

A Declarative Conceptual Modelling Language

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

¹Departamento de Informática
Pontifícia Universidade Católica do RJ
R. Marquês de S. Vicente, 225
22.453, Rio de Janeiro, RJ - Brasil

²Centro Científico Rio
IBM Brasil
Caixa Postal 4624
20.001, Rio de Janeiro, RJ - Brasil

ABSTRACT

A declarative conceptual modelling language, implemented as an extension to Prolog, is described. The language is based on an extended version of the entity-relationship (ER) model for the declaration of the information classes and the formulation of queries, and adopts an abstract data type (ADT) approach to define and execute application-oriented update operations.

The language is an integral part of a workbench that provides rapid prototyping at the conceptual level and that supports expert level features. A simple example to illustrate the direct use of the workbench over a database / knowledge base application is also included.

1. INTRODUCTION

This paper describes a declarative conceptual modelling language that is part of a workbench to support the direct use of knowledge base/database applications, as well as to serve as a foundation for expert level features to be developed over these applications.

The language follows an extended version of the entity-relationship (ER) model for the declaration of the information classes and the formulation of queries, and adopts an abstract data type (ADT) approach to define and execute application-oriented update operations. The language is implemented as an extension to Prolog, following a declarative style, in the sense that every aspect of an application is declared with the help of facts and clauses, including the update operations.

In this ER/ADT information/operation model, individual entity instances retain their identity across the different classes to which they may belong (via the *is_a* hierarchy), with respect to their existence, attributes and participation in relationship instances. Also, in the spirit of abstract data types, update requests are limited to the utilization of application-oriented operations.

The workbench permits rapid prototyping of the ER design, that is, the workbench does not treat the ER design as a mere documentation of the application, but as an executable specification. The workbench also provides a better basis for expert level features, because their specification can take advantage of the richer ER/ADT semantics. As a case where the workbench proved to be a viable alternative to

support expert level features (a topic that will not be further discussed here), we can mention its use in the context of project NICE [CF,HCF], whose objective is the design of *cooperative interfaces* to information systems [We,AP,BJ]. The workbench also contains a transparent SQL interface and a query-the-user facility, described in [Fu2].

This paper is organized as follows. Sections 2 to 4 present the syntax for the structural aspects of the languages. Sections 5 and 6 discuss queries and updates. Section 7 illustrates the direct use of the workbench over an example. Section 8 contains the conclusion. Finally, Appendix A lists the complete specification of the example.

2. FACTS AND FACT FRAMES

A *fact* denotes the existence of either an entity or a relationship instance, or captures that one such instance has a certain value for a given attribute. In the prototype, whenever the same attribute name is used in the definition of more than one entity or relationship class, it implies that the attribute will have the same domain. A *key* is an identifying attribute, in the sense that entity or relationship instances that have the same value for the key are indeed the same instance, regardless of the class to which such instances belong. In the current prototype, entity instances cannot have compound keys, i.e. keys consisting of more than one attribute. The key of a relationship instance, on the contrary, is in general compound, since it consists of the keys of the participating entity instances; an exception is the case of binary one-to-n relationship instances, whose key is that of the determining participant (i.e. the participating entity depicted on the "n side" in the ER diagram).

A *database* for a conceptual schema is a set of facts. The syntax for facts is:

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

where <participants list> is a list of pairs of the form <entity class>φ<key>.

To refer to more than one attribute of an entity or relationship instance, a frame construct can be used:

```
<entity class>φ<key> has <attribute frame>  
<relationship class>#<participants list> has <attribute frame>
```

where <attribute frame> is a list of <attribute>:<value> pairs. If <attribute> is multivalued, then <value> will unify with one of the values the attribute currently has, and with the other values upon backtracking.

If this is not the appropriate behavior, a different construct can be used:

```
<entity class>φ<key> has_gr <attribute frame>  
<relationship class>#<participants list> has_gr <attribute frame>
```

where <attribute frame> contains a pair, <attribute>:<value>, if the attribute is single valued, or a pair <attribute>:<list of values>, if the attribute is multi-

valued. In the latter case, <list of values> naturally is the list of values <attribute> currently has.

3. CLASSES OF FACTS

The conceptual schema of a database at the ER/ADT level is specified through clauses that define the entity and relationship classes that exist and the structure of the *is_a* hierarchy. Relationships of arbitrary arity are allowed and binary one-to-n relationships are singled out. The syntax of the clauses to declare the conceptual schema is:

```
entity(<entity class>,<key>)
is_a(<entity class>,<entity class>)
relationship(<relationship class>,<participant classes>)
one_to_n(<relationship class>,<determining participant class>)
attribute(<entity class>,<attribute>)
attribute(<relationship class>,<attribute>)
domain(<attribute>,<value variable>,<validity check>,<cardinality>)
```

where <participant classes> is a list of <entity class> elements, <validity check> is an expression involving <value variable> to define the possible values that can be associated with <attribute>, and <cardinality> is either "single" or "multi", to distinguish between single and multi-valued attributes.

Attributes and participation in relationships are inherited along the *is_a* hierarchy. The current prototype does not provide mechanisms to avoid ambiguities in case of inheritance from more than one parent class, or when a class inherits an attribute also defined in the class.

4. DATA STRUCTURE DECLARATION AND MAPPING

Facts are stored in relational data structures, which can take the form of ground unit clauses of Prolog predicates or tuples of SQL tables. In both cases, the relational schema is declared by clauses of the form:

```
structure(<structure name>,<attribute list>)
```

where <structure name> is either a predicate symbol or the name of an SQL table. The names of the structures to be handled as SQL tables should be indicated in a clause:

```
sql_structures(<list of structure names>)
```

To ease the mapping between ER/ADT and relational schemas, the current prototype requires that the names of the columns of SQL tables be the same as the names of the corresponding attributes. On the other hand, the names of the data structures (predicates or tables) are arbitrary. The mapping between the two schemas is established by clauses with the following format:

```
ent_structure(<entity class>,<name of data structure>)
rep_ent_structure(<entity class>,<name of data structure>)
```

```
rel_structure(<relationship class>, <name of data structure>)  
rep_rel_structure(<relationship class>, <name of data structure>)  
ext_ent_structure(<entity class>, <relationships>, <name of data structure>)
```

where <relationships> is a list of <relationship class>.

The motivation for these different clauses comes from the way we design relational structures to accommodate the entity-relationship facts. Exactly one data structure, designated respectively by an "ent_structure" or "rel_structure" clause, must correspond to each entity or relationship class, storing the key attributes together with all the single-valued attributes. For each multi-valued attribute, there must be a data structure, indicated in a "rep_ent_structure" or "rep_rel_structure", containing only the key and the attribute involved. Finally, whenever an entity class E participates in one or more one-to-n relationship classes, the data structure of E is extended to also represent the relationships. In such cases, an "ext_ent_structure" clause (instead of an "ent_structure" clause) is used to designate the data structure. A detailed description of the design method is found in [TCF].

5. OPERATIONS OVER FACTS

In the spirit of abstract data types, the only way to update facts is through pre-defined application-oriented operations. Following a convenient STRIPS-like scheme [FN,LA], each operation O is specified by a set of clauses, which indicate the facts that are added and deleted by O (i.e., the effects of O) and the preconditions for the execution of O, in terms of logical expressions involving facts that should or should not hold. The syntax of the clauses to specify operations is:

```
<operation>(<name of operation>, <parameter list>)  
added(<fact>,<operation>) <- <antecedent>  
deleted(<fact>,<operation>) <- <antecedent>  
precond(<operation>,<expression involving facts>) <- <antecedent>
```

where <parameter list> consists of the names of the attributes to which the parameter values must belong. An "operation" clause provides the *signature* of an operation, and the designer must ensure its consistency with the other clauses referring to the operation. In the "added", "deleted" and "precond" clauses, the <antecedent>, which is a Prolog expression, is often omitted. When present, it provides additional criteria to check whether the clause is applicable and contributes to the instantiation of variables appearing in the head of the clause. Notice that the Prolog expression may in particular refer to other such clauses and to database facts. Of special interest is the case of the antecedent expression of a "precond" clause of an operation O referring to "added" and "deleted" clauses of O; in such cases, the "precond" clause may indeed express a post-condition rather than a precondition, since it is allowed to look at the effects that the execution of O would have.

Preconditions are used to enforce integrity constraints dynamically, in the sense that they restrict the application of the defined operations to guarantee that they can only lead to valid states.

In adherence to the original ADT principles, operations do not "belong" to classes, as happens with strict object-oriented systems. Instances of several classes may be

affected by an operation that refers to them through its parameters. As a consequence, inheritance of operations along the `is_a` hierarchy is provided in a trivial way. To see why this is true, assume the existence of an instance `i` of an entity class `E`, such that `E is_a F`. Assume further that an operation `O` includes as one of its parameters a reference to an instance of class `F`. Then, since we require that instances of an entity class must also exist as instances of all classes located above it in the `is_a` hierarchy, we conclude that `O` is applicable to `i` simply because `i` is also an instance of `F`.

As a related point that can be illustrated by further elaborating the above example, consider the specification of an operation `O'` this time referring to instances of `E`. Suppose that we want the effects of `O'` to subsume the effects of `O`, in the sense that `O'` has all the effects of `O` plus some others. The indication of subsumed effects can be succinctly done by including either or both of the following clauses in the definition of `O'`:

```
added(F, O') <- added(F, O)
deleted(F, O') <- deleted(F, O)
```

the same provision being possible for preconditions, through the inclusion of "precond" clauses of an analogous format.

In addition to the `precond`, `added` and `deleted` clauses belonging to a specific application, there may be present a number of general (i.e. application-independent) clauses of these types distinguished by the prefix "sys". The current version of the prototype contains "sys:precond" clauses establishing that:

- P1.** an instance of an entity-class `E` such that `E is_a F` can be added only if the instance exists in class `F`
- P2.** a value of an attribute of an entity or relationship instance can only be added if the instance exists
- P3.** a relationship instance can be added only if all participating entity instances exist

Clauses of type "sys:deleted" are also included, establishing that:

- D1.** if an instance of an entity-class `E` is deleted, then it is also deleted from all entity-classes `F` such that `F is_a_ E` (letting "is_a_" be the transitive closure of "is_a")
- D2.** if an instance of an entity or relationship class is deleted then all its attributes are also deleted
- D3.** if an instance of an entity-class is deleted then all relationship instances where it participates are deleted

These defaults are based on assumptions that are often adopted with the entity-relationship model. Broadly speaking, they preserve integrity constraints inherent in the model. The "sys:precond" clauses *restrict* additions, whereas the "sys:deleted" clauses *propagate* deletions. The presence of these "sys" clauses reduces the number of clauses that an application designer has to introduce for each operation. On the other hand, the designer can make a "sys:precond" clause vacuous for a specific operation `O` by simply providing an appropriate "precond", "added" or "deleted" clause in the definition of `O`. For example, pre-condition P2 becomes vacuous, if an

operation O that is allowed to add a value for an attribute of an instance, also adds the instance itself. Similarly, the propagation of deletions can be changed into blocking for an operation O by attaching a "precond" clause to O that enforces the blocking of the operation. For example, the designer may include a "precond" clause preventing the deletion of an entity instance, if a certain attribute of the entity is still defined, or if the instance still participates in some instance of a specified relationship class.

A few "sys:added" and "sys:deleted" clauses were included to handle certain situations where null values are involved. Although these clauses are meaningful at the conceptual level, since nulls are used here to express undefined values, we must point out that their presence is mainly justified to ensure the correct mapping of the ER facts into the relational structures. In our STRIPS-based method to define operations, a "deleted" clause is the way to indicate that an operation O causes, as one of its effects, a single-valued attribute A of an entity or relationship instance to become undefined. A "sys:added" clause complements the deletion of the current value of A , by assigning to it the null value. Conversely, the addition of a value to a currently undefined attribute is complemented by the removal of its null value, through a "sys:deleted" clause. Note that, in the present prototype, to replace a non-null value of a single-valued attribute by another non-null value, both a "deleted" and an "added" clause must be provided. One-to- n relationships are treated in about the same way as single-valued attributes. The removal of an one-to- n relationship instance, which of course entails the removal of all its attributes, is complemented through "sys:added" clauses to indicate (by inserting nulls) that the participant on the "one side" and the single-valued relationship attributes have become undefined. Notice that, if this participant is replaced by another one, rather than removed, the current relationship attributes are equally removed. Finally, a "sys:deleted" clause provides the deletion of a null denoting an undefined participant when a valid participant is added.

We have still two more "sys:precond" clauses to mention. They implement our strategy (proposed in [VF]) to handle operations in case some of its effects already hold. These clauses prevent the execution if one or more facts that the operation should add are already present in the database or if facts to be deleted are absent. We find that this "all or nothing" strategy is compatible with the notion of *database transactions*, where several commands are involved and there is no commitment with respect to database updates if any failure occurs.

6. QUERY AND UPDATE REQUESTS

Over an ER/ADT database, a user can formulate *query requests* and *update requests* as Prolog goals. For queries, a goal would consist of a Prolog expression involving one or more facts with the syntax described in section 2.

If a query refers to an attribute of an entity or relationship instance and, although the instance exists, the value of the attribute is currently undefined, the query fails as would be expected. However, we decided that the prototype should allow queries on undefined attributes declared as single-valued to succeed in the special case where the query mentions the "null" value explicitly.

The frame construct is convenient in the formulation of queries if more than one attribute is mentioned in connection to the same entity or relationship instance. Frames can be used in flexible ways. If a term corresponding to a frame is indicated by a variable, the execution of the goal will instantiate the variable to a list involving all attributes of the given entity or relationship instance which have non-null values in the database. If the user is only interested in a few specific attributes, he may indicate the frame explicitly as a list containing the desired attributes in any order he chooses, paired with variables to be instantiated with the corresponding values; in this case, for attributes whose value is not defined the respective variables will remain uninstantiated. Powerful operations have been introduced for frames, especially unification and generalization [Fu1]. Moreover, a query with frames has a better performance than a conceptually equivalent query where attributes of the same instance are indicated separately, since by working on entire frames the prototype is able to collapse database accesses so that each access retrieves all values requested that happen to be kept in the same underlying data structure.

Query requests can also involve schema information. All types of declarative clauses described in sections 3 and 5 (and even section 4, if one needs to reach a lower level) can appear in goal expressions.

Update requests are effected by goal expressions containing calls to the defined operations. Although, syntactically, these calls are direct, they are actually intercepted by a meta-predicate "exec_op" which checks the values of the parameters that are not variables or "null"s, tests the preconditions and, in case of success, applies additions and deletions to the appropriate data structures to reflect what the added and deleted clauses specify.

At the beginning of a session, where query and update requests will be posed, two preparatory goals must be executed:

```
<- enable_structures().  
<- enable_operations().
```

the effect of the former being that the "sql_fact" predicate of the PSQL tool is applied (as described in [Fu2]) to all structures in the "sql_structures" clause, whereas the effect of the latter is to add to the workspace clauses of the form:

```
<operation template> <- exec_op(<operation template>)
```

where <operation template> consists of the operation name followed by a parenthesized sequence of variables denoting the formal parameters of the operation. The ability to enter calls to operations directly, that we mentioned earlier in this section, results from the presence of these clauses.

7. EXAMPLE OF DIRECT UTILIZATION OF THE WORKBENCH

This section briefly describes an application and illustrates the power of the query language. Appendix A contains the complete description of the example as it runs under the Prolog prototype.

The conceptual level specification defines entity classes that correspond to employees, trainees, departments, projects and clients, where trainees are a sub-class of employees. It also defines relationship classes capturing that employees work in departments and participate in projects, and that clients sponsor departments in view of specific projects. Furthermore, the specification contains integrity constraints requiring that an employee can work in only one department and that he can only participate in sponsored projects of his department.

The mapping between the conceptual level specification and the relational data structure level specification has the following properties: it keeps the data on employees and on departments in SQL tables; it embeds the "works" one-to-n relationship in the "emp" table, together with the attributes of employees; and it maintains the attribute "task" of relationship "participates", which is multivalued, in a separate table.

The application has operations to install a department indicating the city where its headquarters will be, to hire employees to work in a department, to hire trainees, to separately designate the job that an employee will have in his department, to raise an employee's salary, to fire an employee, to propose a project, to associate in a sponsorship contract a client and a department with respect to a project, to assign employees to projects, to add more tasks to assigned employees, to give final approval to a project, and a few others.

Some features in the definition of operations deserve comments (we refer the reader to Appendix A). The assign operation has a precondition saying that an employee E can be assigned to a project P only if E works in a department that sponsors P. The salary raise operation affects only the salary of an employee, by adding the indicated amount (to reflect this update, only one field of the appropriate "emp" tuple is changed). When a project is initially proposed, it is marked as pending, a condition that can be later removed by an execution of the approve operation issued by the Projects Control Department, say (this removal is implemented by setting to "null" the second field of the corresponding "pr" clause. The definition of operation to hire trainees includes an "added" clause concisely declaring that the operation adds all facts added by the operation that hires employees.

Suppose that the database is initially empty and that the following operations are executed:

```
G1. <- install('D1', 'NY').
G2. <- hire('McCoy', 100, 'D1').
G3. <- designate('McCoy', 'chair').
G4. <- propose('Alpha').
G5. <- associate('Spock Ltd.', 'D1', 'Alpha', 1991, 'c123').
G6. <- hire_tr('Savik', 80, 'D1', 'graduate').
G7. <- assign('Savik', 'Alpha', 'record-keeping').
G8. <- add_task('Savik', 'Alpha', 'communications').
```

From the definition of the operations in Appendix A, the reader may find what facts will start to hold or cease to hold when these goals are executed, and how the data structures will be updated. In particular, the reader may appreciate the consequences of the application-independent clauses (prefixed with "sys") that establish

general preconditions and effects of operations. For instance, if the goal "`<-fire('Savik')`" is executed, the direct effect is that Savik ceases to exist as an employee, but the "sys" clauses will also make her cease to exist as a trainee, and all facts related to attributes of this entity instance in both entity classes, as well as of its participation in relationships, will be also removed.

By way of an example, we follow the execution of G6. Recall from the appendix the definition of "hire" and "hire-tr":

```
H1. operation(hire, [name,sal,dname]).
H2. added(empΦN, hire(N,S,D)).
H3. added(empΦN\sals(S), hire(N,S,D)).
H4. added(works#[empΦN,deptΦD], hire(N,S,D)).
H5. operation(hire_tr,[name,sal,dname,level]).
H6. added(F, hire_tr(N,S,D,L)) <- added(F, hire(N,S,D)).
H7. added(traineeΦN, hire_tr(N,S,D,L)).
H8. added(traineeΦN\level(L), hire_tr(N,S,D,L)).
```

The execution of goal G6, "`<- hire_tr('Savik',80,'D1','graduate')`", directly creates the following new facts, via H7 and H8:

```
F1. added(traineeΦ'Savik', hire_tr('Savik',80,'D1','graduate')).
F2. added(traineeΦ'Savik'\level('graduate'),
          hire_tr('Savik',80,'D1','graduate')).
```

and, indirectly, the following new facts, via H6 and H2, H3 and H4:

```
F3. added(empΦ'Savik', hire('Savik',80,'D1')).
F4. added(empΦ'Savik'\sal(80), hire('Savik',80,'D1')).
F5. added(works#[empΦ'Savik',deptΦ'D1'], hire('Savik',80,'D1')).
```

The conceptual information expressed by F1 through F5 is in fact stored, via the mapping clauses, as the following set of ground unit clauses (but recall that it is in part physically stored as SQL tuples):

```
R1. emp('Savik',80,'D1',null).
R2. tr('Savik','graduate').
```

The complete database at the end of the execution of the operations also contains the clauses:

```
R3. dept('D1','NY').
R4. emp('McCoy',100,'D1','chair').
R5. pr('Alpha',true).
R6. cIn('Spock Ltd.','new').
R7. spon('Spock Ltd.','D1','Alpha',1991,'c123').
R8. part('Savik','Alpha').
R9. tsk('Savik','Alpha','record-keeping').
R10. tsk('Savik','Alpha','communications').
```

Sample queries, together with the result they produce against this database, now follows (notice that queries (3) and (5) use the frame construct):

```
(1) query:      who works in department D1?
    in Prolog: <- forall(works#[empΦN,deptΦ'D1'], write(N)).
    answer:     'McCoy', 'Savik'
```

- (2) query: to what entity classes does Savik belong?
in Prolog: <- forall(E:'Savik', write(E)).
answer: emp, trainee
- (3) query: give all information available on Savik, as trainee.
in Prolog: <- traineeφ'Savik' has F & write(F).
answer: [level:graduate, sal:80]
- (4) query: is there some employee whose job is still undefined?
in Prolog: <- works#[empφN,deptφD]\job(null) & write(N-D).
answer: 'Savik' - 'D1'
- (5) query: which tasks have been assigned to Savik in project Alpha?
in Prolog: <- participates#[empφ'Savik',projφ'Alpha'] has_gr F
& write(F).
answer: [task: ['communications','record-keeping']]
- (6) query: is it true that project Alpha is sponsored for 1991?
in Prolog: <- sponsorsφ[clientφ*,deptφ*,projφ'Alpha']\year(1991)
& write(yes).
answer: yes
- (7) query: has project Alpha been approved already?
in Prolog: <- (¬ projφ'Alpha'\pending(*) & write(yes)
| prst('still pending') & nl).
answer: still pending

Until now we have only considered a factual database in the present example. Knowledge bases would, in addition, include *rules*. To close this section, we introduce a rule establishing that a project is "ongoing", in the sense that its execution is under way, if it is being sponsored for the current year and it is no longer pending. Besides the rule, we assume some way to indicate the current year, which could be an access to the system's internal clock or a unit clause. The Prolog declarations follow. As a step towards a pseudo-natural language notation, "ongoing" is introduced as a prefix operator, obviating the need for the special symbols used at the ER/ADT level:

```
op("ongoing",prefix,50).
```

```
current(1991).
```

```
ongoing P <-
  current(Y) &
  sponsors#[clientφ*,deptφ*,projφP]\year(Y) &
  ¬projφP\pending(true).
```

Given the state of the database captured in clauses R1 through R10, the query request

```
<- ongoing 'Alpha'.
```

will fail, since the project is indeed currently sponsored but it is still pending.

8. CONCLUSION

We described a declarative conceptual modelling language that is an integral part of a workbench that provides rapid prototyping at the conceptual level and that supports expert level features. We also provided a simple example to illustrate the direct use of the workbench over a database / knowledge base application. Exploration of its use in connection with expert level features is under way as part of project NICE [CF, HCF], whose objective is to provide cooperative interfaces to information systems.

The prototype of the workbench is at an early stage of development, but it already implements all features of the language here described. It can be extended in several ways, either as a consequence of enriching the ER/ADT model, or to focus on the optimization of the algorithms and their implementation, among other points.

REFERENCES

- [AP] J. F. Allen and C. R. Perrault, "Analyzing intentions in utterances", *Artificial Intelligence* 15:3 (1980), 143-178.
- [BJ] L. Bolc and M. Jarke (eds.), *Cooperative Interfaces to Information Systems*, Springer-Verlag (1986).
- [CF] M. A. Casanova and A. L. Furtado, "An Information System Environment based on Plan Generation", Proc. Int. Working Conference on Cooperating Knowledge based Systems, Keele, UK (1990).
- [FN] R. E. Fikes and N. J. Nilsson - "STRIPS: a new approach to the application of theorem proving to problem solving" - *Artificial Intelligence* 2 (1971) 189-208.
- [Fu1] A. L. Furtado - "Exploring the extensibility of IBM Prolog" - technical report CCR-124 - Rio Scientific Center of IBM Brasil (1991).
- [Fu2] A. L. Furtado - "Two integrated tools for IBM Prolog: query-the-user & transparent use of SQL" - technical report CCR-126 - Rio Scientific Center of IBM Brasil (1991).
- [HCF] A. S. Hemerly, M. A. Casanova and A. L. Furtado, "Cooperative behaviour through request modification", Proc. 10th Int'l. Conf. on the Entity-Relationship Approach, San Mateo, CA, USA (1991) 607-621.
- [LA] D. J. Litman and J. F. Allen - "A plan recognition model for subdialogues in conversations" - *Cognitive Science* 11 (1987) 163-200.
- [TCF] L. Tucherman, M. A. Casanova and A. L. Furtado - "The CHRIS consultant - a tool for database design and rapid prototyping" - *Information Systems* 15:2 (1990).
- [VF] P. A. S. Veloso and A. L. Furtado - "Towards simpler and yet complete formal specifications" - in "Information systems: theoretical and formal aspects" - A. Sernadas, J. Bubenko and A. Olive (eds.) - North-Holland Pub. Co. (1985) 175-189.

[We] B. L. Webber, "Questions, answers and responses: interacting with knowledge base systems", in *On knowledge base management systems*, M.L. Brodie and J. Mylopoulos (eds.) - Springer (1986).

APPENDIX A

```
% === FILE: EXAMP PROLOG ===
% EXAMPLE DATABASE / KNOWLEDGE BASE

% declaring entity classes and attributes

entity(emp,name).
entity(trainee,name).
entity(dept,dname).
entity(proj,pname).
entity(client,cname).

trainee is_a emp.

attribute(emp,sal).
attribute(trainee,level).
attribute(dept,city).
attribute(proj,pending).
attribute(client,status).

relationship(works, [emp,dept]).
relationship(participates, [emp,proj]).
relationship(sponsors, [client,dept,proj]).

one_to_n(works, emp).

attribute(works,job).
attribute(sponsors,contract).
attribute(sponsors,year).
attribute(participates,task).

domain(name,V,is_string(V),single).
domain(dname,V,is_string(V),single).
domain(pname,V,is_string(V),single).
domain(cname,V,is_string(V),single).
domain(sal,V,is_num(V),single).
domain(level,V,
    V == 'graduate' | V == 'undergraduate',single).
domain(city,V,is_string(V),single).
domain(pending,V,V == true,single).
domain(status,V,is_string(V),single).
domain(job,V,is_string(V),single).
domain(contract,V,
```

```
    stconc('c',N,V) & st_to_at(N,M) & is_int(M),single).
domain(year,V,in_range(V,1900,2000),single).
domain(task,V,is_string(V),multi).
```

```
% declaring data structures
```

```
structure(emp, [name,sal,dname,job]).
structure(dept, [dname,city]).
structure(pr, [pname,pending]).
structure(tr, [name,level]).
structure(acc, [name,account]).
structure(part, [name,pname]).
structure(tsk, [name,pname,task]).
structure(cIn, [cname,status]).
structure(spon, [cname,dname,pname,year,contract]).

sql_structures([emp,dept]).
```

```
% mapping between classes and data structures
```

```
ext_ent_structure(emp, [works], emp).
ent_structure(trainee, tr).
ent_structure(dept, dept).
ent_structure(proj, pr).
ent_structure(client, cIn).
```

```
rel_structure(participates, part).
rel_structure(sponsors, spon).
```

```
rep_rel_structure(participates, tsk).
```

```
% defining operations
```

```
operation(install, [dname,city]).
added(dept@D, install(D,C)).
added(dept@D\city(C), install(D,C)).
```

```
operation(propose, [pname]).
added(proj@P, propose(P)).
added(proj@P\pending(true), propose(P)).
```

```
operation(approve, [pname]).
deleted(proj@P\pending(true), approve(P)).
```

```
operation(hire, [name,sal,dname]).
added(emp@N, hire(N,S,D)).
added(emp@N\sals(S), hire(N,S,D)).
added(works#[emp@N,dept@D], hire(N,S,D)).
```

```
operation(hire_tr,[name,sal,dname,level]).
added(F, hire_tr(N,S,D,L)) <- added(F, hire(N,S,D)).
```

```

added(traineeφN, hire_tr(N,S,D,L)).
added(traineeφN\level(L), hire_tr(N,S,D,L)).

operation(raise,[name,increment]).
added(empφN\sai(S), raise(N,I)) <-
  empφN\sai(S0) & S := S0 + I .
deleted(empφN\sai(S0), raise(N,I)) <-
  empφN\sai(S0).

operation(designate, [name,job]).
added(works#[empφN,deptφ*]\job(J), designate(N,J)).
deleted(works#[empφN,deptφ*]\job(J), designate(N,K)) <-
  works#[empφN,deptφ*]\job(J).

operation(fire, [name]).
deleted(empφX, fire(X)).

operation(assign, [name,pname,task]).
precond(assign(N,P,T),
  works#[empφN,deptφD] &
  sponsors#[clientφ*,deptφD,projφP]).
added(participates#[empφN,projφP], assign(N,P,T)).
added(participates#[empφN,projφP]\task(T), assign(N,P,T)).

operation(add_task, [name,pname,task]).
added(participates#[empφN,projφP]\task(T), add_task(N,P,T)).

operation(associate, [cname,dname,pname,year,contract]).
added(clientφC,associate(C,D,P,Y,Cn)) <- ~clientφC.
added(clientφC\status('new'),associate(C,D,P,Y,Cn)) <-
  ~clientφC\status(*).
added(sponsors#[clientφC,deptφD,projφP],
  associate(C,D,P,Y,Cn)).
added(sponsors#[clientφC,deptφD,projφP]\year(Y),
  associate(C,D,P,Y,Cn)).
added(sponsors#[clientφC,deptφD,projφP]\contract(Cn),
  associate(C,D,P,Y,Cn)).

% example of a knowledge-base rule

op("ongoing",prefix,50).

current(1991).

ongoing P <-
  current(Y) &
  sponsors#[clientφ*,deptφ*,projφP]\year(Y) &
  ~projφP\pending(true).

```