Expert Database Systems

Proceedings from the Second International Conference

Editor

Larry Kerschberg

Proceedings From the Second International Conference on

EXPERT DATABASE SYSTEMS

LARRY KERSCHBERG, EDITOR

George Mason University

THE BENJAMIN/CUMMINGS PUBLISHING COMPANY, INC.

Redwood City, California • Fort Collins, Colorado
Menlo Park, California • Reading, Massachusetts • New York
Don Mills, Ontario • Wokingham, U.K. • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan

UPDATE-ORIENTED DATABASE STRUCTURES

by

Luiz Tucherman and Antonio L. Furtado*

Rio Scientific Center
IBM Brasil

ABSTRACT

The paper proposes a non-conventional update-oriented structure and an update discipline for rule-driven temporal database applications. The rules incorporate, in a declarative form, (a) knowledge related to the update discipline and (b) knowledge of specialists in the given application area. Essentially, the rules of type (a) determine how to apply the type (b) rules to the manipulation of the database specified, so as to preserve correctness. Specifications in terms of these rules are immediately executable under a prototyping tool. The tool is written in Prolog, employing meta-level programming techniques, and SQL.

1. INTRODUCTION

Database structures are usually **query-oriented**, in the sense that they store sets of **facts** of some database application, in a form as close as possible to what will be presented to users as the result of query operations. In a relational database, sets of facts are assigned to different tables and the individual facts in each set correspond to tuples of the respective table.

The entire collection of facts that hold at a certain instant of time is a database state. A state is said to be valid if it obeys all the **static integrity constraints** prescribed for the database application. Not all transitions between valid states are in turn valid, but only those that obey the declared **transition integrity constraints.**

Transitions are effected by **update operations.** If we assume that a database starts with an initial (valid) "empty" state, then if a discipline is imposed on the updates it should be possible to ensure the

* On leave from the Pontifícia Universidade Católica do Rio de Janeiro

validity of both the states reached and of the transitions leading to them.

On the other hand, the **time dimension** is an important attribute of facts in some applications [BADW,Sn2]. We may want to maintain the history of a database, keeping all states rather than only the current one. To this end, a common solution is to include in each tuple a timestamp to express when the respective fact started to hold. Existing tuples are never deleted or modified.

Besides becoming an attribute of interest, time may be involved in several kinds of transition constraints, which imposes new demands on how updates should be disciplined.

In this paper we propose one such discipline. Taking the *abstract data type approach* [Gu,VF1,FN], it relies on a specification of the update operations pertaining to the database application on hand, and it involves a non-conventional **update-oriented** data structure. The major contribution of this discipline is that it leads to specifications whose style is declarative and that can be immediately used for prototyping. Section 2 describes the structure. Section 3 presents the specification techniques. These and their use within a prototyping tool are illustrated in section 4 on an example academic database application. Section 5 shows an extension for an alternative treatment of temporal databases. Section 6 contains the conclusion. An implementation of the tool, combining Prolog and SQL, is given in the Appendix.

2. UPDATE-ORIENTED ORGANIZATIONS

In the usual query-oriented structures, which are repositories of facts, records of operations executed are not part of the database, even though they are sometimes kept on a temporary basis, usually to help recovery. Such records are termed logs or **traces**.

The organization that we shall use originates from two separate research lines: the formal treatment of database states as traces, seen from an abstract data type viewpoint [BP,VF1], and an approach to temporal databases first suggested by Bubenko [Bu].

Abstract data type specifications are based on the properties of the operations associated with the type rather than on how its instances are represented. The more formal methods often denote an instance of a type by a Herbrand term, consisting of a sequence of update operations able to create the instance. Such terms correspond to the intuitive notion of traces. If we view a database application as a data

type whose instances are the database states, then it is natural to use traces to denote them.

However, to readily derive a running version of a formal specification, traces must be adapted to the record-driven data structures available in database management systems (DBMSs). To see how this can be done, consider the trace partially shown below of a state of an academic database application (we used a "flat" format rather than the more common but less readable nested format):

initiate().enroll(John,c1).enroll(Mary,c1). ... .transfer(John,c1,c2). ...

In a relational DBMS we can simply assign a table to each update operation; thus we would have tables named ENROLL, TRANSFER, etc. Their tuples are the lists of arguments, e.g. < John,c1 >, over which the operation is executed, plus a timestamp, marking the instant of execution of the operation, under the format year/month/day concatenated with the time of the day, as hour : minute : second. Notice that without the timestamp the tables would not fully capture a trace, since the order of execution of the constituent operations would be lost.

But we only store a tuple if the application of the update operation is **valid**, that is, if the established pre-conditions for its execution currently hold. Besides, we require that the operation be **productive** [VF2], i.e., none of the facts that it should add already holds and all those it should remove currently hold. Finally, triggers may be associated with some operations in order to restore integrity [Es].

Our structures are update-oriented, in the sense that, instead of storing the facts, we record which update operations were executed and when. The facts themselves are inferred from an analysis of the updates. In [Bu], updates are kept in base tables and facts correspond to views. In that reference a distinction is established between the time when an operation takes place in the outside world and the time when the operation is registered in the database. Here we take the simplifying assumption that these times coincide, which is realistic in some situations, especially in office automation environments. The duality and complementarity of query-oriented and update-oriented approaches to database specifications and their possible impact on database structuring were discussed in [VCF, page 419]. The work in [KS] is similar to ours but treats time in a different way, mentioned in section 5.

Apart from the convenience of update-oriented structures for specification and rapid prototyping purposes (which are our primary interest), we briefly indicate their possible advantages for operational usage. One can expect that update-oriented structures will tend to perform more efficiently for update-intensive applications. Also, they may compare favorably with query-oriented time-stamped tables whose tuples contain many items; to represent a change on a single item, one would normally add to the table a new time-stamped tuple, redundantly repeating all unchanged items (or would use some more compact, but also more complex, record structure). Besides, update-oriented structures, by definition, cannot suffer from update anomalies.

More important is to note that update-oriented structures are more informative, in that they allow to ask how a fact became true. This will be illustrated at the end of section 4, in a situation where there is more than one operation able to add a certain fact, and it is possible to find out which operation was performed in each case. However, more work is needed to compare them in detail with the various alternative structures for temporal databases.

3. DECLARATIVE SPECIFICATION OF UPDATES

Adopting the abstract data type approach, the specification of database applications where static and transition integrity constraints are preserved can be reduced to an adequate specification of a fixed set of update operations.

The decision for the abstract data type approach still leaves ample freedom to choose a notation. We would like to write the specifications in a declarative style. Also, it would be convenient if the formalism used were a programming language, so that a running prototype would readily result. Logic programming is therefore a natural candidate.

An operation is specified in terms of its syntax and semantics. The semantics of an update operation is expressed by the changes (adding or deleting facts) that it effects on database states. To enforce integrity constraints, the specification of an operation must include **pre-conditions** to its execution. The pre-conditions should be such that, if they hold at the current state, the execution of the operation cannot violate any constraint. An alternative to imposing pre-conditions is to permit that the operation be executed and then to **trigger** further operations so that at the final state reached all constraints are pre-

served. Combinations of pre-conditions and triggers are also possible in some cases.

The list below summarizes the items to be included in the specification of an operation. The first is syntactical and the others are semantic:

- its signature, i.e. the types of its parameters;
- what facts hold after its execution;
- what facts cease to hold after its execution;
- pre-conditions for its application;
- further operations to be triggered.

The specification of an operation O can be expressed as a set of Prolog clauses of the form:

- oper(O(D1,D2, ... ,Dn)) - the parameters of O are of types D1, D2, ... ,Dn
- added(F,O) - fact F is added by O
- deleted(F,O) - fact F is deleted by O
- valid(O,T) <- P - the execution of O is valid at time T if pre-condition P holds at T
- trigger(O,T) <- TR - the execution of O at T causes the execution of trigger TR

It is a non-trivial problem to "tune" pre-conditions, triggers and combinations thereof so that constraints are effectively enforced. We have discussed elsewhere formal methods to do so (see, in particular, [VF2]). In the next section we limit ourselves to an informal justification of the interplay of pre-conditions and triggers for an example application. We should also add that the analogy between time-stamped update operation tables on the one hand and traces on the other hand favors the use of the algebraic methods developed for proving properties of abstract data types [Gu,BP,VF1].

The specifications of all update operations for the database application on hand, expressed as Prolog clauses, capture the knowledge that we have about the particular application. To apply these clauses, however, we need a different kind of knowledge, which is often left implicit as unformalized assumptions. Those that we shall adopt here

are:

- An operation produces effects only if it is both
  - valid and
  - productive

  in the sense indicated in the previous section. Attempts to execute operations that are not valid or productive are ignored.
- In a sequence of operations (chained by the trigger mechanism) either all succeed or none has any effect. In other words, these sequences are treated as transactions [AV].
- The effects of an operation are exactly those declared in the clauses of the "added" and "deleted" predicates; in the new state all other facts remain unaffected. This assumption refers to the well-known frame problem [Ko].
- The facts that hold at a given instant of time are those that can be inferred from the operations able to add or delete them and which have been executed at the appropriate instants. This reflects the update-oriented strategy, whereby the execution of operations is recorded and the facts are deduced as their consequences.

These assumptions can be regarded as sentences about the clauses that describe particular applications. To formalize them, we resort to the meta-level programming features of Prolog [Ko]. Such techniques were first proposed in the above reference to face the frame problem in the context of plan-generation, but they are applicable in many other contexts [WMSW,VF2]. Based on the assumptions, we must express how to use the clauses specific to the application to:

- execute an operation;
- infer the result of queries from the executed operations;
- answer queries about the executed operations themselves.

The items above correspond to the three main meta-predicates that we need to run a database application:

- execute(O) - to execute a pre-defined operation O
- holds(F,T) - to verify the occurrence of fact F at instant T
- done(O,T) - to verify the execution of operation O at instant T

A short verbal description of how these meta-predicates work follows. To *execute* an operation op(a1,a2, ... ,an), one checks whether the operation is valid and productive at the current state; if not, the operation fails; otherwise, the clock is read for the timestamp ts and the tuple <ts,a1,a2, ... ,an> is added to table OP. Next, any associated

triggers are invoked and, if all triggered operations succeed (at all levels of chaining), an overall "commit" is executed, whilst if any operation fails the entire chain of operations suffers a "rollback". Chaining a sequence of updates through triggers and treating the sequence as a transaction, where either the entire sequence is applied or no effect results, is achieved by Prolog's recursion and backtracking machinery.

A fact f(b1,b2, ... ,bm) *holds* at an instant t, if the execution of an operation able to add it has been recorded in the respective table with a timestamp t' at most equal to t, and there is no record of an operation able to remove it with a timestamp ranging from t' to t. Since this is the only case where a fact is declared to hold and in view of the closed world assumption that underlies Prolog, the frame problem is avoided.

Retrieving information about the execution of operations, which is the task of the *done* meta-predicate, is of course immediate with update-oriented organizations.

Other meta-predicates are needed, for example, to express the meaning of "productive".

The set of these meta-predicates constitutes what we call the **generic part** in the architecture of our running prototyping tool, since it is intended for all (or at least for a broad class of) applications, and hence is written once and for all. The generic part works essentially as a meta-interpreter for the **specific part**, which consists of the clauses describing the operations of the application in question.

But the prototyping tool does not consist of Prolog alone. Clearly the strategy that we adopted will only have practical significance if it combines logic programming with some database management system. With logic programming only, we shall be limited to "simulations" in main storage, which may be unsatisfactory even for prototyping. On the other hand, having only a database management system, it would not be easy to include, in declarative format, the rules corresponding to pre-conditions, triggers, etc.

The implementation we use is a combination of two systems VM/PROLOG [VM] and SQL/DS [SQ]. VM/Prolog is close to the Edinburgh Prolog de facto standard; the differences are described in [WMSW, page 428]. The interface between VM/Prolog and SQL/DS was introduced in [CW] and is fully documented in [VM].

4. AN EXAMPLE DATABASE APPLICATION

We shall consider an academic database where the facts are:

- courses are offered
- students take courses

The integrity constraints are:

- static constraint

- r1 - a student can only take courses that are being offered

- transition constraint

- r2 - a new course can only start until March 15th of the current year;
- r3 - a student cannot reenter a course he has taken before;
- r4 - no course that stays without students after March 20th can continue to be offered;
- r5 - the number of courses taken by a student cannot decrease.

Assume that only the operations below are considered necessary:

- offer a course
- enroll a student in a course
- transfer a student from a course to another
- cancel a course

Besides these operations, we shall have operations to initiate and terminate a session, which resemble the familiar "open" and "close" commands of database management systems. The Appendix contains the specification of each operation in terms of the "oper", "added", "deleted", "valid" and "trigger" predicates. In this section we shall limit the discussion to pre-conditions (handled by "valid") and triggers.

Since we chose not to provide the usual operation "drop", to undo the enrollment of a student in a course, constraint r5 is trivially enforced, so that we are not required to impose pre-conditions or associate triggers with the operations for its sake.

However the other constraints do require such measures. Operation "offer" can only be executed until the deadline, and "enroll" only if the course involved is offered and the student has not taken it before, analogous pre-conditions being required for "transfer".

The "transfer" operation must have an associated trigger; if after March 20th a student A is transferred from a course C to some other one, A being the only student taking C, then course C should be automatically cancelled and the user simply warned that this was done.

This provision is not enough however to enforce r4, since the mere occurrence of the deadline might make it necessary to cancel courses that until then stayed (validly) without students. To cover this case, we must associate a trigger with "initiate", which suffices if we admit that each database session spans no more than a single day.

The "cancel" operation might have either a pre-condition:

- that no student is taking the course

or a trigger (the solution we chose here):

- that all students that are taking it be transferred to other courses.

Naturally this trigger must not be fully automatic, allowing instead that the user be asked to indicate to which course each student should be transferred.

The example can be much expanded. Of special interest are transition constraints based on elapsed time. For instance, we might rule that a student cannot be transferred to a new course before he has stayed in the course he is about to leave for at least one month, except naturally if the latter is being cancelled. To have this kind of constraint we must be able to perform arithmetic on dates, a feature that can be easily added by adapting an algorithm like the one in [Ro].

We now show how the example runs on the prototyping tool. As VM/Prolog goals are invoked, the appropriate SQL commands are synthesized and passed to SQL/DS for execution via the interface. Assume that at the beginning of a session on April 15th the following facts hold:

- course c1 is offered
- course c2 is offered
- course c3 is offered
- student John takes course c1
- student Mary takes course c1
- student Paul takes course c2

The execution of the goal statement:

<- initiate.

will cause course c3 to be cancelled, since it stayed without students after the March 20th deadline. Next, suppose we enter:

<- execute(cancel('c1')).

The trigger associated with this operation prompts us to indicate new courses for John and Mary; in this case, c2 is the only possible choice. With this, both students are transferred and the course is cancelled. To find which facts now hold, we enter:

<- now(T) & which(X,holds(X,T)).

being informed that course c2 (only) is offered and that Paul, John and Mary take this course.

And yet all previous states remain accessible. We can, say, retrieve the series of courses in which John has ever participated by:

<- which(C,holds(takes('John',C),T)).

Moreover, we can ask about the execution of operations. Besides learning that Paul, John and Mary now take c2, we can find that John and Mary were transferred to this course, instead of being originally enrolled. The answer to the query below consists of (two) pairs giving both the names of the students and the time (the same, in this case) of execution of each transfer operation.

<- now(T) & which([X,V], holds(takes(X,'c2'),T) & done(transfer(X,Y,'c2'),V)).

The database tables at the end of the session will contain:

OFFER

ts	course
87/02/20 14:30:51	c1
87/02/21 14:30:47	c2
87/02/27 16:25:03	c3

ENROLL

ts	student	course
87/03/03 15:20:23	John	c1
87/03/03 15:30:04	Mary	c1
87/03/03 15:35:32	Paul	c2

TRANSFER

ts	student	course1	course2
87/04/15 11:27:55	John	c1	c2
87/04/15 11:27:55	Mary	c1	c2

CANCEL

ts	course
87/04/15 11:26:30	c3
87/04/15 11:27:55	c1

To end the session, we type:

<- terminate.

5. INTERVAL-ORIENTED OPERATORS

In contrast to our approach to temporal databases, where we consider time instants, some authors prefer to deal with **time intervals** [Al,KS,Sn1]. It is possible to adopt one of the approaches and then to define, on its "primitive" operators, the operators pertaining to the other approach.

As an illustration, we defined the "precedes" and "overlaps" operators (see Appendix), based on [Sn1]. With "overlaps" we can, for example, characterize the set Y of colleagues of a student X, as being those students C who have taken a course Z during a time interval overlapping the time interval during which X took the same course Z.

The "overlaps" operator is denoted by "><" and used in infix format.

```
colleagues(X,Y) <-
  isall(Y, C, takes(C,Z) >< takes(X,Z) & ¬(C = X)).
```

In our sample session, if we enter:

```
<- colleagues('John',S).
```

the result will be the set {Mary,Paul}, since originally John and Mary were together in c1 and later in c2, the latter course still being taken by Paul.

6. CONCLUSION

Combining logic programming with large database applications [Sm] involves efficiency problems that must be solved before its use can become widespread. There is still, among others, the problem of designing user interfaces that be easy to use and hide the complexity of such systems.

Until now most proposals to use logic programming to handle databases claim that the main advantage to be gained is additional power to formulate queries, particularly by introducing recursion and inference. Our line of research takes the position that logic programming has an even broader scope, being also relevant to update activities and to express the rules governing database behaviour.

The declarative specification of updates together with update-oriented structures lead to the derivation of running prototypes. Further research is needed to assess the possible advantages of such structures in operational environments.

ACKNOWLEDGEMENT

We are grateful to our colleague Marco A. Casanova for carefully reading the paper and for several useful suggestions.

REFERENCES

[Al] J. F. Allen - Towards a general theory of action and time - Artificial Intelligence 23 (1984) 123-154.

[AV] S. Abiteboul and V. Vianu - Transactions in relational databases (preliminary report) - Proc. of the 10th International Conference on Very Large Data Bases (1984) 46-56.

[BADW] A. Bolour, T. L. Anderson, L. J. Dekeyser and H. K. T. Wong - The role of time in information processing: a survey - ACM SIGMOD Record 12, 3 (1982) 28-48.

[BP] W. Bartussek and D. Parnas - Using traces to write abstract specifications for software modules - T.R. 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 (1977) 93-118.

[CW] C. L. Chang and A. Walker - PROSQL: a Prolog programming interface with SQL/DS - in Expert database systems - L. Kerschberg (ed.) - The Benjamin/Cummings Publishing Company (1986) 233-246.

[Es] K. P. Eswaran - Specification, implementation and interaction of a trigger subsystem in an integrated database system - IBM Research Report RJ1820 (1976).

[FN] A. L. Furtado and E. J. Neuhold - Formal Techniques for Data Base Design - Springer-Verlag (1986).

[Gu] J. V. Guttag - Abstract data types and the development of data structures - SIGPLAN Notices 8, 2 (1976).

[Ko] R. Kowalski - Logic for problem solving - North-Holland (1979).

[KS] R. Kowalski and M. Sergot - A logic-based calculus of events - technical report - Department of Computing, Imperial College (1985).

[Ro] J.D. Robertson - R398 Tableless date conversion - Communications of the ACM, vol. 15, n. 10, (1972) p. 918.

[Sm] J. M. Smith - Logic programming and databases - in Expert Database Systems - L. Kerschberg (ed.) - The Benjamin/Cummings Publishing Company (1986) 3-15.

[Sn1] R. Snodgrass - The temporal query language TQuel - Transactions on Database Systems, vol. 12, n. 2 (1987).

[Sn2] R. Snodgrass - Research concerning time in databases: project summaries - ACM SIGMOD Record, vol. 15, n. 4 (1986) 19-39.

[SQ] SQL/Data System Application programming - doc. IBM SH24-5018-2 (1983).

[VCF] P. A. S. Veloso, J. M. V. de Castilho and A. L. Furtado - Systematic derivation of complementary specifications -

Proc. of the 7th International Conference on Very Large Data Bases (1981) 409-421.

[VF1] P. A. S. Veloso and A. L. Furtado - Stepwise construction of algebraic specifications - in Advances in Data Base Theory, vol. 2 - H. Gallaire, J. Minker and J. M. Nicolas (eds.) - Plenum (1984) 321-352.

[VF2] 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 (1985) 175-189.

[VM] VM/Programming in Logic - Program Description and Operations Manual - doc. IBM SB11-6374-0 (1985).

[WMSW] A. Walker, M. McCord, J. F. Sowa and W. G. Wilson - Knowledge systems and Prolog - Addison-Wesley (1987).

APPENDIX

```
/* UTILITIES */

which(X,Y) <- call(Y) & nl & prst('==> ')
  & writes(X) & fail().
which(X,Y) <- nl &
  prst('No (more) answers') & nl & nl & /().

list(X) <- axn(X,*,Y) & nl & writes(Y) & fail().
list(*) <- nl & nl.

isall(X,Y,Z) <-
  (compute(set,Y,Z,[],X) & /() | X = []).

forall(X,Y) <- ¬ (call(X) & ¬ call(Y)).

append([],X,X).
append([X!Y],Z,[X!W]) <- append(Y,Z,W).

on(X,[X!Y]).
on(X,[Y!Z]) <- on(X,Z).

cat(X,Y) <- cat1(X,Z) & st_to_li(Y,Z).
```

```
catl([X!Y],Z) <-
  (stringp(X) & W = X & / |
   st_to_at(W,X)) &
  st_to_li(W,L) &
  catl(Y,U) &
  append(L,U,Z).
catl([],[]).

/* GENERIC PART */

/* predicates directly accessible */

initiate() <-
  now(T) & /* in assembly-language */
  trigger(initiate(),T).

terminate() <-
  fin().

execute(X) <-
  now(T) &
  (exec(X,T) & sql('commit work',*) & /() |
   sql('rollback work',*) & fail()).

holds(X,T) <-
  added(X,Y) &
  inc_time(Y,V,O) &
  quest(O) &
  at_most(V,T) &
  ¬ (deleted(X,Z) &
     inc_time(Z,W,P) &
     quest(P) &
     before(V,W) &
     at_most(W,T)).

done(O,T) <-
  inc_time(O,T,P) &
  quest(P).

before(X,Y) <-
  (var(X) | var(Y)) & /() & fail().
before(X,Y) <-
  lt(X,Y).
```

```
at_most(X,Y) <-
  X = Y & /() | lt(X,Y).

display() <-
  now(T) &
  which(X,holds(X,T)).

/* auxiliary predicates */

exec(X,T) <-
  valid(X,T) &
  productive(X,T) &
  inc_time(X,T,O) &
  ins(O) &
  trigger(X,T).

productive(X,T) <-
  forall(added(Z,X), ¬ holds(Z,T)) &
  forall(deleted(W,X),holds(W,T)).

inc_time(X,T,O) <-
  X =.. [F!A] &
  O = [F,T!A].

ins(X) <-
  mk_ins(X,Y) &
  sql(Y,*).

mk_ins([F!A],Y) <-
  inc_sep(A,Z) &
  cat(Z,W) &
  Y := 'insert into ' || F ||
       ' values(' || W || ')'.

inc_sep([X],[X]) <- numb(X) & /().
inc_sep([X],['''',X,'''']).
inc_sep([X!Y],[X,',',!Z]) <-
  numb(X) & /() & inc_sep(Y,Z).
inc_sep([X!Y],['''',X,''',',!Z]) <-
  inc_sep(Y,Z).

quest(X) <-
  mk_cons(X,Y,Z) &
  sql(Y,Z,*).
```

```
mk_cons([F!A],Y,A) <-
  oper(Z) &
  Z =.. [F!B] &
  comb(B,A,S) &
  Y := 'select * from ' || F || S.

comb(nil,nil,'').
comb([X!Y],[Z!W],S) <-
  ¬ var(Z) &
  comb(Y,W,T) &
  (numb(Z) & Q = '' & /() | Q = '''') &
  (T == '' & A := ' where ' & /() |
   A := ' and ') &
  S := T || A || X || ' = ' ||
       Q || Z || Q.
comb([X!Y],[Z!W],S) <-
  var(Z) & comb(Y,W,S).

/* SPECIFIC PART -          */
/* academic database example */

oper(offer(ts,course)).
oper(enroll(ts,student,course)).
oper(transfer(ts,student,course1,course2)).
oper(cancel(ts,course)).

added(offered(X),offer(X)).
added(takes(X,Y),enroll(X,Y)).
added(takes(X,Z),transfer(X,Y,Z)).

deleted(offered(X),cancel(X)).
deleted(takes(X,Y),transfer(X,Y,Z)).

valid(offer(X),T) <-
  substring(T,V,3,5) &
  before(V,'03/15').
valid(enroll(X,Y),T) <-
  holds(offered(Y),T) &
  ¬ holds(takes(X,Y),V).
valid(transfer(X,Y,Z),T) <-
  holds(offered(Z),T) &
  ¬ holds(takes(X,Z),V).
valid(cancel(X),T).
```

```
trigger(initiate(),T) <-
  substring(T,V,3,5) &
  before(V,'03/20') & /() |
  forall(holds(offered(Z),T) &
         ¬ holds(takes(W,Z),T),
         exec(cancel(Z),T) &
         M := 'course ' || Z ||
              ' cancelled' &
         nl & prst(M) & nl).
trigger(offer(X),T).
trigger(enroll(X,Y),T).
trigger(transfer(X,Y,Z),T) <-
  ¬ holds(offered(Y),T) & /() |
  substring(T,V,3,5) &
  before(V,'03/20') & /() |
  holds(takes(S,Y),T) & ¬ (S = X) & /() |
  M := 'course ' || Y ||
       ' cancelled' & nl & prst(M) & nl &
  exec(cancel(Y),T).
trigger(cancel(X),T) <- nl &
  forall(holds(takes(Y,X),T),
    W := 'indicate another course for ' || Y &
         prst(W) & nl &
         read(Z) &
         exec(transfer(Y,X,Z),T)).

/* INTERVAL-ORIENTED */
/* TEMPORAL OPERATORS */

started(X,T) <-
  added(X,O) &
  inc_time(O,T,F) &
  quest(F).

ended(X,T) <-
  deleted(X,O) &
  inc_time(O,T,F) &
  quest(F).

interval(X,S,E) <-
  started(X,S) &
  (ended(X,E) |
   now(E)) &
```

```
  before(S,E) &
  ¬ (ended(X,F) &
    before(S,F) &
    before(F,E)).

precedes(X,Y) <-
  interval(X,SX,EX) &
  interval(Y,SY,EY) &
  before(EX,SY).

overlaps(X,Y) <-
  interval(X,SX,EX) &
  interval(Y,SY,EY) &
  before(SX,EY) &
  before(SY,EX).

op("><",rl,60).
op("<<",rl,60).
<<(X,Y) <- precedes(X,Y).
><(X,Y) <- overlaps(X,Y).
```