Chapter 5
Overview of Systems for Coupling Prolog to Relational Databases

This chapter presents an overview of some of the CPR systems and prototypes which have been developed for coupling Prolog to relational databases. We present:

a) PRO-SQL, a system for coupling Prolog to the system SQL/DS, developed at the IBM Research Center at Yorktown Heights.

b) EDUCE, a system for coupling Prolog to the database system Ingres, developed at the European Computer Industry Research Center in Munich.

c) The ESTEAM interface, developed in the framework of the Esprit Project ESTEAM, for coupling generic Prolog and database systems.

d) BERMUDA, a prototype developed at the University of Wisconsin, for coupling Prolog to the Britton-Lee Intelligent Database Machine IDM 500.

e) CGW, an architecture for coupling Prolog to a database system developed at Stanford University, and PRIMO, a prototype of an interface between ARITY-PROLOG and the database system ORACLE, developed at the University of Modena, Italy.

f) The QUINTUS interface between QUINTUS-PROLOG and the Unify Database System, a product developed by Quintus Computer Systems of Mountain View, California.

Though these systems interface different relational databases (SQL/DS, Ingres, Oracle, Unify, and IDM500), their architecture is in fact system-independent in most cases, and several of these projects state explicitly that the selection of the database system has little impact on their general architecture. The six approaches listed above inspired the previous chapter on CPR systems.

5.1 PRO-SQL

The relevant feature of PRO-SQL is the total absence of transparency. The basic assumption is that PRO-SQL programs should be written by persons who are familiar with both the Prolog and SQL languages. Thus, a special predicate SQL is used to include statements which are executed over the SQL/DS database system, as follows:

\[ SQL(< SQL - Statement >). \]
The SQL statements supported include data definition, insertion, and retrieval statements, as well as statements for transaction control.

An example of the use of the SQL predicate for a query over the EMPLOYEE relation is:

\[
\text{SQL('SELECT NAME, SAL FROM EMPLOYEE WHERE SAL > 50').}
\]

The effect of this query is to assert, in the form of Prolog facts, all tuples which satisfy the query predicate in the memory-resident Prolog database. The predicate name used for asserted facts is the name of the first relation mentioned in the \textit{FROM} clause. The execution of the SQL query takes place synchronously: the Prolog engine is suspended until completion of the loading of selected tuples into main memory.

A second example of use of the SQL predicate is the following:

\[
\text{SQL('SELECT CHILD, PARENT INTO X, Y FROM EMPLOYEE WHERE SAL > 50').}
\]

In this case, the variables \(X\) and \(Y\) are bound, after execution, to values from the first tuple extracted from the database; once the first tuple has been retrieved, the control is returned to the Prolog engine, while the loading of subsequent tuples of the result is performed asynchronously.

The execution of a recursive program takes place by iterating calls to the SQL predicate. These calls are built by using the knowledge of the adornment of query predicates (i.e., of the positions that will be bound to query constants). The following program is an example. It computes the \textit{ancestor} relation, and works correctly for evaluating ancestors of a given person, but not descendents:

\[
\begin{align*}
\text{ancestor}(X,Y) & : - \text{SQL('SELECT CHILD, PARENT INTO X, Y FROM FATHER WHERE CHILD = X')} . \\
\text{ancestor}(X,Y) & : - \text{father}(X,Z), \text{ancestor}(Z,Y).
\end{align*}
\]

In this case, the Prolog program interacts with the SQL/DS database only when executing the first \textit{ancestor} rule, while the second rule attempts unification with main-memory facts which have been loaded by the first rule. Thus, the second rule is normally executed under the control of the Prolog engine.

PRO-SQL supports transaction control. By default, each activation of the SQL predicate is considered a transaction, and is protected by recovery and concurrency control systems from the effects of failures and concurrent execution of other transactions. The programmer can also indicate that several SQL statements should be executed as a single transaction, by means of two SQL/DS statements: \textit{COMMIT WORK} and \textit{AUTOCOMMIT OFF}. The following is