PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 11/11

IDB - An Environment for Experimenting with Intelligent Database-Resident Information Systems

Antonio L. Furtado

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 11/11 ISSN 0103-9741
Editor: Prof. Carlos José Pereira de Lucena September, 2011

IDB - An Environment for Experimenting with Intelligent Database-Resident Information Systems

Antonio L. Furtado

furtado@inf.puc-rio.br

Abstract: The IDB tool aims to provide an environment for the conceptual specification and testing of intelligent information systems. The IDB environment comprises three different options. Initially, it runs in the workspace of a Prolog program. Next, still under the control of this program, it operates upon database-resident relational tables via an ODBC interface. Finally leading to an operational stage, it provides automatically generated stored procedures, which enforce integrity and collect execution traces for continuing maintenance and redesign purposes. The tool relies on plan-generation to conduct experiments, both in main memory and over the Oracle database.

Keywords: Conceptual Specification, Entity-Relationship Model, Relational Databases, Plan Generation, Simulation, Logic Programming, Constraint Programming.

Resumo: A ferramenta IDB visa criar um ambiente para a especificação conceitual e teste de sistemas inteligentes de informação. O ambiente IDB compreende três diferentes opções. Inicialmente, roda no espaço de trabalho de um programa em Prolog. Em seguida, ainda sob o controle desse programa, opera sobre tabelas relacionais residentes em banco de dados através de uma interface ODBC. Finalmente levando a um estágio operacional, fornece procedimentos armazenados, os quais garantem a integridade e coletam traços de execução, visando a manutenção contínua e o reprojeto. A ferramenta se serve da geração de planos para conduzir os experimentos, tanto em memória principal quanto sobre o banco de dados em Oracle.

Palavras-chave: Especificação Conceitual, Modelo Entidades-Relacionamentos, Bancos de Dados Relacionais, Geração de Planos, Simulação, Programação em Lógica, Programação por Restrições.

In charge of publications

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br

1. Introduction

The objective of this paper is to document our initial effort to provide an environment, supported by the **Intelligent Data Base** (IDB) plan-driven tool, rich enough to support the development of intelligent database-resident information systems. To behave "intelligently", such systems should have:

1. online access to detailed *knowledge* about the application domain [FC]
2. one or more *reasoning* capabilities, such as interpretation, diagnosis, prediction, planning and monitoring [Wa]
3. running records of its use by the various authorized agents, to support double-loop *learning* [Mo]

As a preliminary step, the system must be specified at the conceptual level. Our specification method comprises three schemas: the *static schema*, describing the entity classes and their properties in terms of the Entity-Relationship (ER) model [BCN], the *dynamic schema*, defining the domain-specific operations in a declarative style by their pre-conditions and post-conditions according to the STRIPS proposal [FN], and the *behavioural schema* – not to be treated here (cf. [FC]) –, wherein goal inference rules are formulated, indicating what goals the agents involved would be expected to pursue when certain situations arise. The entire specification, expressing domain knowledge in clausal notation, is kept accessible on line, thus serving the first requirement above.

For the first option, all data, also expressed in clausal form, is manipulated in workspace memory by the IDB program (written in SWI-Prolog) that handles the environment. With the help of a planner, which implements one of the capabilities enumerated in requirement 2, simulation experiments can be run through the execution of the generated plans (which consist of sequences of the pre-defined operations). As a side-effect, the plan sequence is appended to a list representing the *execution log* (also in the main memory Prolog workspace), whereby requirement 3 is also served.

A second option is supplied, utilizing an ODBC interface, through which the Prolog interpreter is allowed to invoke Oracle commands to access data represented in relational tables residing in secondary memory. To take advantage of this facility, the IDB program includes a compiler that converts the operations, previously formulated in declarative style as mentioned above, into a procedural format, wherein the pre-conditions give origin to `select` clauses, and the post-conditions to `insert`, `delete` or `update` clauses. Such clauses, programmed in IDB as Prolog predicates, call the corresponding Oracle commands via the ODBC interface. Now, whenever a flag attached to this option is turned on, the same operation sequences generated by the planner will be executed as IDB-controlled transactions over the actual database tables, with the side-effect of registering the execution in a log also represented as a relational table in the Oracle database.

A third option is enabled by a second compiler supplied by IDB, which converts the procedural format operations into SQL stored procedures, to be executed directly (i.e. outside the IDB environment) by some appropriate Oracle shell. Assembled in packages, the stored procedures should constitute an effective way to enforce database integrity, since they are created to reflect the pre-conditions/ post-conditions interplay specified at the conceptual design stage. Indeed, the policy of granting user access to such packages, but

not to the primitive data manipulation commands, is in conformance with a strict *abstract data type* discipline [GG].

Although moving to this third option would seem to indicate the end of the design phase, designers and end users are aware that any system needs maintenance, often involving extensive redesign. For this purpose, one may revert, whenever convenient, to the first two Prolog-driven options, to consider possible revisions of the conceptual schemas and evaluate their consequences by once again performing simulated experiments. Note in particular that, by consulting the log of executed operations, it is possible to learn (as anticipated in requirement 3) from the practical use of the system.

Sections 2, 3 and 4 illustrate, on the basis of a small example, how the environment operates for the three options. Section 5 contains concluding remarks. The entire Prolog program implementing IDB is shown in the Appendix, and the relevant features of the plan-generating program are briefly reviewed in section 2.2.

2. First option: Prolog only

2.1. Conceptual specification in clausal notation

The example deals with an academic database. The entity classes are `student`, `course`, and `program`. The identifying attributes are, respectively: `student_name`, `course_name`, and `program_name`. The other attributes, in this simple example just one for each entity class, are, respectively, `credits_won`, `credits` and `requirement`. Three relationships associate students with courses: `takes`, `has_finished`, and `has_dropped`. Relationship `graduated_in` associates students with programs.

The Prolog clauses are shown below, noting that the identifying attributes are introduced in the entity clauses themselves, whereas the other attributes appear in separate attribute clauses. The relationship clauses indicate, enclosed in square brackets, the participating entity classes. The current version of IDB is restricted to binary relationships.

% static schema - classes of facts

```
entity(student,student_name).
attribute(student,credits_won).
entity(course,course_name).
attribute(course,credits).
entity(program,program_name).
attribute(program,requirement).
relationship(takes,[student,course]).
relationship(has_finished,[student,course]).
relationship(has_dropped,[student,course]).
relationship(graduated_in,[student,program]).
```

Students begin with zero `credits_won`, the value of this attribute being incremented by the credits of the courses that they manage to finish successfully. Different academic programs may be offered, which, in this oversimplified example, differ only in terms of the total credits required. For graduation in a program, the number of credits won by the student must reach, exactly, the program's requirement.

Here is a possible initial *state*, containing a few *facts* also in clause format, which are instances of the specified schema; notice the absence of students:

```
course('Art').
credits('Art',2).
course('Semiotics').
credits('Semiotics',3).
course('Design').
credits('Design',1).
program('Alpha').
requirement('Alpha',5).
program('Beta').
requirement('Beta',4).
```

State changes will be limited to those that can be performed by a pre-defined repertoire of operations, specified in terms of their pre-conditions and post-conditions (effects) according to the STRIPS proposal. The IDB planning and execution algorithms enforce a discipline that in most cases simplifies the definition of the operations. First of all, it is not mandatory (although it may be done, in order to guarantee the instantiation of certain parameters) to declare as a pre-condition that a fact to be added by an operation must not already hold at the current state, or that a fact to be deleted does not hold.

In addition, an operation is caused to fail if for any reason it cannot produce all the specified effects, except for those that, in the clausal notation, are prefixed with a "/". This notation serves to indicate that an effect of the form `/F` (or `/(not F)`) will be performed "if needed"; in particular, if `F` already holds (or already does not hold) its addition (or deletion) will simply be recognized as unnecessary.

The planner interprets the pre-conditions both as tests for the applicability of an operation `Op` and, in case of failure, as sub-goals to be fulfilled by operations to be introduced before `Op` in the plan. This recursive treatment of pre-conditions as sub-goals constitutes the *backward chaining* strategy, on which many planning algorithms (including the one used by IDB) are based. However not all failed pre-conditions are so treated; a second use of the "/" notation is to mark positive or negative pre-conditions that are to be handled exclusively as tests. Thus, if a positive `/F` or negative `/(not F)` pre-condition for an operation `Op` fails, operation `Op` is simply not included in the plan. It must also be stressed, again in view of backward chaining, that such tests are *delayed* until the planning algorithm, after returning from the recursive calling sequence, is about to append `Op` to the plan.

An especially powerful feature that allows the planner to pursue goals or sub-goals involving numerical expressions is constraint programming, which is conveniently provided by SWI-Prolog. In IDB, we utilize the clpd (constraint logical programming over finite domains) library module. Constraint programming also relies on delayed evaluation, since the numerical expressions cannot be computed until all variables have been instantiated.

Before showing the clausal specification of the operations, we give an informal description. As usual, the existence of integrity constraints and of some sort of regulatory procedures is assumed. For example, it is prescribed that each instance `S` of the student entity class is created when `S` is for the first time enrolled in some course `C`. The assumed regulations further establish that `S` can drop a course `C` only by being transferred to some

other course not taken before (i.e. that `S` has neither finished nor dropped before). As will be noted, the two uses of the "/" notation are conveniently illustrated in the `transfer(S,C1,C2)` operation in view of the above requirements: information about someone being a student, initially with 0 credits won, is added only if needed (i.e. if not already present as the result of a previous enrollment); the pre-conditions requiring that `S` has not previously finished nor dropped `C` are mere tests, not sub-goals ─ contrary to the `course(C)` pre-condition, whose presence without "/" tells the planner that a course may be created on demand, i.e. if there are students willing to take it.

Another noteworthy example is the pre-condition specification of `pass(S,C,T1,T2)`, wherein the relation between `T1` and `T2` is expressed in constraint programming terms. An effect of passing a course `C` is that the number of credits of `C` is summed to `T1` (the credits accumulated before) to yield the new total `T2`. Clearly the sum must wait for the (delayed) evaluation of `T1`. Because constraint programming interprets numerical expressions as goals to be satisfied, this specification of `pass(S,C,T1,T2)` is able to lead the planner to compose a sequence to which operation `receive_degree(S,P)`, which depends on `S` achieving the number of credits required by program `P`, is successfully appended.

```
- operation offer(C,N) - offer course C with N credits
pre-conditions: course C is not already offered, even with a different
number of credits
post-conditions: clauses course(C) and credits(C,N) are added to the
workspace

- operation enroll(S,C) - enroll student S in course C
pre-conditions: course C is currently offered, and student S has neither
finished nor dropped it before
post-conditions: clause takes(S,C) is added; the clause student(S) is
added if this is the first enrollment of S in a course

- operation transfer(S,C1,C2)
pre-conditions: student S is taking C1, course C2 is currently offered,
and student S has neither finished nor dropped it before
post-conditions: clause takes(S,C1) is deleted and clauses dropped(S,C1)
and takes(S,C2) are added

- operation cancel(C)
pre-conditions: no student is currently taking C
post-conditions: clauses course(C) and credits (C,N) are deleted

- operation change_cr(C,N1,N2)
pre-conditions: C is being offered with N1 credits
post-conditions: clause credits(C,N1) is deleted and clause credits(C,N2)
is added

- operation pass(S,C,T1,T2)
pre-conditions: S is taking C, which gives N credits, and has completed
T1 credits; T2 is obtained by summing N to T1
post-conditions: clauses takes(S,C) and credits_won(S,T1) are deleted,
and clauses credits_won(S,T2) and has_finished(S,C) are added

- operation create_program(P,R)
pre-conditions: program P is not aleady being offered, with any number of
total credits required
```

```
post-conditions: clauses program(P) and requirement(P,R) are added

- operation receive_degree(S,P)
pre-conditions: student S has obtained exactly the total number of
credits required for the completion of program P and is no longer taking
courses
post-conditions: clause graduated_in(S,P) is added
```

The full specification of the dynamic schema in clause format is shown below.

```
% dynamic schema - operations to produce events

operation(offer(C,N)).
added(course(C),offer(C,N)).
added(credits(C,N),offer(C,N)).
precond(offer(C,N),/(not(course(C)))).

operation(enroll(S,C)).
/added(student(S),enroll(S,C)).
/added(credits_won(S,0),enroll(S,C)) :-
  not credits_won(S,_).
added(takes(S,C),enroll(S,C)).
precond(enroll(S,C),
  (course(C),
   /(not has_finished(S,C)),/(not has_dropped(S,C)))).

operation(transfer(S,C1,C2)).
deleted(takes(S,C1),transfer(S,C1,C2)).
added(has_dropped(S,C1),transfer(S,C1,C2)).
added(takes(S,C2),transfer(S,C1,C2)).
precond(transfer(S,C1,C2),
  (course(C2),/takes(S,C1),
   /(not has_finished(S,C2)),/(not has_dropped(S,C2)))).

operation(cancel(C)).
deleted(course(C),cancel(C)).
deleted(credits(C,N),cancel(C)).
precond(cancel(C),
  not takes(S,C):(student(S),takes(S,C))).

operation(change_cr(C,N1,N2)).
deleted(credits(C,N1),change_cr(C,N1,N2)).
added(credits(C,N2),change_cr(C,N1,N2)).
precond(change_cr(C,N1,N2),credits(C,N1)).

operation(pass(S,C,T1,T2)).
deleted(credits_won(S,T1),pass(S,C,T1,T2)).
deleted(takes(S,C),pass(S,C,T1,T2)).
added(has_finished(S,C),pass(S,C,T1,T2)).
added(credits_won(S,T2),pass(S,C,T1,T2)).
precond(pass(S,C,T1,T2),(credits_won(S,T1),takes(S,C))) :-
    credits(C,N),
    T2 #= T1 + N, T1 #>= 0.

operation(create_program(P,R)).
added(program(P),create_program(P,R)).
```

```
added(requirement(P,R),create_program(P,R)).
precond(create_program(P,R),/(not program(P))).

operation(receive_degree(S,P)).
added(graduated_in(S,P),receive_degree(S,P)).
precond(receive_degree(S,P),
  (requirement(P,T),credits_won(S,T),/(not takes(S,_)))).
```

2.2. Plan generation and execution in workspace memory

Once the static schema and the dynamic schema have been specified, and a (possibly empty) initial state has been provided in workspace memory, one can perform state transitions through the `execute` predicate. For example, `execute(enroll('Bea','Art'))` would have the double effect of enrolling Bea in the Art course and, since this would be her first enrollment, of registering her as a student.

But the crucial resource for experimenting with a specification is the plan-generator. Given a goal expression as its first parameter the `plans` predicate will substitute a suitable plan for the variable placed at the second parameter position. The example below shows how Bea could fulfill the requirements to earn a degree in program Alpha, assuming that she is already taking the Art course.

```
?- plans(graduated_in('Bea','Alpha'),Plan), narrate(Plan).

student Bea enrolled in course Semiotics.
student Bea, having passed course Semiotics, has a total of 3 credits.
student Bea, having passed course Art, has a total of 5 credits.
student Bea has graduated in program Alpha.

Plan = start=>enroll(Bea, Semiotics)=>pass(Bea, Semiotics, 0, 3)=>pass(Bea, Art, 3,
5)=>receive_degree(Bea, Alpha) .
```

The `execute` predicate also works on plans, but another predicate, `goal_exec`, has the advantage of exhibiting the alternative plans that have been found to achieve the given goal expression, and allowing the user to choose the alternative to be executed. As goal expression, let us consider some course for Joe, different from those that Bea is taking (assuming that the plan whereby Bea would graduate in program Alpha has not been executed, so that she is still taking the Art course).

```
?- goal_exec((takes('Joe',C),/(not takes('Bea',C)))).

student Joe enrolled in course Semiotics.

choose one: yes/no/stop - no.

student Joe enrolled in course Design.

choose one: yes/no/stop - yes.
C = Design .
```

Successive executions of single operations and of entire plans has the side-effect of updating a `log` clause, to which the `narrate` predicate (with no argument) can be applied:

```
?- narrate.

student Bea enrolled in course Art.
student Joe enrolled in course Design.
```

3. Second option: Prolog, ODBC interface, Oracle

The second option is still performed in Prolog, but now a real Oracle database is involved instead of the main memory workspace. A first step is required, which is not automatic in the current implementation of IDB (though it is not difficult to add this feature, as we did in [TCF]). One must write and then run a script (i.e. a ".sql" program) in SQL*Plus, in order to create relational tables corresponding to the specified entities, attributes and binary relationships:

```
student(student_name, credits_won)
course(course_name, credits)
takes(student_name, course_name)
has_finished(student_name, course_name)
has_dropped(student_name, course_name)
program(program_name, requirement)
graduated_in(student_name, program_name)
```

In addition to these *information tables*, two other tables must be created for data administration purposes:

```
ref(r)
log(ref,ts,event).
```

The REF table stores the granted references, which are distinct positive integers that will serve as identifiers for the *IDB-transactions*. This special type of transaction controlled by IDB allows to characterize a potentially long process, which may be resumed in future sessions. The LOG table registers the execution of the operations, giving, for each execution, the reference of the transaction to which it belongs, the time-stamp read from the system's clock, and the event (name and parameters of the operation executed).

The script program used for our example is given below:

```
CREATE TABLE  "STUDENT"
   ( "STUDENT_NAME" VARCHAR2(100),
     "CREDITS_WON" NUMBER
   );

CREATE TABLE  "COURSE"
   ( "COURSE_NAME" VARCHAR2(100),
     "CREDITS" NUMBER
   );
```

```
CREATE TABLE  "TAKES"
   ( "STUDENT_NAME" VARCHAR2(100),
     "COURSE_NAME" VARCHAR2(100)
   );

CREATE TABLE  "HAS_FINISHED"
   ( "STUDENT_NAME" VARCHAR2(100),
     "COURSE_NAME" VARCHAR2(100)
   );

CREATE TABLE  "HAS_DROPPED"
   ( "STUDENT_NAME" VARCHAR2(100),
     "COURSE_NAME" VARCHAR2(100)
   );

CREATE TABLE  "PROGRAM"
   ( "PROGRAM_NAME" VARCHAR2(100),
     "REQUIREMENT" NUMBER
   );

CREATE TABLE  "GRADUATED_IN"
   ( "STUDENT_NAME" VARCHAR2(100),
     "PROGRAM_NAME" VARCHAR2(100)
   );

CREATE TABLE  "LOG"
   ( "REF" NUMBER,
     "TS" VARCHAR2(100),
     "EVENT" VARCHAR2(100)
   );

CREATE TABLE  "REF"
   ( "R" NUMBER
   );
```

The IDB program, connected from then on with Oracle via the ODBC interface, is able to check the structure of the information tables:

?- table_info.

```
GRADUATED_IN        -  PROGRAM_NAME (varchar2)
GRADUATED_IN        -  STUDENT_NAME (varchar2)
COURSE              -  CREDITS (number)
COURSE              -  COURSE_NAME (varchar2)
HAS_DROPPED         -  COURSE_NAME (varchar2)
HAS_DROPPED         -  STUDENT_NAME (varchar2)
HAS_FINISHED        -  COURSE_NAME (varchar2)
HAS_FINISHED        -  STUDENT_NAME (varchar2)
PROGRAM             -  REQUIREMENT (number)
PROGRAM             -  PROGRAM_NAME (varchar2)
STUDENT             -  CREDITS_WON (number)
STUDENT             -  STUDENT_NAME (varchar2)
TAKES               -  COURSE_NAME (varchar2)
TAKES               -  STUDENT_NAME (varchar2)
```

The next step, however, is totally automatic. IDB implements in Prolog, as augmented with a number of ODBC special predicates, the select, insert, delete and update commands of the SQL language. By entering the line:

```
:- compile_ops.
```

a *procedural* version of the operations is compiled from the previously introduced declarative specification. Their main components are predicates that were implemented to execute, via ODBC, the basic SQL commands. The pre-conditions are now expressed in terms of `select` calls, and the post-conditions (effects) in terms of `insert`, `delete` and `update` calls.

```
offer(A, B) :-
        ref(C),
        not select(course(A)),
        insert(course(A, B)),
        ins_log(C, offer(A, B)).

enroll(B, A) :-
        ref(C),
        select(course(A)),
        not select(has_finished(B, A)),
        not select(has_dropped(B, A)),
        (   select(student(B))
        ;   not select(student(B)),
            insert(student(B, 0))
        ),
        insert(takes(B, A)),
        ins_log(C, enroll(B, A)).


transfer(B, C, A) :-
        ref(D),
        select(course(A)),
        select(takes(B, C)),
        not select(has_finished(B, A)),
        not select(has_dropped(B, A)),
        delete(takes(B, C)),
        insert(has_dropped(B, C)),
        insert(takes(B, A)),
        ins_log(D, transfer(B, C, A)).

cancel(A) :-
        ref(B),
        not select(takes(_, A)),
        delete(course(A, _)),
        ins_log(B, cancel(A)).

change_cr(A, B, C) :-
        ref(D),
        select(credits(A, B)),
        update(course(A, B=>C)),
        ins_log(D, change_cr(A, B, C)).
```

```
pass(A, C, B, D) :-
        ref(F),
        select(credits_won(A, B)),
        select(takes(A, C)),
        select(credits(C, E)),
        D#=B+E,
        update(student(A, B=>D)),
        delete(takes(A, C)),
        insert(has_finished(A, C)),
        ins_log(F, pass(A, C, B, D)).

create_program(A, B) :-
        ref(C),
        not select(program(A)),
        insert(program(A, B)),
        ins_log(C, create_program(A, B)).

receive_degree(B, A) :-
        ref(D),
        select(requirement(A, C)),
        select(credits_won(B, C)),
        not select(takes(B, _)),
        insert(graduated_in(B, A)),
        ins_log(D, receive_degree(B, A)).
```

Notice that the first line in the body of the operation clauses asks for the current transaction reference, which must have been previously set by entering a line such as:

```
:- grant_ref(126).
```

with the side effect of adding the tuple [126] to the REF table. Now, if to enroll Bea in the Art course one enters:

```
:- enroll('Bea','Art').
```

the tuples ['Bea',0] and ['Bea','Art'] will be inserted, respectively, into tables STUDENT and TAKES, and the tuple [126,2010/09/20/10/31/17/900,enroll('Bea','Art')] will be inserted into the LOG table. A transaction can be continued in another day by entering `grant_ref` with the same reference, in which case the reference (already registered in the REF table) will again become current.

As one might expect, the predicates implemented in correspondence with the SQL commands are perfectly apt to query and modify the Oracle tables. Any of their parameters can be either a constant value or a variable, in which case it will be instantiated as a result of execution. This makes the usage of the `select` predicate particularly convenient, since the Prolog implementation takes care of determining the proper formulation of the select/from/where parts of the SQL select command to be executed.

However, for modifying the data, both in workspace and, even more crucially, in actual databases, it is most convenient to adopt the abstract datatype discipline [GG]. Whereas queries can safely be done (except for authorization rules, that may be later prescribed) via the `select` predicate, changes to the data should be restricted to the pre-defined repertoire

of operations whose pre-conditions and post-conditions have been adjusted in view of integrity constraints and prevailing regulations.

A sequence of operations belonging to the same IDB-transaction can be executed as a single unit by using the "$" operator (in special, notice the use of variables in the calls to operation `pass`, to be instantiated by selection and numerical computation as determined in the body of the operation's clause):

```
$ pass('Bea', 'Semiotics',X1, Y1), pass('Bea','Art',X2,Y2),
       receive_degree('Bea','Alpha')).
X1 = 0  Y1 = 3  X2 = 3 Y2 = 5
```

where by "single unit" we mean that, if any of these three operations fails, those previously executed will be rolled back. It might happen, for instance, that the sum of credits would not add up to what program Alpha requires. Only if no such problems occur, the appropriate commit commands with be issued.

The planner is still available to work upon the information now stored in the Oracle database. To redirect the planner, it suffices to enter the line "`:- env_option(db).`". (To return to the workspace regime, the required parameter is `memory` or simply `mem`).

The LOG table can be inspected at any time, either in the Prolog environment by entering:

```
?- select_log(R,Ts,Ev).
```

or via a select command over the **Oracle Database XE** shell, with the output shown below:

REF	TS	EVENT
123	2010/09/20/10/12/44/838	create_program(Alpha, 5)
123	2010/09/20/10/12/44/869	offer(Art, 2)
123	2010/09/20/10/12/44/885	offer(Semiotics, 3)
124	2010/09/20/10/16/46/072	create_program(Beta, 4)
124	2010/09/20/10/16/46/088	offer(Design, 1)
125	2010/09/20/10/29/52/322	enroll(Joe, Art)
125	2010/09/20/10/29/52/353	enroll(Joe, Design)
125	2010/09/20/10/29/52/400	enroll(Moe, Design)
126	2010/09/20/10/31/17/900	enroll(Bea, Art)
126	2010/09/20/10/31/17/932	enroll(Bea, Semiotics)
127	2010/09/20/10/34/25/478	transfer(Joe, Design, Semiotics)
127	2010/09/20/10/34/25/525	transfer(Moe, Design, Art)
128	2010/09/20/10/36/03/603	cancel(Design)
129	2010/09/20/10/46/28/916	pass(Bea, Semiotics, 0, 3)
129	2010/09/20/10/46/28/947	pass(Bea, Art, 3, 5)
129	2010/09/20/10/46/29/010	receive_degree(Bea, Alpha)

A more user-friendly pseudo-natural language format is provided, again in the Prolog environment, by typing:

```
?- show_trace.

ref: 123
program Alpha is open, requiring a total of 5 credits.
course Art created with 2 credits.
course Semiotics created with 3 credits.

ref: 124
program Beta is open, requiring a total of 4 credits.
course Design created with 1 credits.

ref: 125
student Joe enrolled in course  Art.
student Joe enrolled in course  Design.
student Moe enrolled in course  Design.

ref: 126
student Bea enrolled in course  Art.
student Bea enrolled in course  Semiotics.

ref: 127
student Joe transferred from course  Design to course  Semiotics.
student Moe transferred from course  Design to course  Art.

ref: 128
course Design cancelled.

ref: 129
student Bea, having passed course  Semiotics, has a total of 3 credits.
student Bea, having passed course  Art, has a total of 5 credits.
student Bea has graduated in program  Alpha.
```

4. Third option: Oracle alone

The first two options are intended for the specification tasks and for the performance of simulation runs, with the vital help of the plan-generator. For operational usage, the common practice is to fully employ some commercially available DBMS, such as Oracle, possibly together with a suitable host language, such as C.

To facilitate the transition, the IDB tool has a second compiler to translate from the procedural version of the operations, created by the first compiler mentioned in the preceding section, into Oracle *stored procedures*.

These procedures will run over the same tables, and will equally update the LOG table. Two advantages accrue from adopting this policy:

- the constraint-preserving abstract data type discipline will still be enforced if the procedures are incorporated in Oracle packages, and users will not be granted the right to update the database except through such packages;

- the story of the database usage kept in the LOG table will help towards the maintenance and eventual redesign of the system, especially by returning to the early stages of the IDB plan-driven environment.

The compiler generates the procedures displayed below, and stores them in a text file:

```
:- print_procs_txt.

create procedure offer(rn number, co varchar2, cr number) is
 found number;
 ev varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select count(*) into found
 from course
 where course_name = co;
 if found > 0 then return;
 end if;
 insert into course
 values (co, cr);
 ev := 'offer(' || co || ',' || cr || ')';
 ins_log(rn,ev);
 commit;
end offer;

create procedure enroll(rn number, s varchar2, c varchar2) is
 found number;
 ev varchar(100);
 course_name_c varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select course_name into course_name_c
 from course
 where course_name = c;
 select count(*) into found
 from has_finished
 where student_name = s and course_name = c;
 if found > 0 then return;
 end if;
 select count(*) into found
 from has_dropped
 where student_name = s and course_name = c;
 if found > 0 then return;
 end if;
 select count(*) into found
 from student
 where student_name = s;
 if found = 0 then
 insert into student
 values (s, 0);
 end if;
 insert into takes
```

```
 values (s, c);
 ev := 'enroll(' || s || ',' || c || ')';
 ins_log(rn,ev);
 commit;
end enroll;

create procedure transfer(rn number, s varchar2, c1 varchar2, c2
varchar2) is
 found number;
 ev varchar(100);
 course_name_c1 varchar(100);
 course_name_c2 varchar(100);
 student_name_s varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select course_name into course_name_c2
 from course
 where course_name = c2;
 select student_name, course_name into student_name_s, course_name_c1
 from takes
 where student_name = s and course_name = c1;
 select count(*) into found
 from has_finished
 where student_name = s and course_name = c2;
 if found > 0 then return;
 end if;
 select count(*) into found
 from has_dropped
 where student_name = s and course_name = c2;
 if found > 0 then return;
 end if;
 delete from takes
 where student_name = s and course_name = c1;
 insert into has_dropped
 values (s, c1);
 insert into takes
 values (s, c2);
 ev := 'transfer(' || s || ',' || c1 || ',' || c2 || ')';
 ins_log(rn,ev);
 commit;
end transfer;

create procedure change_cr(rn number, co varchar2, cr1 number, cr2
number) is
 found number;
 ev varchar(100);
 course_name_co varchar(100);
 credits_cr1 number;
begin
 select r into found
 from ref
 where r = rn;
 select course_name, credits into course_name_co, credits_cr1
 from course
 where course_name = co and credits = cr1;
```

```
 update course
 set credits = cr2
 where course_name = co and credits = cr1;
 ev := 'change_cr(' || co || ',' || cr1 || ',' || cr2 || ')';
 ins_log(rn,ev);
 commit;
end change_cr;

create procedure cancel(rn number, c varchar2) is
 found number;
 ev varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select count(*) into found
 from takes
 where course_name = c;
 if found > 0 then return;
 end if;
 delete from course
 where course_name = c;
 ev := 'cancel(' || c || ')';
 ins_log(rn,ev);
 commit;
end cancel;

create procedure pass(rn number, st varchar2, co varchar2, t1 number, t2
in out number) is
 found number;
 ev varchar(100);
 course_name_co varchar(100);
 credits_A number;
 credits_won_t1 number;
 student_name_st varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select student_name, credits_won into student_name_st, credits_won_t1
 from student
 where student_name = st and credits_won = t1;
 select student_name, course_name into student_name_st, course_name_co
 from takes
 where student_name = st and course_name = co;
 select course_name, credits into course_name_co, credits_A
 from course
 where course_name = co;
 t2:=t1+credits_A;
 update student
 set credits_won = t2
 where student_name = st and credits_won = t1;
 delete from takes
 where student_name = st and course_name = co;
 insert into has_finished
 values (st, co);
 ev := 'pass(' || st || ',' || co || ',' || t1 || ',' || t2 || ')';
```

```
 ins_log(rn,ev);
 commit;
end pass;

create procedure create_program(rn number, p varchar2, r number) is
 found number;
 ev varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select count(*) into found
 from program
 where program_name = p;
 if found > 0 then return;
 end if;
 insert into program
 values (p, r);
 ev := 'create_program(' || p || ',' || r || ')';
 ins_log(rn,ev);
 commit;
end create_program;

create procedure receive_degree(rn number, st varchar2, pr varchar2) is
 found number;
 ev varchar(100);
 credits_won_A number;
 program_name_pr varchar(100);
 requirement_A number;
 student_name_st varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select program_name, requirement into program_name_pr, requirement_A
 from program
 where program_name = pr;
 select student_name, credits_won into student_name_st, credits_won_A
 from student
 where student_name = st and credits_won = requirement_A;
 select count(*) into found
 from takes
 where student_name = st;
 if found > 0 then return;
 end if;
 insert into graduated_in
 values (st, pr);
 ev := 'receive_degree(' || st || ',' || pr || ')';
 ins_log(rn,ev);
 commit;
end receive_degree;
```

Notice that all procedures have an additional first parameter, rn, to be filled-up with the reference of the IDB-transaction under which it should be registered. Also notice that at the end, just before committing, all procedures duly update the LOG table.

Characteristics of Oracle led us to implement tests in a way that would not cause the programs to abort as the result of a failed select command. A typical case is that of the `enroll` procedure, when checking whether or not this is the person's first enrollment. Directly selecting a tuple with the person's name from the STUDENT table should be the normal way to do the test, but to be able to proceed if no such tuple is found we resorted to the count function:

```
select count(*) into found
from student
where student_name = s;
if found = 0 then
insert into student
values (s, 0);
end if;
```

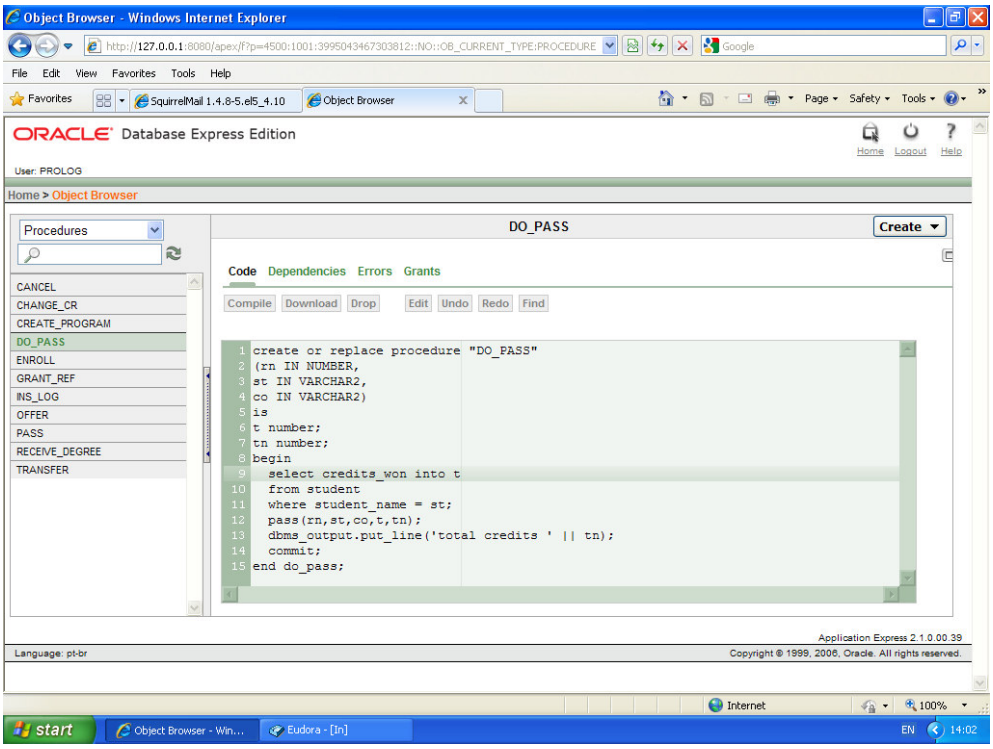
A complementary procedure that had to be produced manually, was made necessary by the "in out" (input-output) nature of parameter `t2` of the `pass` procedure, which due to the presence of such parameter cannot be called directly:

```
create procedure do_pass(rn number, st varchar2, co varchar2) is
  t number;
  tn number;
begin
  select credits_won into t
  from student
  where student_name = st;
  pass(rn,st,co,t,tn);
  dbms_output.put_line('total credits ' || tn);
  commit;
end do_pass;
```

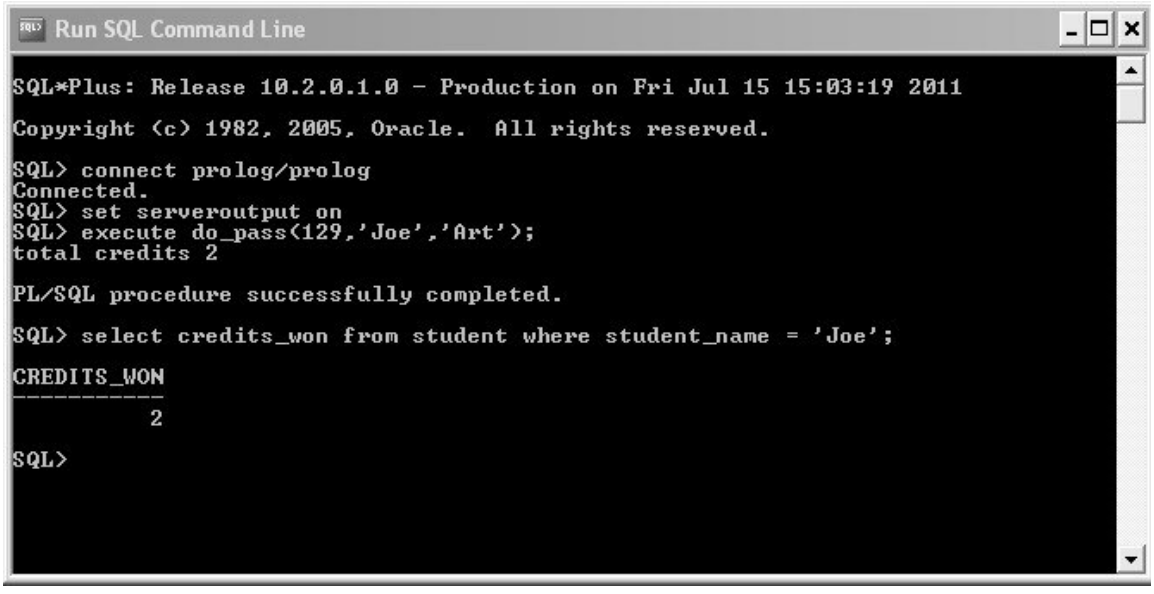
Yet another manually-produced procedure is the one that handles the LOG table ─ it is invoked by all procedures (except `do_pass`, of course) and should remain the same for any future application that may be designed with the help of the IDB tool.

```
create procedure ins_log(ref number, ev varvhar2) is
   ts varchar(100);
begin
   select to_char(localtimestamp,'YYYY/MM/DD/HH24/MI/SS/FF3')
   into ts
   from DUAL;
   insert into log
   values (ref,ts,ev);
   commit;
end ins_log;
```

We found it expedient to employ two different shells to handle the stored procedures: the **Oracle Database Xe** shell to enter the compiled programs, performing a "cut and paste" upon the text file generated by the IDB tool; and the **SQL*Plus** shell for execution runs. Reproduced next are the screens showing the `do_pass` procedure being registered in the former shell, and its execution in the latter ─ note the "set serveroutput on" line, required for the onscreen exhibition of the printed output.



Entering the procedures via the Oracle Database XE shell



Execution in the SQL*Plus shell

5. Concluding remarks

The work developed until now was enough to demonstrate that the availability of a plan-generation facility, with access to the meta-data knowledge conveyed by the conceptual schemas, is an asset towards the specification and experimentation of intelligent information systems. Moreover, collecting traces of execution runs, both at the design stages and after the system is delivered for routine usage, eases the maintenance and redesign tasks, for which the IDB tool can again be conveniently used. The analysis of traces (plot mining) is the object of a previous study of our group [FCBB]. Also promising is the reuse and adaptation of the narrative patterns discovered in one domain, whenever they are found to be applicable to analogous situations in other domains [KL, BBFC].

In a broader context, our tool may come to serve as one more resource among other systems that similarly provide flexible environments (or meta-environments) for software development. The Talisman system, reviewed in [St][1], is a significant contribution of our department in this direction.

A prototype implementation of IDB is fully operational, but more effort is needed to improve performance and detect and correct possible deficiencies. Moreover, after a sizeable number of different users have tried it, a thoroughly-documented user-friendly interface can be designed and made available. Indeed, the greater benefit of environments to support the production of information systems is to give to all classes of prospective users the chance of participating in the specification, and then of making sure, through early usage, that their needs are adequately satisfied.

Referencces

[BBFC] Barbosa, S. D. J., Breitman, K. K., Furtado, A. L. , Casanova, M. A.. "Similarity and analogy over application domains". *Proceedings of SBBD*, 2007.

[BCN] Batini, C., Ceri, S. and Navathe, S. *Conceptual Design – an Entity-Relationship Approach.* Benjamin Cummings, 1992.

[FC] Furtado, A.L., Ciarlini, A.E.M. "Generating Narratives from Plots using Schema Information". In: *Proc. 5th International Workshop on Applications of Natural Language for Information Systems*. Springer-Verlag, 2000.

[FCBB] Furtado, A.L., Casanova, M.A., Barbosa, S.D.J., Breitman, K.K. "Plot mining as an aid to characterization and planning". Technical Report MCC07, PUC-Rio, 2007.

[FN] Fikes, R. E. and Nilsson, N. J. "STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence* , 2(3-4), 1971.

[GG] Garland, S.J., Guttag, J.V. "Inductive Methods for Reasoning about Abstract Data Types". In *Proc. Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[KL] Kolodner, J.L., Leake, D. "A Tutorial Introduction to Case-Based Reasoning". In *Case-Based Reasoning: Experiences, Lessons and Future Directions*, Leake, D. (ed.), MIT Press. pp. 31-65, 1996.

[Mo] Morgan, G. *Images of organization - Executive edition*. Sage Publications, 1998.

[St] Staa, A. v. "Overview of the Talisman Version 5 software engineering meta-environment". Technical Report MCC08/11, Pontifícia Universidade do R.J., 2011.

[TCF] Tucherman, L., Casanova, M.A., Furtado, A.L. "The CHRIS consultant-a tool for database design and rapid prototyping". *Information Systems* 15(2): 187-195, 1990.

[Wa] Waterman, D. A. *A Guide to Expert Systems*. Addison-Wesley, 1985.

[1] ftp://ftp.inf.puc-rio.br/pub/docs/techreports/11_08_staa.pdf

Appendix

```
/*  The IDB logic programming tool */

% a Casa_Soft product
% version 1.0 - 2010
% distributed with absolutely no guarantee

:- set_prolog_flag(verbose,silent).
:- style_check([-singleton,-discontiguous]).
:- dynamic op_level/1,log/1,
   state_rep/1,ini/0,state_rep_ini/1,
   ref/1,ref_list/1.
:- dynamic fail_flag/0,not_exs_flag/0, db_flag/0.
:- dynamic include_ex/0.

:- consult(war_idb).
:- use_module(library(odbc)).
:- use_module(library(clpfd)).
:- set_prolog_flag(clpfd_goal_expansion,false).

:- (include_ex, !; consult(ex1_idb), assert(include_ex)).

:- odbc_connect('XE', C, [user(prolog), password(prolog),
                          alias(prolog),
                     open(once)]),
   odbc_set_connection(C,null(null)).


% choose whether to work on memory or on database

env_option(Eo) :-
  (Eo == mem; Eo == memory),
  (db_flag, retract(db_flag);
   not db_flag).

env_option(Eo) :-
  (Eo == db; Eo == database),
  (db_flag;
   not db_flag, assert(db_flag)).


% primitive query and update operations

insert(E) :-
  ground(E),
  E =.. [T|R],
  insert(R,T).

insert(R,T) :-
  odbc_connect('XE', C, [user(prolog), password(prolog),
                          alias(prolog),
                     open(once)]),
  Cond =.. [T|R],
  (not select(Cond), !; !, fail),
  findall('?',on(X,R),Lq),
  conj_list(Cq,Lq),
  term_to_atom(Cq,Aq),
  concat('insert into ',T,Ins1),
  concat(Ins1,' \ values(',Ins2),
  concat(Ins2,Aq,Ins3),
```

```
  concat(Ins3,')',Ins),
  upcase_atom(T,T1),
  findall(Dt,
    (odbc_prepare(prolog,
     'select data_type from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
   odbc_execute(S1, [T1], row(Dt))),
  Dts),
  prep_set2(R,Dts,Preplist),
  odbc_prepare(prolog,Ins,Preplist,St),
  odbc_execute(St,R).

delete(E) :-
  E =.. [T|R],
  delete(R,T).

delete(R,T) :-
  odbc_connect('XE', C, [user(prolog), password(prolog),
                      alias(prolog),
                 open(once)]),
  upcase_atom(T,T1),
  findall(Col,
    (odbc_prepare(prolog,
     'select column_name from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
   odbc_execute(S1, [T1], row(Col))),
  Cols),
  findall(Dt,
    (odbc_prepare(prolog,
     'select data_type from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
   odbc_execute(S1, [T1], row(Dt))),
  Dts),
  sel1(Cols,R,Where,Vals,Del,Lvar),
  sel3(Where,Vals,Del_cont),
  concat('delete from ',T,Del_tab),
  (not (Del_cont = []),!,
   concat(Del_tab,' where ',Del_expr1),
   concat(Del_expr1,Del_cont,Del_expr);
   Del_expr = Del_tab),
  sel4(R,Dts,Preplist),
  sel6(Vals,Valsx),
  odbc_prepare(prolog,Del_expr,Preplist,St),
  odbc_execute(St,Valsx).

user:update(E) :-
  E =.. [T|P],
  upd(P,Po,Pn),
  update(Po,Pn,T).

upd([],[],[]).
upd([P|R],[Vo|S],[Vn|T]) :-
  not var(P),
  P = (Vo => Vn), !,
  upd(R,S,T).
upd([P|R],[P|S],[_|T]) :-
  not var(P), !,
  upd(R,S,T).
upd([P|R],[P|S],[_|T]) :-
  upd(R,S,T).
```

```
update(R,R1,T) :-
  odbc_connect('XE', C, [user(prolog), password(prolog),
                         alias(prolog),
                  open(once)]),
  upcase_atom(T,T1),
  findall(Col,
    (odbc_prepare(prolog,
     'select column_name from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
    odbc_execute(S1, [T1], row(Col))),
   Cols),
  findall(Dt,
    (odbc_prepare(prolog,
     'select data_type from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
    odbc_execute(S1, [T1], row(Dt))),
   Dts),
  sel1(Cols,R,Where,Vals_w,Upd,Lvar),
  sel1(Cols,R1,Set1,Vals_s,Upd1,Lvar1),
  sel3(Where,Vals_w,Upd_cont),
  prep_set1(Set1,Upd_cont1),
  concat('update ',T,Upd_tab),
  concat(Upd_tab,' set ', Upd_expr2),
  concat(Upd_expr2,Upd_cont1,Set),
  (not (Upd_cont = []),!,
   concat(Set,' where ',Upd_expr1),
   concat(Upd_expr1,Upd_cont,Upd_expr);
   Upd_expr = Upd_tab),
  prep_set2(R1,Dts,Preplist1),
  sel4(R,Dts,Preplist2),
  append(Preplist1,Preplist2,Preplist),
  sel6(Vals_w,Vals_w1),
  append(Vals_s,Vals_w1,Vals),
  odbc_prepare(prolog,Upd_expr,Preplist,St),
  odbc_execute(St,Vals).

select(E) :-
  E =.. [En,I],
  entity(En,_),
  findall(X,attribute(En,_),As),
  not (As == []),
  Q =.. [En,I|As],
  select(Q).

select(A) :-
  A =.. [An,I,V],
  attribute(E,An),
  odbc_connect('XE', _, [ user(prolog), password(prolog), alias(prolog),
open(once) ]),!,
  upcase_atom(E,T),
  findall(Col,
    (odbc_prepare(prolog,
     'select column_name from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
    odbc_execute(S1, [T], row(Col))),
   Cols),
  upcase_atom(An,Ancol),
  item(Ancol,Cols,N),
  findall(X,on(Y,Cols),Parm),
```

```
  item(I,Parm,1),
  item(V,Parm,N),
  Q =.. [E|Parm],
  select(Q).

select(S) :-
  odbc_connect('XE', _, [ user(prolog), password(prolog), alias(prolog),
open(once) ]),!,
  S =.. [T1|Plist],
  upcase_atom(T1,T),
  findall(Col,
    (odbc_prepare(prolog,
     'select column_name from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
    odbc_execute(S1, [T], row(Col))),
   Cols),
  findall(Dt,
    (odbc_prepare(prolog,
     'select data_type from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
    odbc_execute(S1, [T], row(Dt))),
   Dts),
  sel1(Cols,Plist,Where,Vals,Sel,Lvar),
  sel2(Sel,Sel_ini),
  sel3(Where,Vals,Sel_cont),
  concat('select ',Sel_ini,Sel_ask),
  concat(Sel_ask,' from ',Sel_tab1),
  concat(Sel_tab1,T,Sel_tab),
  (not (Sel_cont = []),!,
   concat(Sel_tab,' where ',Sel_expr1),
   concat(Sel_expr1,Sel_cont,Sel_expr);
   Sel_expr = Sel_tab),
  sel4(Plist,Dts,Preplist),
  sel5(Plist,Dts,Optlist),
  (Optlist = [], Topl = [];
   not (Optlist = []), Topl = [types(Optlist)]),
  (T1 == log, concat(Sel_expr,' order by TS',Sel_exprx);
   not (T1 == log), Sel_exprx = Sel_expr),
  sel6(Vals,Valsx),
  odbc_prepare(prolog,Sel_exprx,Preplist,Stat,Topl),
  odbc_execute(Stat, Valsx, Answer),
  (not (Lvar == []),
     Answer =.. [row|Lvar];
   Lvar == []).

sel1([],[],[],[],[],[]).
sel1([C|Rc],[V|Rv],[C|W],[V|Wv],A,Lv) :-
  nonvar(V),
  sel1(Rc,Rv,W,Wv,A,Lv).
sel1([C|Rc],[V|Rv],W,Wv,[C|A],[V|Lv]) :-
  var(V),
  sel1(Rc,Rv,W,Wv,A,Lv).

sel2([],'*') :- !.
sel2([C],C) :- !.
sel2([C|R],S) :-
  sel2(R,S1),
  concat(C,', ',S2),
  concat(S2,S1,S).

sel3([],[],[]) :- !.
```

```
sel3([C],[V],CV) :-
  V == null,!,
  concat(C,' is null',CV).
sel3([C],_,CV) :- !,
  concat(C,' = ?',CV).
sel3([C|R],[V|S],CV) :-
  V == null,!,
  sel3(R,S,CV1),
  concat(C,'is null and ',CV2),
  concat(CV2,CV1,CV).
sel3([C|R],[_|S],CV) :- !,
  sel3(R,S,CV1),
  concat(C,' = ? and ',CV2),
  concat(CV2,CV1,CV).

sel4([],[],[]) :- !.
sel4([P|R1],[Dt|R2],R3) :-
   P == null, !,
   sel4(R1,R2,R3).
sel4([P|R1],[Dt|R2],[Dc|R3]) :-
  nonvar(P),!,
   (Dt == 'NUMBER', Dc = (integer > numeric);
    Dt == 'FLOAT', Dc = (double > float);
    Dt == 'VARCHAR2', Dc = (atom > varchar(100))),
  sel4(R1,R2,R3).
sel4([P|R1],[_|R2],R3) :-
  var(P),
  sel4(R1,R2,R3).

sel5([],[],[]) :- !.
sel5([P|R1],[Dt|R2],[Dc|R3]) :-
  var(P),!,
   (Dt == 'NUMBER', Dc = integer;
    Dt == 'FLOAT', Dc = float;
    Dt == 'VARCHAR2', Dc = atom),
  sel5(R1,R2,R3).
sel5([P|R1],[_|R2],R3) :-
  nonvar(P),
  sel5(R1,R2,R3).

sel6([],[]) :- !.
sel6([V|Vl],[V|Vl1]) :-
  not (V == null), !,
  sel6(Vl,Vl1).
sel6([V|Vl],Vl1) :-
  sel6(Vl,Vl1).

prep_set1([],[]) :- !.
prep_set1([C],CV) :- !,
  concat(C,' = ?',CV).
prep_set1([C|R],CV) :- !,
  prep_set1(R,CV1),
  concat(C,' = ? and ',CV2),
  concat(CV2,CV1,CV).

prep_set2([],[],[]) :- !.
prep_set2([P|R1],[Dt|R2],[Dc|R3]) :-
  nonvar(P),!,
   (Dt == 'NUMBER', Dc = (integer > numeric);
    Dt == 'FLOAT', Dc = (double > float);
    Dt == 'VARCHAR2', Dc = (atom > varchar(100))),
  prep_set2(R1,R2,R3).
prep_set2([P|R1],[_|R2],R3) :-
```

```
  var(P),
  prep_set2(R1,R2,R3).


% queries based on the catalogue

table_info :-
  odbc_connect('XE', _, [ user(prolog), password(prolog), alias(prolog),
open(once) ]),!,
  nl,
  forall(
  (odbc_query(prolog,
  'select
user_tab_columns.table_name,user_tab_columns.column_name,user_tab_columns.data_ty
pe
  from user_tab_columns, user_tables
  where user_tab_columns.table_name = user_tables.table_name',
  row(T,C,D)), downcase_atom(T,Tt), (entity(Tt,_);relationship(Tt,_) )),
  (write(T), spc, write(C), write(' ('),
   downcase_atom(D,D1), write(D1), write(')'), nl)).

spc :-
  stream_property(S,alias(user_output)),
  stream_property(S,position(Pos)),
  Pos =.. [F,A,B,P,C],
  P1 is 19 - P,
  tab(P1),
  write(' -  ').

tables(Ts) :-
  odbc_connect('XE', _, [ user(prolog), password(prolog), alias(prolog),
open(once) ]),!,
  findall(T,
    (odbc_query(prolog,'select table_name from user_tables',row(T1)),
    downcase_atom(T1,T),(entity(T,_);relationship(T,_))),
    Ts).

traverse_table(T,A) :-
  odbc_connect('XE', _, [user(prolog), password(prolog), alias(prolog),
open(once)]), !,
  tables(Ts),
  on(T,Ts),
  concat('select * from ',T,Sel),
  odbc_query(prolog, Sel, R),
  R =.. [row|P],
  A =.. [T|P].

er_tab(E,T) :-
  entity(E,N),
  findall(A,attribute(E,A),As),
  T =.. [E,N|As].
er_tab(R,T) :-
  relationship(R,[E1,E2]),
  entity(E1,N1),
  entity(E2,N2),
  T =.. [R,N1,N2].

db_tab(E,Tab) :-
  odbc_connect('XE', _, [ user(prolog), password(prolog), alias(prolog),
open(once) ]),!,
  upcase_atom(E,T),
  findall(Col,
    (odbc_prepare(prolog,
```

```
     'select column_name from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
    odbc_execute(S1, [T], row(Col))),
   Cols),
  findall(Coli,(on(X,Cols),downcase_atom(X,Coli)),Colns),
  Tab =.. [E|Colns].

db_types(T,Dts) :-
  odbc_connect('XE', _, [ user(prolog), password(prolog), alias(prolog),
open(once) ]),!,
  upcase_atom(T,T1),
  findall(Dt,
    (odbc_prepare(prolog,
     'select data_type from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
    odbc_execute(S1, [T1], row(Dt))),
   Dts).
```

% transaction management

```
trans(T) :-
  check_ref(R),
  trans1(T).

$(T) :- trans(T).

trans1(Ops) :-
  odbc_connect('XE', C, [user(prolog), password(prolog),
                      alias(prolog),
                  open(once)]),!,
  odbc_set_connection(C,auto_commit(false)),
  (Ops,
   odbc_end_transaction(C, commit), !;
   assert(fail_flag),odbc_end_transaction(C, rollback)),
  odbc_set_connection(C, auto_commit(true)),
  (fail_flag,retract(fail_flag),!,fail;
   not fail_flag).
```

% grant and check a reference

```
grant_ref(R) :-
  (ref(X), retract(ref(X));
   not ref(X)),
   assert(ref(R)),
  (select(ref(R));
   not select(ref(R)),
   insert(ref(R))).

check_ref(R) :-
  not var(R),
  ref(R), !.
check_ref(R) :-
  not var(R),
  select(ref(R)), !.
check_ref(R) :-
  nl,write('valid reference needed'),nl,
  !, fail.

ref_list(L) :-
```

```
  findall(R,select(ref(R)),L).


% add a refererence test and log-insertion clause to the operations

add_ins_log :-
  forall(operation(O), add_ins_log(O)).

add_ins_log(O) :-
  operation(O),
  clause(O,C),
  appc(ref(R), (C,ins_log(R,O)),C1),
  retract((O :- C)),
  assert((O :- C1)).


% update and query the log

ins_log(R,Op) :-
  term_to_atom(Op,Ev1),
  rem_quote(Ev1,Ev),
  ref(R),
  get_time(T),
  ts_date(T,Td),
  date_char(Td,Ts),
  odbc_prepare(prolog, 'insert into log \ values (?,?,?)',
               [integer > numeric, atom > varchar(100), atom > varchar(100)], S),

  odbc_execute(S, [R,Ts,Ev]).

select_log(R,Ts,Ev) :-
  ref_list(L),
  on(R,L),
  select(log(R,Ts1,Ev1)),
  ev_term(Ev1,Ev),
  char_date(Ts1,Ts).

ev_term(Ev,T) :-
  sub_string(Ev,I,L,R,'('),
  J is I + 1,
  K is R - 1,
  sub_string(Ev,J,K,_,P),
  ev_t1(P,Pe),
  sub_string(Ev,0,J,_,F),
  concat(F,Pe,T1),
  concat(T1,')',T2),
  term_to_atom(T,T2), !.

ev_t1(P,Pe) :-
  not sub_string(P,_,_,_,','),
  term_to_atom(T,P),
  (not number(T),
   concat('''',P,P1),
   concat(P1,'''',Pe);
   number(T),
   Pe = P).
ev_t1(P,Re) :-
  sub_string(P,I,L,R,','),
  sub_string(P,0,I,R1,P1),
  J is I + 1,
  sub_string(P,J,R,_,P2),
  ev_t1(P1,P1e),
  ev_t1(P2,P2e),
```

```
  concat(P1e,',',Re1),
  concat(Re1,P2e,Re).

rem_quote(Eq,E) :-
  name(Eq,Lq),
  q_rem(Lq,L),
  name(E,L).

q_rem([],[]) :- !.
q_rem([T|L],R) :-
  T == 39,!,
  q_rem(L,R).
q_rem([T|L],[T|R]) :-
  q_rem(L,R).

date_list(D,L) :-
  date_list1(D,L1),
  reverse(L1,L).

date_list1(D,[D]) :-
  (atomic(D);var(D)),!.
date_list1(D/X,L) :-
  date_list1(D,L1),
  append([X],L1,L).

list_date(L,D) :-
  reverse(L,L1),
  list_date1(L1,D).

list_date1([X],X) :- !.
list_date1([A|R],D) :-
  list_date1(R,S),
  D = S/A.

date_char(D,C) :-
  date_list(D,L),
  dat_c1(L,C).

dat_c1([F],Fc) :- !,
  T is F + 1000,
  term_to_atom(T,A),
  sub_string(A,1,3,_,Fc).
dat_c1([N|R],C) :-
  (N > 2000, !, term_to_atom(N,Nc);
   T is N + 100,
   term_to_atom(T,A),
   sub_string(A,1,2,_,Nc)),
  dat_c1(R,Rc),
  concat(Nc,'/',T1),
  concat(T1,Rc,C).

char_date(C,D) :-
  term_to_atom(D,C).

date_ts(Dt,Ts) :-
  Dt = (Y/Mo/D/H/Mi/S/F),
  S1 is S + F/1000,
  date_time_stamp(date(Y,Mo,D,H,Mi,S1,10800,local,true),Ts).

ts_date(Ts,(Y/Mo/D/H/Mi/S/F)) :-
  convert_time(Ts,Y,Mo,D,H,Mi,S,F).
```

```
% trace extraction

get_trace(Ref,Tr) :-
  ref_list(Lref),
  on(Ref,Lref),
  findall(Ev,
    (select(log(Ref,Ts,Ev1)),
   ev_term(Ev1,Ev)),
   Tr).

show_trace :-
  nl,
  get_trace(R,Tr),
  write('ref'),write(': '),write(R),nl,
  forall(on(Op,Tr),show(Op)),
  nl.


% display events or facts in template-driven sentence format

show(F) :-
  templatex(F,T),
  xclist(T,S),
  write(S), nl.


% default templates for facts

template(Einst,T) :-
  entity(E,I),
  Einst =.. [E,Iv],
  name(E,[N1|_]),
  name(aeiou,N),
  (on(N1,N),A = 'an ';
   not on(N1,N),A = 'a '),
  T = ['There is ',A,E,' with ',I,' ',Iv,'.'].

template(Ainst,T) :-
  attribute(E,A),
  Ainst =.. [A,Iv,Av],
  T = ['The ',A,' of ',Iv,' is ',Av,'.'].

template(Rinst,T) :-
  relationship(R,[E1,E2]),
  Rinst =.. [R,Iv1,Iv2],
  T = [Iv1,' is related to ',Iv2,' by ',R,'.'].


% calls for plan generation and execution

goal_exec(G) :-
   plans(G,P),
   narrate(P),
   nl,write('choose one: yes/no/stop - '),
   read(A),
   (A == yes, !, plan_exec(P);
    A == no, fail;
    A == stop, !).

plan_exec(P) :-
   ex1p(P,L),
   conj_list(Tr,L),
   (db_flag, $Tr;
```

```
    not db_flag,
   execute(P)).

% perform the db-implementation of the operations

compile_ops :-
  forall(operation(O),
    compile_op(O,Oc)).

list_ops :-
  forall(operation(O),
    listing(O)).

compile_op(O,Oc) :-
  operation(O),
  (clause(O,C),retract((O :- C));
   not clause(O,C)),
  compile_op1(O,Oc),
  assert(Oc).

compile_op1(O,Oc) :-
  operation(O),
  comp_preconds(O,Pr),
  comp_effs(O,Ef),
  appc((ref(R),Pr),Ef,B1),
  appc(B1,ins_log(R,O),B),
  Oc = (O :- B).

comp_preconds(O,P) :-
  operation(O),
  clause(precond(O,P1),B),
  (B == true,!, P1x = P1;
   B = holds_now(S),!,
   appc(P1,S,P1x);
   appc(P1,B,P1x)),
  repl_r(/X,X,P1x,P2),
  repl_r(X:_,X,P2,P3),
  rem_comp(P3,P4),
  replace_f(P4,P).

comp_effs(O,Oc) :-
  if_needed(O,Nd,Na),
  ndgroup(O,Nd,Ndg1),
  nagroup(O,Na,Nag1),
  ugroup(O,Ndg1,Nag1,Ndg,Nag),
  necess_effects(O,D,A),
  dgroup(O,D,Dg1),
  agroup(O,A,Ag1),
  ugroup(O,Dg1,Ag1,Dg,Ag),
  append(Ndg,Nag,Oc1),
  append(Oc1,Dg,Oc2),
  append(Oc2,Ag,Oc3),
  conj_list(Oc,Oc3).

necess_effects(O,D,A) :-
  operation(O),
  xbagof(T,
    (P,C)^(current_predicate(deleted,P),
    clause(deleted(T,O),C),
    not var(T)),
   D),
  xbagof(T,
```

```
    (P,C)^(current_predicate(added,P),
    clause(added(T,O),C),
    not var(T)),
   A).

if_needed(O,Nd,Na) :-
  operation(O),
  xbagof(T,
   (P,C)^(current_predicate(/,P),
    clause(/deleted(T,O),C)),
   Nd),
  xbagof(T,
   (P,C)^(current_predicate(/,P),
    clause(/added(T,O),C)),
   Na).

dgroup(O,[],[]).
dgroup(O,[X|R],[delete(T)|S]) :-
  gr_t(X,T,Tab,Ar),
  (Ar > 1, !,
   N is Ar - 1,
   count(L,N),
   append(L,U,R),
   gr_t1(T,Tab,E,L),
   dgroup(O,U,S);
   dgroup(O,R,S)).

agroup(O,[],[]).
agroup(O,[X|R],[insert(T)|S]) :-
  gr_t(X,T,Tab,Ar),
  (Ar > 1, !,
   N is Ar - 1,
   count(L,N),
   append(L,U,R),
   gr_t1(T,Tab,E,L),
   agroup(O,U,S);
   agroup(O,R,S)).

ugroup(O,[],As,[],As).
ugroup(O,[D|R],As,[update(T2)|S],C) :-
  D = delete(T),
  T =.. [A,I,Vo],
  attribute(E,A), !,
  on(insert(T1),As),
  T1 =.. [A,I,Vn],
  er_tab(E,Row),
  arg(Pa,Row,A),
  functor(Row,E,N),
  functor(T2,E,N),
  arg(1,T2,I),
  arg(Pa,T2,Vo=>Vn),
  append(W,[insert(T1)|Z],As),
  append(W,Z,B),
  ugroup(O,R,B,S,C).
ugroup(O,[D|R],As,[D|R1],As1) :-
  ugroup(O,R,As,R1,As1).

ndgroup(O,[],[]).
ndgroup(O,[X|R],
          [(select(T),delete(T);not select(T))|S]) :-
  gr_t(X,T,Tab,Ar),
  (Ar > 1, !,
   N is Ar - 1,
```

```
   count(L,N),
   append(L,U,R),
   gr_t1(T,Tab,E,L),
   ndgroup(O,U,S);
   ndgroup(O,R,S)).

nagroup(O,[],[]).
nagroup(O,[X|R],
         [(select(T1);not select(T1),insert(T))|S]) :-
  gr_t(X,T,Tab,Ar),
  (Ar > 1, !,
   N is Ar - 1,
   count(L,N),
   append(L,U,R),
   gr_t1(T,Tab,E,L),
   T =.. [Tf,I|_],
   T1 =.. [Tf,I],
   nagroup(O,U,S);
   T1 = T,
   nagroup(O,R,S)).

gr_t(X,T,Tab,Ar) :-
  X =.. [E,I],
  entity(E,N),!,
  er_tab(E,Tab),
  functor(Tab,E,Ar),
  functor(T,E,Ar),
  arg(1,T,I).
gr_t(X,X,_,1) :-
  X =.. [R,I1,I2],
  relationship(R,_), !.
gr_t(X,X,_,1) :-
  X =.. [A,I,V],
  attribute(_,A).

gr_t1(T,Tab,E,[]).
gr_t1(T,Tab,E,[P|R]) :-
  P =.. [A,_,V],
  attribute(E,A),
  arg(J,Tab,A),
  arg(J,T,V),
  gr_t1(T,Tab,E,R).

replace(_,_,X,[]) :- X == [], !.
replace(X,Y,X,Y) :-
  atomic(X), !.
replace(X,Y,X,Y) :-
  var(X), !.
replace(X,Y,Z,Z) :-
  var(Z), !.
replace(X,Y,Z,Z) :-
  atomic(Z), !,
  not (X = Z), !.
replace(X,Y,X,Y) :-
  X =.. [F|L], not var(Y),!.
replace(X,Y,ZL,[Z1|L1]) :-
  not var(ZL),
  ZL = [Z|L], !,
  replace(X,Y,Z,Z1),
  replace(X,Y,L,L1).
replace(X,Y,Z1,Z2) :-
  not var(Z1),
  Z1 =.. [F|L],
```

```
  replace(X,Y,F,F1),
  replace(X,Y,L,L1),
  Z2 =.. [F1|L1].

repl_r(X,Y,R,S) :-
  ground(X),
  replace(X,Y,R,S).
repl_r(X,Y,R,S) :-
  repl_r1(X,Y,(R,nil),(S,nil)).

repl_r1(X,Y,R,S) :-
  replace(X,Y,R,R),
  S = R.
repl_r1(X,Y,R,S) :-
  copy((X,Y),(X1,Y1)),
  replace(X,Y,R,T),
  not (R = T),
  repl_r1(X1,Y1,T,S).

rem_comp(P,Q) :-
  conj_list(P,L),
  rem_comp1(L,L1),
  conj_list(Q,L1).

rem_comp1([],[]) :- !.
rem_comp1([A|R],S) :-
  A =.. [F|_],
  term_to_atom(F,Fa),
  name(Fa,[35|T]),
  not (T == [61]),!,
  rem_comp1(R,S).
rem_comp1([A|R],[A|S]) :-
  rem_comp1(R,S).

replace_f(X,[]) :- X == [], !.
replace_f(X,X) :-
  atomic(X), !.
replace_f(X,X) :-
  var(X), !.
replace_f(X,select(X)) :-
  fact(X),!.
replace_f([Z|L],[Z1|L1]) :- !,
  replace_f(Z,Z1),
  replace_f(L,L1).
replace_f(Z1,Z2) :-
  Z1 =.. [F|L],
  replace_f(F,F1),
  replace_f(L,L1),
  Z2 =.. [F1|L1].
```

% implement the operations as stored procedures

```
compile_proc(O,P) :-
  O =.. [F|Parms],
  operation(O),
  clause(O,Cx),
  once(op_body(Cx,C)),
  conj_list(C,L1),
  once(pre_comp(O,L1,L)),
  copy((O,L),(Ocop,Lcop)),
  varnames((Ocop,Lcop)),
  O = Ocop,
```

```
  once(comp_proc(L,Lcop,Ps,Lv)),
  concat('end ',F,En1),
  concat(En1,';',En),
  conj_list(C1,L),
  once(header(O,C1,H)),
  prep_parms(F,Parms,Ev),
  xsetof(Vi,on(Vi,Lv),Lv1),
  Ps1 = [H,' found number;',' ev varchar(100);',Lv1,
      'begin',
      ' select r into found',' from ref',' where r = rn;',
      Ps,
      Ev,' ins_log(rn,ev);',
      ' commit;',En],
  flatten(Ps1,P).

op_body(Bx,B) :-
  conj_list(Bx,L),
  append([ref(_)],L1,L),
  append(L2,[ins_log(_,_)],L1),
  conj_list(B,L2).

comp_proc([],[],[],[]) :- !.
comp_proc([Cl|R],[Clcop|Rcop],[Clc|S],Vs) :-
  cl_patt(Cl,Clcop,Clc1),
  (is_list(Clc1),Clc = Clc1,Vs1 = [];
   Clc1 = (Clc/Vs1);
   Clc1 = (Clca + Clcb),append(Clca,Clcb,Clc),Vs1 = []),
  comp_proc(R,Rcop,S,Vs2),
  append(Vs1,Vs2,Vs).

cl_patt(Cl,Clcop,P) :-
  Cl = select(_), !,
  select_query(Cl,Clcop,S,Lv),
  (not_exs_flag, !,
   retract(not_exs_flag),
   P = ([S,' end if;']/Lv);
   P = (S/Lv)).
cl_patt(Cl,_,P) :-
  Cl = insert(_), !,
  insert_eff(Cl,S),
  (not_exs_flag, !,
   retract(not_exs_flag),
   P = [S,' end if;'];
   P = S).
cl_patt(Cl,_,P) :-
  Cl = delete(_), !,
  delete_eff(Cl,S),
  (not_exs_flag, !,
   retract(not_exs_flag),
   P = [S,' end if;'];
   P = S).
cl_patt(Cl,_,P) :-
  Cl = update(_), !,
  update_eff(Cl,S),
  (not_exs_flag, !,
   retract(not_exs_flag),
   P = [S,' end if;'];
   P = S).
cl_patt(Cl,_,P) :-
  Cl = (not select(_)),
  notsel_cond(Cl,S),
  P = S.
cl_patt(Cl,_,P) :-
```

```
  Cl = (X #= E),
  term_to_atom((X := E),Clat),
  concat(' ',Clat,P1),
  concat(P1,';',P2),
  term_to_atom(P2,P3),
  rem_quote(P3,P4),
  P = [P4].
cl_patt(Cl,Clcop,P) :-
  Cl = (not select(_); select(_), Op),
  Clcop = (_;_,Opcop),
  op_exists_cond(Cl,S1,Op),
  cl_patt(Op,Opcop,S2),
  P = (S1 + S2).
cl_patt(Cl,Clcop,P) :-
  Cl = (select(_); not select(_), Op),
  Clcop = (_;_,Opcop),
  op_notexists_cond(Cl,S1,Op),
  cl_patt(Op,Opcop,S2),
  P = (S1 + S2).

pre_comp(O,[],[]) :- !.
pre_comp(O,[Cl|R],[Cl|S]) :-
  not (Cl = select(_)),!,
  pre_comp(O,R,S).
pre_comp(O,[Cl|R],[Cl|S]) :-
  Cl = select(T),
  T =.. [F|L],
  not (attribute(_,F)),!,
  pre_comp(O,R,S).
pre_comp(O,[Cl|R],[Cle|S]) :-
  Cl = select(T),
  sel_attr(Cl,Cle),
  pre_comp(O,R,S).

sel_attr(select(A),select(Q)) :-
  A =.. [An,I,V],
  attribute(E,An),
  odbc_connect('XE', _, [ user(prolog), password(prolog), alias(prolog),
open(once) ]),!,
  upcase_atom(E,T),
  findall(Col,
    (odbc_prepare(prolog,
     'select column_name from user_tab_columns \
      where table_name = ?',
      [atom > varchar(100)],S1),
   odbc_execute(S1, [T], row(Col))),
  Cols),
  upcase_atom(An,Ancol),
  item(Ancol,Cols,N),
  findall(X,on(Y,Cols),Parm),
  item(I,Parm,1),
  item(V,Parm,N),
  Q =.. [E|Parm].

header(O,C,H) :-
  operation(O),
  parms_type(O,C,L,Ts,Ps),
  O =.. [T|_],
  concat('create procedure ',T,S1),
  concat(S1,'(',S2),
  Ts1 = ['rn number'|Ts],
  c_list(Ts1,As),
  concat(S2,As,S3),
```

```
  concat(S3,') is',H).

c_list([B],B).
c_list([B|A],D) :-
  c_list(A,C),
  xconc(B,', ',B1),
  xconc(B1,C,D).

parms_type(O,B,L,Ts,Cs) :-
  O =.. [T|L],
  pty(O,L,Ts,Cs,B).

pty(O,[],[],[],_) :- !.
pty(O,[P|R],[Pt|S],[C|W],B) :-
  on_conj(Cl,B),
  (Cl = insert(E);
   Cl = delete(E);
   Cl = update(E1),
     E1 =.. [F|E1l],
    upd(E1l,Le1,Le2),
    varnames((Le1,Le2)),
    (E =.. [F|Le1]; E =.. [F|Le2]);
   Cl = select(E);
   Cl = (not select(E))),
   E =.. [T|L],
   v_on(P,L),
   db_tab(T,Tab),
   db_types(T,Dts),
   arg(I,E,P),
   arg(I,Tab,C),
   item(Ty1,Dts,I),
   downcase_atom(Ty1,Ty),
   (on_conj((P #= Expr),B),!,
    concat(P,' in out ',Pt1);
    concat(P,' ',Pt1)),
   concat(Pt1,Ty,Pt),
   pty(O,R,S,W,B).

prep_parms(F,Parms,Ev) :-
  prep_parms1(Parms,Ps),
  concat(' ev := ''',F,Ps1),
  concat(Ps1,'('''',Ps2),
  concat(Ps2,' || ',Ps3),
  concat(Ps3,Ps,Ev).

prep_parms1([P],Ps) :-
  concat(P,' || '')'';',Ps).
prep_parms1([P|R],Ps) :-
  prep_parms1(R,Rs),
  concat(P,' || '','''',Ps2),
  concat(Ps2,' || ',Ps3),
  concat(Ps3,Rs,Ps).

select_query(select(E),select(Ecop),S,Lvs) :-
  E =.. [T|L],
  Ecop =.. [T|Lcop],
  db_types(T,Typs),
  db_tab(T,Tab),
  Tab =.. [T|Cols],
  findall(N,
    (item(Pi,Lcop,I),
   item(Ci,Cols,I),
   concat(Ci,'_',N1),
```

```
    concat(N1,Pi,N)),
   Lc),
  findall(Col,
    (item(Pi,Lcop,I),
     item(Col,Cols,I)),
    Lcol),
  conj_list(Ccol,Lcol),
  term_to_atom(Ccol,Acol),
  findall(Vt,
    (item(Ni,Lc,I),
     item(Ti1,Typs,I),
    (Ti1 = 'VARCHAR2', Ti = 'varchar(100)';
     Ti1 = 'NUMBER', Ti = number),
    concat(' ',Ni,N0),
     concat(N0,' ',N1),
     concat(N1,Ti,N2),
    concat(N2,';',Vt)),
    Lvs),
  conj_list(Cc,Lc),
  term_to_atom(Cc,Ac),
  concat(' select ',Acol,S01),
  concat(S01,' into ',S02),
  concat(S02,Ac,S1),
  concat(' from ',T,S2),
  where_cl(E,T,S3),
  (S3 == [], concat(S2,';',S2f),S = [S1,S2f];
   not (S3 == []),
  S = [S1,S2,S3]),
  assign_v(L,Lc).

assign_v([],[]) :- !.
assign_v([X|R],[Y|S]) :-
  (var(X), X = Y;
   nonvar(X)),
  assign_v(R,S).

insert_eff(insert(E),S) :-
  E =.. [T|L],
  concat(' insert into ',T,S1),
  c_list(L,A),
  concat(' values (',A,S2),
  concat(S2,');',S3),
  S = [S1,S3].

delete_eff(delete(E),S) :-
  E =.. [T|L],
  concat(' delete from ',T,S1),
  where_cl(E,T,S2),
  S = [S1,S2].

update_eff(update(E),S) :-
  E =.. [T|L],
  concat(' update ',T,S1),
  upd(L,L1,L2),
  E1 =.. [T|L1],
  where_cl(E1,T,S5),
  E2 =.. [T|L2],
  db_tab(T,Tab),
  findall(CV,
    (arg(I,E2,V),
    nonvar(V),
    arg(I,Tab,C),
    concat(C,' = ',CV1),
```

```
    concat(CV1,V,CV)),
   Ps),
  conc_w1(Ps,S2),
  concat(' set ',S2,S3),
  S = [S1,S3,S5].

notsel_cond((not select(E)),S) :-
  E =.. [T|L],
  concat(' from ',T,S1),
  where_cl(E,T,S2),
  S = [' select count(*) into found ',S1,S2,' if found > 0 then return;',' end
if;'].

notexists_cond((not select(E)),S) :-
  E =.. [T|L],
  concat(' from ',T,S1),
  where_cl(E,T,S2),
  S = [' select count(*) into found ',S1,S2,' if found = 0 then '],
  assert(not_exs_flag).

exists_cond((select(E)),S) :-
  E =.. [T|L],
  concat(' from ',T,S1),
  where_cl(E,T,S2),
  S = [' select count(*) into found ',S1,S2,' if found > 0 then '],
  assert(not_exs_flag).

op_notexists_cond((select(E); not select(E), Op),S,Op) :-
  notexists_cond((not select(E)),S).

op_exists_cond((not select(E); select(E), Op),S,Op) :-
  exists_cond((select(E)),S).

where_cl(E,T,W) :-
  E =.. [T|_],
  db_tab(T,Tab),
  findall(CV,
    (arg(I,E,V),
   nonvar(V),
    arg(I,Tab,C),
    concat(C,' = ',CV1),
    concat(CV1,V,CV)),
   Ps),
(Ps == [],W = Ps;
 not (Ps == []),
  conc_w(Ps,S1),
  concat(' where ',S1,S2),
  concat(S2,'; ',W)  ).

conc_w([X],X) :- !.
conc_w([X|R],Y) :-
  conc_w(R,S),
  concat(X,' and ',T),
  concat(T,S,Y).

conc_w1([X],X) :- !.
conc_w1([X|R],Y) :-
  conc_w1(R,S),
  concat(X,', ',T),
  concat(T,S,Y).
```

```
% compiles and displays the stored procedures on screen

print_procs :-
  forall(operation(O),print_proc(O)).

print_proc(O) :-
  operation(O),
  compile_proc(O,P),
  nl,
  forall(on(Cl,P), (write(Cl),nl)),
  nl.


% compiles and registers the stored procedures on a text file

print_procs_txt :-
  prep,
  reset_teste,
  forall(operation(O),
    (compile_proc(O,P),
     nl(text),
     forall(on(Cl,P), (write(text,Cl),nl(text))),
     nl(text))),
  close(text).

print_proc_txt(O) :-
  operation(O),
  compile_proc(O,P),
  open_append,
  nl(text),
  forall(on(Cl,P), (write(text,Cl),nl(text))),
  nl(text),
  close(text).

prep :-
  (stream_property(S,alias(text)),stream_property(S,input),!,close(text);
   true),
  open('c:/Documents and Settings/furtado/Desktop/teste.txt',
       write,P,
      [alias(text)]).

reset_teste :-
  set_stream_position(text,'$stream_position'(1,1,1,0)).

open_append :-
  (stream_property(S,alias(text)),stream_property(S,input),!,close(text);
   true),
  open('c:/Documents and Settings/furtado/Desktop/teste.txt',
       append,P,
      [alias(text)]).
```

```
/* SMALL ACADEMIC EXAMPLE */

:- set_prolog_flag(verbose,silent).
:- style_check([-singleton,-discontiguous]).
:- dynamic student/1,credits_won/2,
   program/1,requirement/2,graduated_in/2,
   course/1,credits/2,takes/2,has_finished/2,has_dropped/2.
:- dynamic offer/2, enroll/2, transfer/3, cancel/1, change_cr/3,
   pass/4, create_program/2, receive_degree/2.
:- set_prolog_flag(clpfd_goal_expansion,false).


% static schema - classes of facts

entity(student,student_name).
attribute(student,credits_won).
entity(program,program_name).
attribute(program,requirement).
entity(course,course_name).
attribute(course,credits).
relationship(takes,[student,course]).
relationship(has_finished,[student,course]).
relationship(has_dropped,[student,course]).
relationship(graduated_in,[student,program]).


% dynamic schema - operations to produce events

added(X,Y) :- /added(X,Y).
deleted(X,Y) :- /deleted(X,Y).
/added(nil,dummy).
/deleted(nil,dummy).

operation(offer(C,N)).
added(course(C),offer(C,N)).
added(credits(C,N),offer(C,N)).
precond(offer(C,N),/(not(course(C)))).

operation(enroll(S,C)).
/added(student(S),enroll(S,C)).
/added(credits_won(S,0),enroll(S,C)) :-
  not credits_won(S,_).
added(takes(S,C),enroll(S,C)).
precond(enroll(S,C),
  (course(C),
   /(not has_finished(S,C)),/(not has_dropped(S,C)))).

operation(transfer(S,C1,C2)).
deleted(takes(S,C1),transfer(S,C1,C2)).
added(has_dropped(S,C1),transfer(S,C1,C2)).
added(takes(S,C2),transfer(S,C1,C2)).
precond(transfer(S,C1,C2),
  (course(C2),/takes(S,C1),
   /(not has_finished(S,C2)),/(not has_dropped(S,C2)))).

operation(cancel(C)).
deleted(course(C),cancel(C)).
deleted(credits(C,N),cancel(C)).
precond(cancel(C),
  not takes(S,C):(student(S),takes(S,C))).

operation(change_cr(C,N1,N2)).
deleted(credits(C,N1),change_cr(C,N1,N2)).
```

```
added(credits(C,N2),change_cr(C,N1,N2)).
precond(change_cr(C,N1,N2),credits(C,N1)).

operation(pass(S,C,T1,T2)).
deleted(credits_won(S,T1),pass(S,C,T1,T2)).
deleted(takes(S,C),pass(S,C,T1,T2)).
added(has_finished(S,C),pass(S,C,T1,T2)).
added(credits_won(S,T2),pass(S,C,T1,T2)).
precond(pass(S,C,T1,T2),(credits_won(S,T1),takes(S,C))) :-
    credits(C,N),
    T2 #= T1 + N, T1 #>= 0.

operation(create_program(P,R)).
added(program(P),create_program(P,R)).
added(requirement(P,R),create_program(P,R)).
precond(create_program(P,R),/(not program(P))).

operation(receive_degree(S,P)).
added(graduated_in(S,P),receive_degree(S,P)).
precond(receive_degree(S,P),
  (requirement(P,T),credits_won(S,T),/(not takes(S,_)))).
```

% templates for the update operations of the example

```
op_template(offer(Co,Cr),['course ',Co,' created with ',Cr,' credits.']).
op_template(enroll(S,Co),['student ',S,' enrolled in course ',Co,'.']).
op_template(transfer(S,Co1,Co2),['student ',S,
                              ' transferred from course ',Co1,
                    ' to course ',Co2,'.']).
op_template(cancel(Co),['course ',Co,' cancelled.']).
op_template(change_cr(Co,Cr1,Cr2),['number of credits of course ',Co,
                                   ' changed from ',Cr1,' to ', Cr2,'.']).

op_template(pass(S,C,T1,T2),
  ['student ',S,', having passed course ',C,', has a total of ',T2,' credits.']).
op_template(create_program(P,R),
  ['program ',P,' is open, requiring a total of ',R,' credits.']).
op_template(receive_degree(S,P),['student ',S,' has graduated in program
',P,'.']).
```

% possible initial state to be tried in memory

```
course('Art').
credits('Art',2).
course('Semiotics').
credits('Semiotics',3).
course('Design').
credits('Design',1).
program('Alpha').
requirement('Alpha',5).
program('Beta').
requirement('Beta',4).
```