. The obvious way to allow goals consisting of a conjunction of facts $F1 \ \& \ F2 \ \& \ \dots \ \& \ Fn$ is to extend algorithm 3.1 so that, after forming a partial plan P' to satisfy $F1 \ \& \ F2 \ \& \ \dots \ \& \ Fi$, we apply 3.1 again to transform P' into $P'' \Rightarrow O'$, where O' adds $Fi+1$ and P'' is either P' or P' followed by one or more operation instances leading to a state where O' is applicable.

However, the facts in the goal conjunction may be mutually interfering, so that it may not be possible to generate a plan by trying to satisfy each fact in the order they are given, whereas a feasible plan would be found if a different order were considered. Warren's algorithm [Wa], which he implemented in Prolog, does that by providing an auxiliary predicate - *achieve* - to consider the two following situations:

- (a) firstly, *achieve* tries to add a new operation instance O' to the end of the plan P' generated thus far, after extending P' , if required by the pre-conditions of O'
- (b) if this fails, it tries to insert the operation instance (and possibly others to satisfy the pre-conditions of O') somewhere inside P' , provided that the operation instances in P' that will follow O' do not nullify its effects

The first call to his algorithm then involves two plans among its parameters: the initial plan *start* and a variable, that will be instantiated with the final plan, generated by extending or interpolating the successive partial plans according to the two alternatives for the *achieve* strategy.

We further extended Warren's algorithm [VF,Fu], especially to also allow negative facts in the goal conjunctions and to test for the requirement that applications of operation instances be productive. For brevity, we shall not describe Warren's algorithm and our extensions here. The reader is asked to assume, however, that from now on algorithm 3.1 will designate an algorithm with all such capabilities. In particular, alternative (b) of *achieve* is crucial in the presentation of algorithm 5.3 in section 5.

Plan-generation takes us beyond the current state, but only in a "hypothetical" way. It is easy to see that reachable future states can be denoted by plans. By using algorithm 3.1 in a reverse way - taking P as input and G as output - we can ask what is the difference between the current state and the state that the plan would generate, in terms of facts added and deleted. If we start from an "empty" initial state, this difference is the entire contents of the database at the state that would be reached at the end of the plan's execution. For example, given a suitable specification of an academic database (see section 5), the plan

```
start=>offer(a,2)=>enroll(u,a)=>enroll(v,a)=>offer(b,1)=>transfer(u,a,b)
```

denotes a state where courses a (with two credits) and b (with one credit) are offered, student u takes b and student v takes a.

4. UPDATE-ORIENTED TEMPORAL DATABASES

In this section we leave aside, for a moment, the idea of planning. Instead of viewing a sequence of operation instances as a plan, denoting a hypothetical state, we shall regard it as a *trace* [BP], i.e., an expression recording which operation instances have actually been executed and letting the order of their appearance correspond to the order of execution. Then, instead of storing the facts added by the operation instances that form the trace, we might store the trace itself and infer the facts by applying algorithm 3.1 in the reverse direction, as suggested at the end of the previous section. From a theoretical point of view this is interesting in the sense that it recalls the representation of instances of data structures by terms formed by the operations used to build them, as in the algebraic approach to abstract data types. But from a practical point of view this representation does not fit well with the record-based file structures used for the physical storage of databases.

What we can do is to break the trace into its constituent operation instances and to record these instances together with the time of execution, calling this extra item a *time stamp*. Time stamps are an expedient way to preserve the information about the order of execution, which would be lost when the trace is decomposed. The example trace would thus become:

```
offer(t1,a,2)
offer(t4,b,1)

enroll(t2,u,a)
enroll(t3,v,a)

transfer(t5,u,a,b)
```

where each t_i will be some representation of an instant of time, say the list [year,month,day,hour,minute,second,fraction of second], with a granularity fine enough to guarantee that time stamps be unique, so as to preserve total ordering among executed operations. To simplify the discussion and the design of the initial prototype implementation, we chose to ignore the important distinction between time of execution of operation instances in the "real

world" and in the "system" [Bu]; for the same reason, we assume that the execution of an operation instance is instantaneous.

To pass to file structures, we can establish that the parameter lists of all instances of an operation be physically kept in a file bearing the name of the operation. A file TRANSFER would have columns TS (time stamp), STUDENT, COURSE1, COURSE2. Clearly, these *update-oriented structures* are compatible with the relational model and can be handled by a relational DBMS. Our Prolog / SQL implementation uses an interface whereby ground unit Prolog clauses and SQL records are treated uniform and transparently in Prolog.

These structures enable a form of temporal databases since we can now ask queries of the form <fact> @ <time instant>, to retrieve facts holding at time instants denoting the present or past states. The principles on which they are based were first formulated in [Bu] and they are described in detail in [TF], where their advantages and disadvantages are compared to those of other structures for temporal databases. A similar approach is adopted in [KS]. Finding whether a fact holds at a specific time is done indirectly by examining the time-stamped records of the appropriate operation instances. On the other hand, executing an operation is now somewhat simpler: instead of performing additions and deletions of facts, we install the parameter list (plus time stamp) of the operation instance in the corresponding file. However, the relevance of update-oriented structures to our present purposes comes mostly from their compatibility with plan and schedule generation, as will be shown in the next section.

Algorithm 4.1: query a fact at an indicated time

query(F,T)

input:

F - fact

T - time instant

steps:

- 1) look, in the *added* clauses, for an operation instance O capable of adding facts of the type of F
- 2) transform O into Ot, by adding to the parameter list of O an extra parameter St to correspond to the time stamp
- 3) search for an Ot record
- 4) check if the St of the located record is smaller or equal than T
- 5) look, in the *deleted* clauses for all kinds of operation instances O' capable of deleting F
- 6) transform them into Ot' by adding time stamp parameters St'
- 7) check if there is no Ot' with St' in the interval St :T

Algorithm 4.2: execute an operation instance

execute(O)

input:

O - operation instance

steps:

- 1) check if the pre-conditions to the execution of O indicated in the respective *precond* clause hold
 - 2) obtain the sets Sa and Sd of facts to be added and deleted by O, as indicated in the respective *added* and *deleted* clauses
 - 3) check if no fact in Sa currently holds and if all facts in Sd do hold
 - 4) obtain the current time T and transform O into Ot, by adding T to the parameter list of O
 - 5) store the parameter list of Ot in the file whose name is the functor of O (as also of Ot)
-

5. FUTURE STATES AND PLANS WITH SCHEDULES

Since the method we are using to specify operations can enable both plan-generation and temporal databases, it should allow the simultaneous use of both features. In this section we investigate useful ways to do this combination.

To begin with, let us consider the extension of temporal databases to also cover future states. If we know (or intend) that an operation will be executed at some specific future date, it makes sense to record it immediately, having this date as time stamp. Queries over future states can then be posed in exactly the same way we described for present and past states.

However some difficulties arise. If we record an operation instance O with time stamp T, and later wish to record another operation instance O' with time stamp T' such that $T' < T$, we must first check whether O' would not disturb O, since O' may negate a pre-condition for O or may make O non-productive. In general, when we add the record of an operation instance On+1 with time stamp lesser than those of O1, O2, ... , On, we need to check if On+1 will not disturb any of these operation instances.

It may also happen that we learn that some operation instance O1, with time stamp less than those of O2, O3, ... , On will not in fact be executed, in which case we should revoke the record of O1. Any implementation should provide that, when the activities of a day D start, some trigger mechanism will request that the persons in charge confirm or not the execution of all operation instances with time stamp referring to D. But cancelling an operation instance may disturb one or more of the other operation instances, requiring the propagation of revoke actions.

Fortunately, plan-generation helps implementing the necessary checks, both when recording and when revoking future operation instances. Briefly, the sequence of future operation instances, starting from and including an opera-

tion instance added (or following an operation instance removed), must be a valid plan.

We just saw how, starting with temporal databases, we find that plans may help. Now we start with plan generation and extend the notion by introducing temporal lower and upper bounds to define:

- when applications of a given operation are allowed (to be indicated by *time_bounds* clauses);
- when a plan must be drawn (to be indicated by extra bounds parameters of the plan generation command)

Like the *precond* clauses, *time_bounds* clauses can be regarded as constraints to the execution of operation instances. The bounds parameters limiting the entire plan indicate, respectively, the earliest (latest) time for the first (last) operation in the plan. Intuitively, when lower bound T_i and upper bound T_f are indicated for a plan P to accomplish a goal G , this means: "P must not begin before T_i and must not go beyond T_f ; G must hold at the (hypothetical) state resulting from P and must persist at time T_f ". To capture these time considerations, we define a *schedule* for a plan as a sequence of time bounds for the execution of each operation instance in the plan.

Both extensions may co-exist, i.e., we may want to draw plans coupled with schedules in the presence of future *recorded* operations. To form such a plan, we first copy the future recorded operations that lie in the time interval assigned to the plan, and then "interpolate" other operations (to be called *planned* operations) in order to achieve the desired goal. We note that the two classes of operations have a different strength, in the sense that the planned operations are hypothetical whereas there is a commitment (except for possible revocation) with respect to the recorded ones. The schedule S for such plan P will accordingly be a sequence consisting of:

- *time_stamps* for the recorded operations
- pairs of time bounds for the planned operations

This is where alternative (b) for the *achieve* predicate mentioned in section 3 can be used in an easy adaptation: the "interpolation" problem can be described as the transformation of a plan P' , consisting of the sequence of all future recorded operations within the indicated time bounds, into a plan P that satisfies the given goal. The time bounds of the new operations will depend on the global bounds of the plan, the *time_bounds* clauses declared for the operation and the bounds determined for the previous and the next operation instances. Finally, we require that P , extended with the operation instances recorded with time stamp beyond the global upper bound of P , be also a valid plan.

The detailed algorithms follow.

Algorithm 5.1: record the execution of an operation instance
execute(O,T)

input:

- O - operation instance
- T - time for its execution

steps:

- 1) check whether T is equal or greater than the current time
- 2) check if T is within the bounds $B_i:B_f$ of O, if these have been indicated
- 3) check that no recorded operation instance has time stamp equal to T
- 4) obtain the sequence Q' of operation instances with time stamps greater than T
- 5) check if $start(T) \Rightarrow O + Q'$ is a valid plan, where $start(T)$ refers to the state at time T and '+' denotes the concatenation of sequences
- 6) record O with time stamp T

Algorithm 5.2: Revoke a future recorded operation instance

revoke(O)

input:

- O - operation instance

steps:

- 1) find the instance of O with smallest time stamp T larger than the current time
- 2) ask if this is the instance that the user wishes to revoke, otherwise look for the next instance and ask again
- 3) remove the record of the instance selected
- 4) obtain the sequence Q' of operation instances with time stamps greater than T
- 5) for every operation instance O' in Q' check if $start(T') \Rightarrow O'$, where T' is the time stamp of the previous operation (or T, when O' is the first operation instance in Q'), is a valid plan and otherwise remove the record of O'

Algorithm 5.3: generate plan with schedule

sch_plans(G, $T_i:T_f$, P, S)

input:

- G - goal
- $T_i:T_f$ - time limits of the plan

output:

- P - plan
- S - schedule

steps:

- 1) Obtain:
 - initial plan Q, formed by all recorded operation instances with time stamps between T_i and T_f
 - initial schedule S' , formed by the time stamps of operation instances in Q

- the sequence Q' of operation instances with time stamps greater than T_f
- 2) Apply plan-generator to expand Q into a new plan P so as to satisfy G
 - 3) Check if $P+Q'$ is a valid plan
 - 4) Expand S' into a new schedule S that includes time limits $L_i:L_f$ for each planned operation instance O in P . The criteria, admitting that there may or may not be a *time_bounds* clause for O indicating $B_i:B_f$ as bounds, are:
 - L_i is equal to the time stamp of the previous operation instance recorded or to B_i , whichever is the largest
 - L_f is equal to the time stamp of the next operation instance recorded or to B_f , whichever is the smallest
 - if O is the first (last) operation instance in P then T_i (T_f) is considered as the previous (next) time stamp for the comparison mentioned in the two items above
 - if the previous (next) operation instance is also a planned operation instance then its L_i (L_f) is inherited by O
-

As an illustration, we now specify a simple-minded academic database. The reader will note that the syntax introduced in section 2 has been slightly extended; the clauses for operations, for example, have an additional state parameter. Also, the notation is exactly that of IBM Prolog [WM], which allows the definition of additional operators (e.g. #).

Some of the consequences of the *added*, *deleted* and *precond* clauses can be readily perceived by inspecting these clauses. Students cannot be enrolled in (or transferred to) a course unless the course exists, and instances of courses can only be created by the *offer* operation. On the contrary, a student exists only as long as he is taking a course (a case of weak entity). A course can only be cancelled after all students taking it, at the state just before the operation is executed, cease to take the course (the appropriate sub-goal, taking the form of a conjunction of negated facts, is generated by predicate *neg_conj_at*). On the other hand, certain constraints need not be explicitly formulated; for example, transferring a student from a course to the same course is disallowed by the productivity requirement, which also guarantees that credits is a single-valued attribute of courses. The *time_bounds* clause imposes the temporal constraint that courses can only be created in March.

The specification follows:

```
% declaration of ER classes
entity(student,s_num).
entity(course,c_num).

relationship(takes,[student,course]).

attribute(course,credits).
```

```

domain(s_numb,X) <- on(X,[u,v]).
domain(c_numb,X) <- on(X,[a,b,c]).
domain(credits,X) <- in_range(X,1,2).

% declaration of operations
added(course#C,offer(C,N),*).
added(course#C\credits(N),offer(C,N),*).
added(takes#[student#S,course#C],enroll(S,C),*).
added(takes#[student#S,course#C2],transfer(S,C1,C2),*).

deleted(course#C,cancel(C),*).
deleted(takes#[student#S,course#C1],transfer(S,C1,C2),*).

precond(offer(C,N),true,*).
precond(cancel(C),X,St) <-
  neg_conj_at(X,takes#[student#*,course#C],St).
precond(enroll(S,C),course#C,*).
precond(transfer(S,C1,C2),course#C2,*).

time_bounds(offer(C,N),[Y,3,1]:[Y,3,31],*) <-
  now([Y!*]).

```

Now suppose the current date is March 23, 1990 and the following operation instances have been recorded in the appropriate files, with the time stamps reduced to year/month/day for readability (note that the last three operation instances belong to the future):

```

offer([90,3,1],a,1).
enroll([90,3,2],u,a).
offer([90,3,25],b,1).
enroll([90,4,2],u,b).
enroll([90,4,18],v,b).

```

The facts that can be derived at the current state are that course a is offered with one credit (course#a\credits(1) @ [90,3,23], using the notation of section 4) and that student u takes it. On April 18th all this is still true and, in addition, b is offered with one credit and is taken by students u and v. Some queries to retrieve facts like these are:

```

<- now(T) & course#a\credits(N) @ T.
<- takes#[student#u,course#C] @ [90,4,20].

```

As a result of the first query, T will be instantiated with the current time, read from the system's clock, and N will be instantiated with 1. The second query causes C to be instantiated with a and, if backtracking is requested, with b.

The execution of the first command below will fail, since 90/4/6 exceeds the time bounds for offering courses, whereas the second will succeed and it will propagate to revoke both *enroll* operation instances involving course b.

```
<- execute(offer[c,2],[90,4,6]).
<- revoke(offer(b,1)).
```

Finally, let us look at the generation of a plan with a schedule. Starting with the original records of operation instances (thus assuming that the above revoke command was not entered), consider the goal: "course a must not be offered, some (unspecified) course with two credits must be offered and student u must take it", with the requirement that "the plan to reach a state where this goal holds should not begin before March 10th and should be completed by April 10th". The command to generate the corresponding plan P with schedule S is:

```
<- sch_plans(-course#a & course#X\credits(2) & takes#[student#u,course#X],
             [90,3,10]:[90,4,10], P, S).
```

with P and S being instantiated with sequences that are best displayed in tabular form. Notice that the creation of course c (with two credits) has been scheduled inside the period of time when this is permitted.

<u>Plan</u>	<u>Schedule</u>
offer(b,1)	----- 90/3/25
offer(c,2)	----- 90/3/25:90/3/31
enroll(u,b)	----- 90/4/2
transfer(u,a,c)	---- 90/4/2:90/4/10
cancel(a)	----- 90/4/2:90/4/10

We also investigated a simpler albeit useful combination of temporal databases and planning, suggested by D. Schwabe. It contemplates plans starting at a past date T and ignores operations with time stamp greater than T. Generating one such plan P given a goal G means: "if we reverted to the situation prevailing at time T, what should be done to satisfy G?"; or, if a plan P is given: "if P were executed at time T, what would be its effects?".

6. CONCLUDING REMARKS

A prototype Prolog / SQL implementation is operational with all features described in section 5. The implementation includes some additional features, whose details are not essential to the present discussion and which will simply be enumerated here:

- is a hierarchies among entity classes, with inheritance of attributes and of participation in relationships;
- frame structures collecting <attribute:value> pairs pertaining to the same entity or relationship instance;
- operations on frames, especially frame unification and frame (most specific) generalization [Kn];
- *imposs* clauses (already part of Warren's original plan-generation algorithm), to declare integrity constraints independently of the execution of specific operations;

- queries over time intervals, defined on top of the basic implementation which refers to instants of time.

Future work will consider, among other objectives, the further extension of the plan/schedule generation algorithm, in special to permit disjunction in goal expressions. We also recognize that much effort is still required to improve the efficiency of the various algorithms, to the point that the prototype can be transformed into a tool able to handle databases of realistic size and complexity.

To characterize the contribution of our research, we recall that information as contained in databases in general is fundamental to decision-making. We claim that adding the time dimension, so as to encompass the anticipated future, and offering plan/schedule generation are significant enhancements to decision support processes.

REFERENCES

- [BP] W. Bartussek and D. Parnas - "Using traces to write abstract specifications for software modules" - technical report 77-012, University of North Carolina (1977).
- [Bu] J. A. Bubenko Jr. - "The temporal dimension in information modelling" - in "Architectures and models in data base management systems" - G. M. Nijssen (ed.) - North-Holland Pub. Co. (1977) 93-118.
- [FN] A. L. Furtado and E. J. Neuhold - "Formal techniques for database design" - Springer Verlag (1986).
- [Fu] A. L. Furtado - "Some extensions to Warren's plan-generation algorithm" - PUC/RJ technical report 20/89 (1989).
- [Ko] R. Kowalski - "Logic for problem-solving" - North-Holland Pub. Co. (1979).
- [Kn] K. Knight - "Unification: a multidisciplinary survey" - ACM Computing Surveys 21, 1 (1989) 93-124.
- [KS] R. Kowalski and M. Sergot - "A logic-based calculus of events" - New Generation Computing 4, 1 (1986).
- [Ni] N. J. Nilsson - "Problem-solving methods in artificial intelligence" - McGraw-Hill Inc. (1971).
- [TF] L. Tucheran and A. L. Furtado - "Update-oriented database structures" - Proc. of the Second International Conference on Expert Database Systems (1988) 185-203.
- [VF] P. A. S. Veloso and A. L. Furtado - "Towards simpler and yet complete formal specifications" - in "Information systems: theoretical and formal aspects" - A. Sernadas, J. Bubenko and A. Olive (eds.) - North-Holland Pub. Co. (1985) 175-189.
- [Wa] D. H. D. Warren - "WARPLAN: a system for generating plans" - memo 76 - University of Edinburgh (1974).
- [WM] A. Walker, M. McCord, J. F. Sowa and W. G. Wilson - "Knowledge systems and Prolog" - Addison-Wesley Pub. Co. (1987).