Abstract

EPSILON is a Knowledge Base Management System (KBMS) based on the integration of database technology and logic programming paradigm. Its aim is to provide an environment offering at the same time powerful structuring mechanisms, such as modularity and inheritance links, and efficient and transparent access to relational database systems.

The EPSILON logic programming environment allows the structuring of large knowledge bases expressed in logic languages using so-called theories. One of the most interesting features of the EPSILON prototype is the ability to use data stored in a relational database from a logic program. The EPSILON KBMS relies on systematic use of metaprogramming combined to optimization techniques. One of them, partial evaluation, is specially valuable when applied to PROLOG-DBMS communication.

This work describes an interface implemented in this environment between PROLOG and a relational DBMS. In order to combine the advantages of interpreted and compiled coupling strategies, the hybrid interface (half-compiled-half-interpreted) uses a two phase (static-dynamic) communication protocol. This strategy minimizes the accesses to the database and generates queries as selective as possible to the database by maximal instantiation of the database predicates using DBMS for relational operations whenever possible. This interface is integrated into EPSILON by defining it as a new inference engine which allows the user to use logical predicates and database predicates in the same way.

Keywords: logic programming, coupling PROLOG-DBMS, object modeling

I. Introduction

Since the beginning of eighties, a number of works has been devoted to coupling rule based systems using the logic programming paradigm to relational database systems. Special attention has been paid to both recursion and negation handling. Several commercial PROLOGs offer an interface to Database Management Systems (DBMS) at different levels of transparency, starting from the use of PROLOG as host language and going to PROLOG systems in which the location of data is completely hidden to the user.

However these systems suffer of two major drawbacks. First of all, inefficiency due to the fact that relational operations as selections or joins are made at PROLOG level rather than by the DBMS. Secondly, standard PROLOG is the unique paradigm offered to the user, which can be restrictive when non standard reasoning is needed.

EPSILON is a prototype developed in the context of the European ESPRIT project. It is built on top of commercial PROLOG (BimiPROLOG) and DBMS (INFORMIX), running on standard Unix environment [11, 7].

The objective of EPSILON is to provide:
- an extension of PROLOG with theories (multiple worlds), allowing to structure PROLOG programs, by defining links between theories,
- transparent fact retrieval interfaces between PROLOG and relational DBMS, using several optimization techniques,
- metaprogramming as a basic technique to define new inference engines and tools (debugger, tracer, explanator, etc...),
- a graphical user interface on a personal computer (e.g. one window for one theory).

In a first prototype, a so-called compiled approach has been implemented for the coupling of PROLOG with Relational DBMS [3, 9]. An enhanced version has been developed using a hybrid approach, so called because it is half-compiled, half-interpreted [4], [8], [10], [11], combining the advantages of interpretative and compiled techniques.

The aim of this paper is to present the salient features of this latter version. It is organized as follows: the PROLOG-DBMS interface, which is based on partial evaluation, is first introduced. Then the basic concepts in EPSILON are presented, showing the use of theory, links and engine theory. Finally, the integration of the database interface in the theory concept is described.

II. The PROLOG-SGBD Interface

The first prototype has been developed, using a compiled approach, to check ideas and to be benchmarked. It is distributed DBMS oriented [9].

The compiled approach consists in treating, for a given user's goal, the PROLOG program in its integrality, in order to split what refers to the inference engine from what refers to the database. After compilation, and translation of the database part to Data Manipulation Language queries, the relational database is accessed only once. Once the facts are loaded in main memory, the final evaluation can start. As a consequence, PROLOG's standard strategy is modified:

Let's suppose a PROLOG program refers to database relations, by the means of edb-predicates noted edb(Rk(x1,...,xn)).

Partial evaluation [5, 14] is a source to source compilation technique, compiling such a PROLOG program. It consists in exploring the resolution tree by freezing some of the sub-goals literals, possibly including the edb-predicates. Edb-predicates are collected, built-ins are evaluated whenever possible, and variables are instantiated (to minimize the number of extracted facts from the database).

The Wisconsin benchmark for DBMS has been applied to this prototype. Measurements are done on a SUN 3.50 workstation. General conclusions are:

- partial evaluation and activant generation (see below) modules are efficient (20% and 17% of the response time respectively in the worst cases);
- the distribution is costly (up to 70% of the response time to a request in the worst case);
- the compiled approach is not suited for recursion handling.
The unification time may dramatically increase for recursive queries (up to 30%), because loading the whole relation may result in:
- a large set of facts in PROLOG memory
- useless facts
- useless inferences

The hybrid approach palliates the weaknesses of the first prototype: useless facts and useless inferences. The aim is to solve both efficiency and recursion handling problems.

The hybrid coupling approach is composed by a compiled part followed by an interpreted one (the interpretative approach uses a standard PROLOG resolution strategy), combining the advantages of these two techniques. It minimizes the accesses to the database and generates queries as selective as possible to the database by a maximal instantiation of the edb-predicates using DBMS for relational operations (joins, selections) whenever possible. It offers the user a full PROLOG including functions, cut, recursion, negation and as many built-in predicates as possible.

This approach allows:
- a static compilation of database queries with partial evaluation,
- a dynamic evaluation of database calls with an interclause optimization,
- an history management.

The strategy used is shown in figure 1.

User's programs (P) are transformed by a source to source compiler made of a partial evaluator and a static translator. The role of the static translator is to transform the partially evaluated output program into a program that can be interpreted by PROLOG. The output is still a PROLOG program which is a specialized version of the source program optimized for the evaluation of a given user's query G.

Thus a final meta-interpretation is avoided. However, this needs to use a transparent internal predicate (edb-call) of the form of which is:

\[ \text{edb-call}(\text{activant}^* x_1, x_2,..., x_n, [g_1(x_1),..., g_k(x_n)]) \]

where the "activants" are automatically generated literals that allow to distinguish the various conjunctions of edb-predicates (g_i).

The gl,..., g_n are the edb-predicates and the x_i are terms verifying:

\[ \{x_1,..., x_k\} \cup \ldots \cup \{x_n\} \supset \{x_1,..., x_n\} \]

The edb-call predicate define a set of edb-predicates (g_1, g_2,..., g_n) which will be translated to a single SQL statement corresponding to the activant^* (x_1, x_2,..., x_n).

Let L_i = e_1,..., e_n be a clause of the partially evaluated program.

- \{e_1,..., e_n\} is an edb-list if e_i is an edb-predicate.
- \{e_1,..., e_n\} is an edb-list if each e_i is an edb-predicate and if each e_i+1 (0 \leq i \leq n-1) has a common variable (joins) with at least one of the others e_j (0 \leq j \leq n).

This constraint insures that the database system will not compute the cartesian product.

\[ e_1,..., e_k \] is a maximal edb-list if none of \[ e_1,..., e_k \] and \[ e_l,..., e_{k+l} \] are edb-lists.

To minimize the number of independent calls to the database, the edb-list must be maximal. For example, with the following user's program where country and customer are edb-predicates:

\[ \text{spec_cust(name, town, country)}, \text{country}(-town, country, pop), \text{pop} \geq 1000000, \text{foreign_cust(name, town, country)}, \text{customer(name, town, country)}, \text{country = 'france'} \]

The maximal edb-list is:

\[ \text{[country(town, country, pop), pop} \geq 1000000, \text{[country(town, country), country = 'france'] \}} \]

Each edb-list will be included into an edb-call primitive the generation of which is the purpose of what follows:

Definition:

The list \[ e_1,..., e_n \] is said to be the edb-formula associated to the edb-list \[ e_1,..., e_n \] if every constant in the edb-list is replaced by variables. Two occurrences of a same constant are replaced by two occurrences of a same variable.

For example, an edb-formula associated to

\[ \text{edo(father(x, b)),edo(father(b, y))} \]

is

\[ \text{edo(father(x, z)),edo(father(z, y))} \]

Let L_1 and L_2 be two edb-formulas respectively associated to L_1 and L_2. L_1 and L_2 will generate only one activant if and only if L_1 = L_2 (the names of variables being possibly different).

The hybrid approach is rather similar to the interpreted approach, with however a fundamental difference: in the interpreted approach, each edb-predicate produces a call to the database, while in the hybrid approach, a call is produced by an edb-call.

The final evaluation step of the hybrid approach is processed by the standard PROLOG interpreter. The communication mode used between PROLOG and the DBMS is presented in [2] and [6]. The aim is to forward the requests and the facts between PROLOG and the DBMS via main memory. Operations are done when they are mandatory (opening and closing the database).

The access to DBMS is processed via the edb-call predicate, as shown in the following [11]:
The definition of the predicate \texttt{edb-call}, that processes calls to the database at the final evaluation, includes the optimizer and the \texttt{history} management predicate:

\begin{verbatim}
edb-call(activant*: edb_list):-
    prologue(activant*, edb_list_and_or_list_name,
        list_attributes, list_relations, conditions),
    activate(name_list_attributes, list_relations, conditions),
    fail.

edb-call(activant*),-
call(activant*).
\end{verbatim}

where

- \texttt{prologue} finds parameters necessary for the activate predicate,
- the activate predicate sends the SQL request

\begin{verbatim}
SELECT -list-attributes
FROM list_relations
WHERE conditions
\end{verbatim}

to the database and translates the extracted tuples into PROLOG facts
(having the corresponding activant name) in the main memory.
- \texttt{call} makes it possible to continue the evaluation with the extracted
tacts.

Let's note \(\{A1 \leftarrow \{ t / A(t)\}\} \) where \( t \) is a tuple variable. \(\{A\}\) expresses the set of facts extracted from the database corresponding
to activant \(A\).

Consider for example \texttt{activanto} corresponding to the\texttt{ conjunction: \{father(x,y), father(s,y)\}}
Suppose that

\[\begin{align*}
A1 &= \text{activant}(a, y) \\
A2 &= \text{activant}(x, y)
\end{align*}\]

Clearly, \(A2 \geq A1\) with respect to the extracted facts.
Generating a call for \(A1\) and \(A2\) will result in extracting twice \(A1\) which is incorrect.

The correct solution is to extract \(A2\) and \(A1\) after the extraction of \(A1\) (in terms of sets, it is \(\{ t \mid A2(t), A1(t)\}\)). For the unification point of view, \texttt{activanto} is subsumed by \texttt{activato}(x.y). So a single \texttt{edbcall} is generated, corresponding to the most general literal: \texttt{activant}(x.y).

Consider now, another example for which the compiled approach may load redundant facts in the PROLOG memory:

\begin{verbatim}
spec_order(ordno, itemno, custno) :-
    order(ordno, itemno, custno),
    spec_cust(custno),
    spec_item(itemno).
\end{verbatim}

where only order is a database relation.

For the request

\begin{verbatim}
?spec_order(ordno).
\end{verbatim}

\texttt{activant}(ordno, itemno, custno) corresponding to the \texttt{order} relation is generated.

In a compiled approach, \texttt{order} is loaded twice, regardless to the fact that two clauses may have common answers.

In the hybrid approach, \texttt{order} is first evaluated using instantiations coming from \texttt{spec_cust}. Clause2 is then evaluated generating an SQL statement which discards tuples whose \texttt{custno} values corresponds to \texttt{spec_cust} (interclause optimization).

It is up to an optimization module, which is a sub-routine of \texttt{edbcall}, to take into account already loaded facts when dynamically generating queries to DBMS. This module manages a meta-base which stores every more general encountered activant (more general in the sens of the "subsumption" order relation). A special predicate \texttt{history} is used to compare a current activant \(A_C\) with the more general activants (in the meta-base) having same name.

The translation into SQL of the \texttt{edbcall}-associated to the current activant \(A_C\), plus the instantiation constraints, join constraints and negative instantiation constraints, is processed by the activate predicate. At each database access, tuples are loaded into main memory in the PROLOG fact form, the names of the facts being of course the activant name. The evaluation can then go on.

III. EPSILON Architecture

The most important aspect of the EPSILON Single User Prototype is the possibility to partition and to structure the knowledge base. The basic component of an EPSILON knowledge base is the \texttt{theory}. All objects in the architecture (inference engines, object level theories in different logic languages, data bases), are represented in a uniform way by the \texttt{theory} feature.

Theories are considered as objects and a \texttt{knowledge base (KB)} is a collection of theories together with relations on theories defined by \texttt{links} \cite{12, 13}. The \texttt{link} feature is a flexible way of defining relationships between theories, including the definition of new inference engines in terms of existing ones (compilation of interpreters). In the EPSILON terminology each theory is accompanied by an inference engine (\texttt{theory processor}) which is itself a meta-level theory, providing operations to query, update and search this one and maybe other object-level theories.

The \texttt{knowledge component} in a \texttt{theory} can be described by different knowledge representation techniques depending on the \texttt{theory processor} (e.g. DB tuples, relational algebra expressions, various logic languages). The logic languages are extensions or restrictions of PROLOG, including control and knowledge representation features.

Theories have two important attributes: their class and their type. Every \texttt{theory processor} defines a \texttt{class} of theories. All theories of one class have the same \texttt{theory processor}. The current prototype contains predefined classes: \texttt{kernel}, \texttt{database} and \texttt{deductive database}. The second distinction concerning theories is their \texttt{type}. All theories defining classes (all \texttt{theory processors}) are of \texttt{type engine}, while all other theories are of \texttt{type object}.

Generally, an object theory contains a text (a chunk of knowledge) to be interpreted by an engine theory defining a (meta)interpreter. Inference engines defined by metaprograms in turn need another inference engine. The inference engine of the first theories in the hierarchy of metainterpreters is PROLOG extended with partitioning of the \texttt{workspace} (\texttt{theory handling primitives}), which corresponds to the \texttt{predefined class kernel}.

Databases are included in EPSILON as theories of the \texttt{database class}. A \texttt{database} theory corresponds to a piece of knowledge which is contained in a relational database. Each \texttt{database} theory is associated to a PROLOG data dictionary which is used to show the structure of the relations.

The class \texttt{inheritance} is an engine theory which extends the PROLOG inference mechanism with inheritance between theories. In order to realize the inheritance mechanism, the \texttt{link} feature is exploited.

The \texttt{inheritance} class provides four different inheritance modalities: \texttt{closed is a}, \texttt{open is a}, \texttt{closed consultation}, \texttt{open consultation}. The \texttt{isa} \texttt{link} defines inheritance of clauses between two theories, which must belong to the same class. The \texttt{consult} \texttt{link} (between theories \texttt{T1} and \texttt{T2}) means that the metainterpreter of \texttt{T2} is consulted for solving a subgoal. The two theories can belong to different classes. An inheritance link between two theories \texttt{T1} and \texttt{T2} is \texttt{closed} if \texttt{T2} is involved in the solution of a goal only when a predicate is undefined in \texttt{T1}. An inheritance link between two theories \texttt{T1} and \texttt{T2} is \texttt{open} if each subgoal is first solved in \texttt{T1} only, then using \texttt{T2}, too.

\texttt{Consult} \texttt{links} allow to evaluate queries using another metainterpreter (than the metainterpreter of the theory which sends the query). \texttt{Isa} \texttt{links} are used to model concept hierarchies well-known in knowledge representation languages.

742
All predicates of a theory are public to the other theories. The predicates can be queried directly from the user interface or within a program using the built-in predicate `demo(goal, _theory_name).

The knowledge bases are structured by specifying suitable links between theories. Some predefined links, which define the structure or visibility rules among theories, are handled by the system. The user can also define his own links and manage them with his programs. The knowledge base dictionary holds the information about theories and links.

IV. Integrating the PROLOG-DBMS interface in a theory based architecture

The kernel-DBMS interface defines the deductive database (ddbt) class in EPSILON [11]. All theories of the ddbt class, containing the user's program, have this metainterpreter (which uses a two phase -static-dynamic- communication protocol) as theory processor or engine.

Each ddbt theory is linked to a database theory (cf. figure 2) with a new type of link (ddhrlink); therefore it is no longer needed to describe the PROLOG version of the database dictionary in a speciale file.

When the user saves a ddbt theory (at the creation time or after an update), a formatting is automatically done by the system. The aim is to have in the memory, at any time, the latter version of the user's program in a special form which can be interpreted by the theory processor. The same operation is realized when the user opens a theory. Reciprocally, the image is cleared out when the user quits the theory.

The initial assumption states that the PROLOG-DBMS interface was only used for querying. However, external database relations can be updated via theories of another class, for example by an inheritance theory (T3). So, the generated activants are deleted at every update of the database theory (T2). Database facts are also cleared out from PROLOG memory automatically when the user quits the environment of a ddbt theory or explicitly by the user, using a special command `clear_theoryname).

The class ddbt does not handle inheritance links. When needed, direct calls through the demo predicate are used.

The ddbt class allows the user to use logical predicates and database predicates in the same way. Obviously, the user (the final one or the knowledge base administrator) has to structure the knowledge base, i.e., to create theories and links that he needs. The database is transparent to the user only after the creation time.

Example of session:

Let's suppose that the database and ddbt theories and the ddbtlink (cf. figure 3) have been created during a previous session.
The opening of the database and the creation of the Unix pipes are transparent to the user. The database theory is not displayed unless the user opens it explicitly, for example to verify the database schema.

V. Conclusion

Principal advantages of the EPSILON KBMS are:

- modelisation of each conceptual entity with a unique concept: theory
- modularity
- conceptual hierarchie (allowed by the inheritance class)
- inheritance

One of the most interesting features of the EPSILON framework is the ability to use data stored in a relational database from a logic program. The current prototype supports three protocols to access databases from logic theories: the first one is the explicit call using the predicate demo and is available to theories belonging to any class; the second one is the inheritance mechanism and is available to the theories of the inheritance class; the third one is the hybrid kernel-DBMS interface which is available to the theories of the dbms class.

Using a half compiled, half interpreted approach for PROLOG-DBMS interface has the following advantages:

- Accesses to the databases are less frequent than in the fully interpreted strategy since edb-predicates are grouped into edb-lists translated to SQL statements. This technique is inherited from the fully compiled approach.
- Less facts are extracted from database than in the compiled method, because the calls to the database are more instanciated, thus more selective, at the interpretation time. Thanks to the history management (on activants), no fact is loaded more than once. Moreover, history management allows persistence of database facts (that is a "cache" management of these facts).
- The user's language is full-PROLOG, with functions, cut, recursion and built-in predicates.

Bibliography

"The EPSILON KBMS: Architecture and DB access optimization", presented at the Workshop on Integration of Logic Programming and Data Bases, Venice, Dec.1986

"BCBG: Bases de Connaissances Bien Gérées"
Proceedings of the Conference INFORSID 1989, Nancy

"Vers une architecture réparti des bases de connaissances",
TSL No 6, Vol 6, 1987

"Prolog-DBMS coupling: a hybrid approach, half interpreted, half compiled”,
Proceedings of the Conference 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem 1988, pp. 121-134

"Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of PROLOG”,
Linkoping University-S-581 S-83-1982

"Contribution à l'amélioration technique d'un prototype de gestion de bases de connaissances et évaluation de ses performances”
Diplôme d’ingénieur CNAM, 1989

"Status and Evolution of the EPSILON system”

[8] L.Li
"L'évaluation partielle et son application pour coupler Prolog et les SGBD”
Ph.D dissertation, Lyon 1, june 88

"Un langage pivot pour le couplage de Prolog avec les bases de données: formalisation et environnement”
Ph.D dissertation, Lyon 1, oct. 87

"Integrating Database Technology and Logic Programming Paradigm”

"Représentation de connaissances et gestion de données évolutives dans le contexte de la programmation logique”
Ph.D dissertation, INSA-Lyon, 16 january 1991

[12] Pisa University, System & Management
"Theories ans Links: an object-oriented modelization”
Working Document, EPSILON six month report, may 87

"EPSILON User Manual”, Esprit P530
Epsilon Six month report, November 88

"A PROLOG metainterpreter for partial evaluation and its application to source to source transformation and query optimization”, ECAI-84, North-Holland, 1984, pp.91-100