Compiling a rule database program into a C/SQL application

G. Kiernan, Christophe de Maindreville

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1281

Programme 4
Bases de Données

COMPILING A RULE DATABASE PROGRAM INTO A C/SQL APPLICATION

Gérald KIERNAN
Christophe de MAINDREVILLE

Août 1990

RR-1281

Un Générateur d'Applications SGBD basé sur un Langage de Règles de Production[1]

G. Kiernan and C. de Maindreville
I.N.R.I.A. Rocquencourt

Résumé

Cet article présente un compilateur pour un langage de règles de production pour SGBD relationnels. Ce langage appellé RDL/C, offre des facilités de programmation déclarative à base de règles de production et des facilités de programmation procédurale autour du langage de programmation C. Conçu pour SGBD relationnel, les règles sont basées sur le calcul relationnel de tuples étendu au support des types abstraits de données. Ceci permet d'exécuter des programmes de règles sur le SGBD sans avoir à extraire les données de celui-ci au cours de l'inférence (sauf pour l'affichage de résultats). L'application générée par le compilateur contient les règles ainsi que le moteur d'inférence qui pilote l'exécution. Le langage offre des variables en mémoire qui peuvent intervenir dans les règles et qui sont prises en compte par le moteur d'inférence et des effets de bords procéduraux (par des appels à des programmes C depuis le corps des règles). Le langage et son implantation sont compatible avec SQL et des modules de règles peuvent être appellés depuis des programmes C.

Compiling a Rule Database Language into a C-SQL application

G. Kiernan and C. de Maindreville
I.N.R.I.A. Rocquencourt

Abstract

This paper presents the design and the implementation of a rule database language compiler. The RDL/C language supports both declarative programming based on a production rule language and C-based procedural programming. The language supports domain variables which can appear in rules. These variables are monitored by the inference engine and included in the semantics of rule firing. A partial ordering among rules is available to the user. The RDL/C compiler translates RDL/C source code into C code with embedded SQL statements. Its implementation is compared to fully integrated deductive databases and to loosely coupled systems. The contribution of the paper is to show how the rule based paradigm for database can be used as a framework for a general purposed database application generator.

---

[1]Cette recherche a été financée partiellement par le Ministère de la Recherche et de l'Enseignement Supérieur.

Compiling a Rule Database Language into a C-SQL application[1]

G. Kiernan and C. de Maindreville

I.N.R.I.A. Rocquencourt
78135 Le Chesnay
FRANCE

Abstract

This paper presents the design and the implementation of a rule database language compiler. The RDL/C language supports both declarative programming based on a production rule language and C-based procedural programming. The language supports domain variables which can appear in rules. These variables are monitored by the inference engine and included in the semantics of rule firing. A partial ordering among rules is available to the user. The RDL/C compiler translates RDL/C source code into C code with embedded SQL statements. Its implementation is compared to fully integrated deductive databases and to loosely coupled systems. The contribution of the paper is to show how the rule based paradigm for database can be used as a framework for a general purposed database application generator.

---

[1]This research has been supported by the Ministère de la Recherche et de l'Enseignement Supérieur.

1. Introduction

Different approaches have been proposed to combine AI and DB technologies. The first approach provides the rule-based system with access to data managed by the DBMS (eg., KEE connection, [Abarbanel86]). This approach is often called loose-coupling. Two related problems appear in this approach: First, expert systems and database systems are based on different data models (eg., frames vs relational model). The data model mapping is left to the application programmer. Second, loosely-coupled systems have severe performance limitations. The data has to be down-loaded into the expert system shell's working memory before it can be processed by the rules. The database system is simply used as a storage system with its interface limited to "store" and "load" commands. In particular, a large part of the pattern matching task is left to the expert system shell's inference engine and thus, the DBMS operators are under-utilized.

An improvement over loose-coupling is the tight-coupling approach, (eg.,the BERMUDA system [Ioannidis88]). The rule language is typically PROLOG and efficiency is reached through caching and pre-analysis of rule programs. This pre-analysis identifies relational operations that are submitted to the DBMS rather than to the rule inference engine.

The second approach consists in the development of a persistent rule language. Apersistent rule system is not necessarily a database system but offers a dedicated tool to develop *intelligent data-intensive* applications. The LDL language [Naqvi89] typifies this approach; it is a Datalog-like language extended with constructs such as negation, updates, control structures, and type constructors.

The third approach extends an existing database system. A rule language interface is offered to the database system users. A new rule language has to be designed, and an ad-hoc rule interpreter (or compiler) has to be integrated within the DBMS. POSTGRES, [Stonebraker88], RDL1 [Kiernan89] and Starburst [Widom89] follow such an approach. They propose production rule languages as extensions of an SQL environment. The POSTGRES and RDL1 projects have developed working prototypes [Stonebraker89, Kiernan90]. It has been shown that this kind of approach is well suited for a restricted class of applications such as active database applications [Dayal88, Stonebraker89]. Furthermore, integrating the inference engine within the DBMS is a way to achieve good performance [Haas89, Kiernan90]. However, two main limitations remain. First, the

rule interface is DBMS dependent. Second, the rule language interface does not provide programming features usually found in expert system shells such as control structures, main memory variables, connections with a procedural language, and user interaction.

A fourth approach, investigated in this paper, provides a rule interface on top of any relational DBMS. More precisely, the approach provides :

(1) a rule language suitable for data-intensive applications. This language is shown to be more suitable than C-SQL for traditional applications and more adequate than PROLOG for *intelligent data-intensive* applications. RDL/C includes a rule section to define in a declarative way, general knowledge about the data stored in the database. This language includes procedural constructs such as explicit control over the rules and calls to a language such as C. Rules and procedures exchange parameters in an easy way. However, the initial semantics of the rule language remains declarative and easily comprehensible to the user.

(2) programs which run on top of any relational DBMS. The interface between the rule language and the DBMS is SQL. The compiler produces C/SQL code which runs over the DBMS.

(3) efficiency, and standard database system features. The first requirement is achieved by the fact that all data are manipulated by the DBMS, and DBMSs in the near future will be able to process global optimization of C/SQL statements. The second feature is achieved by the high-level interface between the rule language and the DBMS, i.e, SQL.

The general architecture of the RDL/C compiler is portrayed in Figure 1.

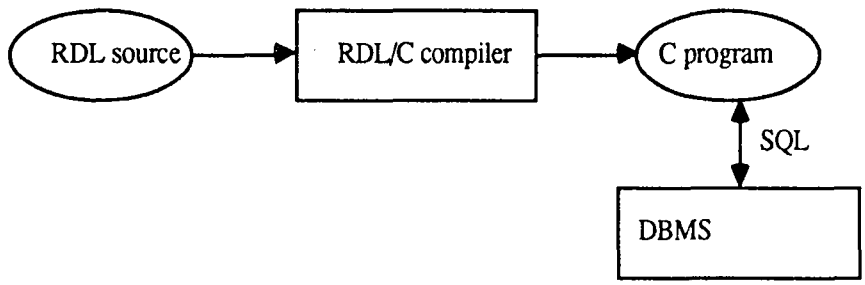

Figure 1. Sketch of the RDL/C compilation and run-time environment.

The compiler accepts a source program and produces as output, a C program which implements the rule program. The DBMS does not require any inferencing capabilities to process the program. The C program contains code to implement each rule and includes the inference engine which fires rules until a fixpoint is reached. All data

remains in the DBMS during the inference process. This is because rules are based on relational calculus and can thus be solved by the DBMS.

The remainder of this paper is organized as follows. Section 2 presents illustrative examples of the RDL/C language. Then, the is informally presented. We describe the procedural constructs of the language and a control sub-language over the rules. The last part of section 2 details the Compilers' target code ,which is a C program generation. Section 3 outlines the compiler architecture and the run-time environment. It also describes the technical choices made during the design and the implementation of the system. Section 4 gives preliminary system performance results. The last section concludes the paper.

2. The Rule Language Compiler

RDL/C is derived from RDL1 [Maindreville88, Kiernan89], a production rule language which has been integrated in a relational DBMS [Kiernan90]. The RDL/C language supports both declarative programming based on production rules and procedural programming based on C code. In this section, we first illustrate the power of the language through examples. The extensions brought to the kernel language to accomodate procedural aspects are described in the next section. We refer to these extensions of the kernel language as RDL/C (This is essentially the production rule environment upgraded to include C programming constructs). The notion of rule module is introduced as the compilation unit of the language. Another extension, the control sub-language, is also described. The last part of this section describes the target code produced by compiler.

2.1 The Source Language

2.1.1 Illustrative examples

Before giving the syntax and semantics of RDL/C, we give simple examples of rule programs. The purpose of these examples is to illustrate the main features of the language : declaration of an intensional database through production rules, integration of C statements which manipulate main memory variables, and interaction with the user.

The following rule program asks the user to enter a value for the C-variable age and then, fires the rule r1 using the current value of age. This illustrates the use of main memory variables in rules and interaction with the user.

```
MODULE M1 ;
BASE
        Person (name text, age integer);
DEDUCED
        OldPerson like Person;
VAR
        integer age;
INIT
        {scanf("%d", age);}
RULES
r1 :
   IF Person(x) and x.age >= age
        THEN + OldPerson(x) ;
END MODULE
◊
```

Let us consider the slightly modified module :

```
MODULE M2 ;
BASE
        Person (name text, age integer);
DEDUCED
        OldPerson like Person;
VAR
        integer old;
INIT
        {scanf("%d", age);}
RULES
r1 :
   IF Person(x) and x.age >= age
        THEN + OldPerson(x) ;
{exit();} else {scanf("%d", age);}
END MODULE
```

This module asks the user an initial value for the C variable age. If the firing of r1 is successful, (ie., the relation OldPerson has been modified) the program terminates with the exit procedure. If the firing does not modify the database, the user is asked for another value of age. If the value of age is modified, (this is checked by the inference engine), the rule r1 is fired again. ◊

2.1.2 The RDL/C language

The kernel language has rich programming constructs: C code sections, C variables in the rules and procedural control over the rules. In the sequel, we describe the language and motivate the use of these programming constructs.

The *rule module* is the compilation unit of the language. Its input and output interfaces are a set of base or deduced relations. The notion of module is very similar to the notion of rule set in rule-based systems. With rule sets, modular knowledge bases can be designed and the possible connection between different rule programs can be defined. Input relations are declared using the INPUT <RelationName>, ..., statement. Output relations are declared with a similar statement. The output relations are the result of a rule program execution and can be used by another rule program. Base and deduced relations are declared in the same way.

After the relation declaration sections, C variables can be declared in the optional VAR section. These variables are local to the module. Their use is shown in the following section of this paper. *Initialization* statements can also be used. They are written in C and are executed at the beginning of program execution. This section is followed by the *rule section.*

The rule section is a sequence of rules. Each rule is preceded by a "<RuleName>: " statement. A rule in RDL/C is preceded by an optional C statement. A C statement can be a compound C statement and hence include more that one statement. This statement is executed just before firing the rule. Then, the rule is declared and is optionally followed by one or two C statements. The first statement is executed if firing the rule has modified the database. The second statement is executed if firing the rule has not been productive.

An optional control section can be declared. It consists of a string that describes an explicit ordering among rules. This control language is described in the following section. After the control section, *wrap-up* statements can be given. These statements are written in C and are executed at the end of program execution. The general syntax of an RDL/C module is the following one :

```
MODULE <Module_name> ;
[INPUT <RelationName>,...;]
```

```
[BASE <RelationName>,...;]
[DEDUCED <RelationName>,...;]
[OUTPUT <RelationName>,...;]
[VAR ...;]
[INIT
        {C code} ]
RULES
<RuleName1> :
        {C code}
        IF <Conditions> THEN <Actions> ;
        {C code} else {C code}
...
<RuleNameN> :
        {C code}
        IF <Conditions> THEN <Actions> ;
        {C code}else {C code}
[CONTROL ...]
[WRAPUP
        {C code} ]
END MODULE
```

2.1.3 Interfacing the C Language and the Rule Language

As described in the previous section, an RDL/C program includes two different languages: A production rule language where conditions and actions range over database relations and the C language which manipulates main memory variables. In this section, we describe the communication between these two languages. We also detail the effect of embedded procedural statements on the semantics of the language.

C programs to RDL/C programs:

The RDL/C compiler produces either C procedures or C programs. C procedures can be linked to any C program. C programs generated by the compiler are immediately executable over the database. For C procedures generated by the compiler, the

parameters are given by the INPUT section declared in the module. The procedure result is given by the OUPTUT section also declared within the module.

RDL/C programs to C programs:

An RDL/C program can include C statements. C variables declared in the VAR section are local to the module. C variables ranging over database domains can appear in a rule and are materialized as constants before firing a rule. The RDL/C inference engine is aware of these main memory variables and includes them in the semantics of rule firing. Updates to variables referenced in a rule make the rule a candidate for firing. Variables are often used to pass parameters between rules or to communicate with the user environment. Variables used in the rule section can be simulated by a relation with a single attribute. This relation would always contain one tuple since variables are mono-valued.

2.1.4 The Control sub-language

This section describes a sub-language for meta control. Knowledge bases need be structured to be usable. Furthermore, the mixing of declarative reasoning and imperative control has been shown necessary for many applications. The RDL/C language allows two kinds of procedural aspects. The first one is the possibility of exchanging parameters between rule programs and C programs. The second aspect is the possibility of specifying an explicit control over the rules. Standard control structures such as sequences and iterations can be specified with the formalism we have chosen. This formalism clarifies the control which would be otherwise spread into the rules. This control might be simulated in the kernel language [Abiteboul89].

The control sub-language specifies an application mode and induces a partial ordering over the rules. Each expression of the sub-language is declared in the module. The basic terms of this language are rule names. The expression r1 means that rule r1 has to be fired once. A general expression exp in this language is :

. [exp] means that exp has to be fired until a fixpoint is reached.

. exp1; exp2 means that exp1 has to be fired before exp2.

It can be demonstrated that this language expresses the iteration (while <Relational condition> do r1,...rn), and conditional (if <Relational expression> then r1,..., rn), over a rule set [Georgeff82].

If a rule does not belong to the control language, its firing is chosen at random by the inference engine. If there is no CONTROL section, the rule interpreter applies a default strategy. If r1,r2,.., rn are the rules declared in a module, the default strategy is given by : [[r1] [r2]...[rn]]. For example:

. r1 ; [r2 r3 r4] ; r5 is a possible expression. It enforces a computation of the form : $(r1)^a ((r2)*(r3)*(r4)*)^\sigma (r5)^a$ using standard notation for context-free grammars. The notation $()^\sigma$ stands for "fire up to saturation" and a is equal to 1 or 0.

. [r1; r2; r3] enforces a computation of the form : $((r1)^a(r2)^a(r3)^a)^\sigma$ ◊

2.1.5 Example of an RDL/C Program

We now present a toy but illustrative example of the language. Let us consider the permanent relation CONNECTIONS having for schema (City1 text, City2 text, length integer) and the relation MAP (City text, X0 integer, Y0 integer) which stores the position of cities on a Cartesian map. For simplicity, we consider the relation Connections to be acyclic. The goal of the following program is to find a road between city A and city B with length less than a certain value.

```
MODULE TRIP ;
BASE
                CONNECTIONS (City1 text, City2 text,Length int);
                MAP (City text, X0 integer, Y0 integer) ;
DEDUCED
        ROADS (City1 text, City2 text, Length int);
OUTPUT
                RESULT (City1 text, City2 text, a1 int, a2 int, b1 int,
b2 int);
VAR
                MaxLength int ; Departure, Arrival text; found int ;
INIT
                {scanf("%d", MaxLength) ; scanf("%s", Arrival) ; found
= 0 ;}

RULES
r1 :
        {scanf("%s", Departure);}
        IF Connections(x) and x.length < MaxLength and x.City1 =
Departure
        THEN + Roads(x) ;
        {} else {printf("No possible road\n"); exit();};

r2 :
```

```
        IF Roads(x) and Connections(y) and x.City2 = y.City1 and
(x.length +                    y.length) < Maxlength and not found
        THEN - Roads(x) + Roads(x.City1, y.City2, x.length + y.length)
;

r3 :
        IF Roads(x) and Map (y) and Map (z) and x.City2 = Arrival and
x.City1 =                      y.City and x.City2 = z.City
        THEN + Result(x.City1, x.City2, y.X0, y.Y0, z.X0, z.Y0) ;
        {found= 1;}
CONTROL
r1 ; [r2 ; r3]
WRAPUP
{displayroads (Result, road);}
END MODULE
```

The entry parameter of this rule module are the base relations CONNECTIONS and MAP. The module output is the RESULT relation. *MaxLength* and *Arrival* are initialized at run time by the user. *Found* is a C variable which is checked in rule r2 and updated in rule r3. The INIT section declares C statements which are computed at the beginning of module execution. Rule r1 asks the user to enter a value for the *Departure* variable and then selects into the ROADS relation, all the roads starting from *Departure* having a length less than *MaxLength* . If no tuples satisfy these conditions, a message is displayed and the program stops. Rule r2 iterates over the ROADS and CONNECTIONS relations to select the roads starting from *Departure*. The intermediate roads are deleted from the final result. The control string specified that rule r3 is fired between two firings of r2. Rule r3 selects into RESULT the road between *Departure* and *Arrival*. If there is a road, the C variable *found* is updated to the value 1, and the computation stops, (see the LHS of r2). Then, the contents of RESULT are displayed using the displayroads procedure.

2.2 The Target Code.

The following example is a summary of the C program produced to implement module M1 given as the first example in section 2.1.1. The program has been simplified for clarity.

```
Initial includes and other declarations
...
static int age1 =0;
static int old_age1 ;
static int change_age1 ;
static relation *person ;
```

```
static relation *oldperson ;

static r1(projClause) int projClause;
{
   sprintf(fromWhere," FROM %s as x    WHERE x.%s   >=     %d   ; " ,
person->name  , findAttName(person, 2), age1);
   switch (projClause) {
   case 1 :
         sprintf(question,"SELECT x.%s, x.%s ", findAttName(person, 1),
findAttName(person, 2));
         break;
   }
   strcat(question, fromWhere);
   rep = sql_exec (question);
}
static action_r1(){
}
static inits()
{
   {scanf("%d",age);
}


main(argc, argv) int argc;
char *argv[];
{

   inits();
   change_age1 = 1;
   old_age1 = age1;
   rep = sql_connection();
   if (rep != 0) printf("Connection error\n");
   else {
         person = searchrel("person", &rels) ;
         oldperson = searchrel("oldperson", &rels) ;
         initListOfRules();
         initActionLinks();
         initListOfPertinentRules();
         initControl();
         rep = validateSchemas() ;

         while (there are pertinent rules left) {
               rule = selectRule();

               fire the rule
               }
         free any temporary relations no longer needed

         Check to see if the rule has an effect on the output place
         if so, fire the procedureal part of the rule.
```

```
            Monitor any changes to the variables

            }
            wrapup();
            rep = sql_deconnection();
}
◊
```

The C code produced by the RDL/C compiler contains the following elements: a skeleton version of each rule; main memory rule program variables; a report generator procedure per report; a set of initialization procedures for meta-control, links between rules and relations, links between rules and main memory variables; run-time validation of base relation schemas; run-time validation of virtual relation schemas; and the inference engine which calculates the list of pertinent rules, selects a rule from this list, and fires the rule.

2.2.1 Compiling a Rule

A rule is compiled into a procedure which calculates a set of tuples for each relation involved in the RHS of the rule. In the above example, procedure r1 implements this function for rule r1. The name of the procedure is the name of the rule. First, the From and Where SQL clauses are composed. The string which will represent this part of the SQL statement is a skeleton because relation names and attribute names evolve over the course of the program and have to be completed before each call. Furthermore, variable expressions are replaced with their values at this time. This is the case for variable age1 in the above example. All ADT constants are automatically type casted by the compiler to provide the DBMS with precise type information. The From and Where clauses remain constant for one rule firing. Then, the projection clause of the SQL query is assembled. The projection clause varies with each relation involved in the RHS. It is also assembled from a skeleton string. There is one such string per relation in the RHS.

2.2.2 Managing variables in modules

All domain variables (for example, age1) are mapped into simple C variables. Integers or ADT specializations of integers are mapped into C ints. Real numbers or ADT specializations of these are mapped into C doubles. Texts or ADT types derived from texts are mapped into string buffers. There are three C variables generated per RDL/C variable. The first C variable has the same name as the RDL/C variable and is of

corresponding type. It is the implementation of the user variable. The second C variable has the same name as the RDL/C variable but prefixed with old_ and it is also of corresponding type. The third variable has the same name as the RDL/C variable but prefixed with change_. It is of type int and used as a boolean indicator. The old_ variables store the last values of each variable. The change_ variable monitors changes to variables using a simple boolean expression. The change_ variables are referenced by the data structures of a rule which includes the corresponding variables. The inference engine can thus monitor these variables to determine if a rule is to be added to the list of possible pertinent rules.

2.2.3 The inference engine

The inference engine is compiled into the C main procedure or into a procedure whose name is the name of the module. This procedure ensures the connection with the DBMS, calls the initialization procedures and cycles until a stable state has been reached. For each cycle, it determines if any domain variables have been updated, calls the C procedure which implements a rule, and performs the relational union or difference operation between temporary results and the contents of relations declared in the rule program. The main loop also checks if firing the rule has had any effect on the database state. If so, a decision is taken to run the C code attached to the rule. Eventually, it executes the reports, the wrapup code supplied by the user in the RDL/C source and calls the SQL disconnect request to terminate the transaction.

The links between rules, relations and main memory variables are the data structures used by the inference engine to solve a rule program. For each rule, the relations involved in the condition part of a rule are mapped into a pre-place structure and the relations involved in the action part of a rule are mapped into a post-place structure. These structures form the internal representation of a rule program. This representation is a graph structure called PCN or Production Compilation Network [Maindreville88]. This graph is the central component of the inference engine. The change_ or monitor variables are also linked into this data structure. One such link exists for each variable appearing in a rule.

The validateSchema procedure has no parameters and is generated by the compiler. It is called at run-time after the base relation schemas have been documented. It reports any changes to the schemas which might require recompiling the RDL/C program. Every base relation referenced in the RDL/C program is validated. For each relation,

attributes are compared for equality of names, types and number. Any mismatching causes a message to be issued and program execution to be halted.

Deduced relation schemas are determined at run-time by the types of expressions in the SQL projection clause. The schemas of temporary relations generated by SQL statements have to be coherent with the schemas declared in the RDL/C source program. Types for expressions which are attributes are taken from the database meta-relations. Types for ADT operator expressions are also obtained from the database meta relations. These operators are selected at run-time and can vary according to different database states. Constant expressions have implicit types. These are casted at compile time to ensure typing. The validateDeducedSchema procedure is called to compare a temporary result with the schema of a deduced or output relation.

2.2.4 The Run-Time Library

The library linked with the C program produced by the RDL/C compiler contains routines for initializing links among rules, initializing and updating the list of pertinent rules, and selecting a rule from the list of pertinent rules.

Action links are initialized to facilitate the calculation of pertinent rules at each cycle. For each rule, the action links are those rules which might be affected by the successful firing of a rule. A rule is affected by the firing of another rule if one of the relations involved in the condition part of the rule appears in the action part of another rule or if a relation that appears positively in the action part of a rule appears negatively in the action part of another rule or inversely.

Two lists of rules are managed by the inference engine; one is the list of rules and the other is the list of pertinent rules. The list of pertinent rules is calculated from the list of rules and from the action links of a successfully fired rule. Each rule in the action link of the fired rule is checked for pertinency. The list of rules is then traversed to check if there have been any updates to the variables involved in a rule. If so, the rule is added to the list of pertinent rules.

The rule selection procedure uses a stack to manage the depth of nesting of the current control structure being processes. For example, the control structure [r1 r2 [r3 r4]] will require a stack depth of level two because of the nested block structure. Each time a block or sequence structure is processes, it is pushed on top of the stack to become the

current structure. When no more rule in a structure is pertinent, the stack is popped and the current control drops one level down.

2.3 Discussion

The RDL/C language presents some significant advantages compared to the C/SQL language. The first one is the expressive power of a rule compared to an SQL query. A rule might include several actions in its action part and is then equivalent to several SQL queries. A rule can be recursive and is equivalent to a C while loop over several SQL queries. Therefore, an RDL/C program is more concise than the equivalent C/SQL program. The second advantage is that the user is free from the management of temporary relations. In RDL/C, these tasks are managed by the inference engine. The resulting program is easier to develop and to maintain. Control in an RDL/C program might be written in two ways: In the language itself or with the control sub-language. This simulates standard control structures such as IF THEN ELSE and DO WHILE. Furthermore, RDL/C modules can be combined and called by C programs.

Compared to PROLOG and to languages offered in expert system shells, the RDL/C language presents the following particularities. The data model is relational, (extended with Abstract Data Types). This implies that all rule programs can be solved without having to down-load data into some working memory. All calculations on data are performed by the DBMS. The application drives the inference through calls to the DBMS. The only information retrieved from the DBMS is cardinality of results. Relations are only read when results need to be displayed. Furthermore, there is no mismatch between the data model used in the rules and the DBMS data model. PROLOG execution is a tuple at a time while RDL/C is based on the set oriented semantics of relational systems. The ordering of rules and predicates in PROLOG is important while it is not in RDL/C. Finally, the rule language is coupled with the SQL query language, which eases application development and portability. However, this type of inference might not be adequate for particular expert system shell applications such as diagnostic applications which rely on backward chaining.

The most important feature of a compiler is making sure that is generates correct code. RDL/C is derived from RDL1. The RDL1 execution model is based on relational calculus. Since RDL1 is integrated within a relational DBMS, some of the relational operators available in RDL1 are not available in SQL. These are the Difference, Intersection, and Merge relational operators. RDL/C semantics differ from RDL1 semantics in two respects. These are deletions and atomicity of multiple actions in the

Action part. In RDL1, deletions are done with the Difference operator. In RDL/C, deletions are implemented with a FOREACH SQL clause. These two operations give different results in the presence of NULL values. Multiple actions are considered atomic in RDL1. Hence, the action -Q(a) followed by the action +Q(a) has no effect irrespective of the current database state. However, these semantics are not supported in RDL/C. Actions in RDL/C are viewed sequentially. If Q does not contain a, the same action will insert a in Q.

3. The System Architecture

3.1 A Single Pass Rule Compiler.

The RDL/C compiler portrayed in Figure 2 is a single pass compiler which accepts RDL/C source and produces C programs which run as C-SQL application programs.

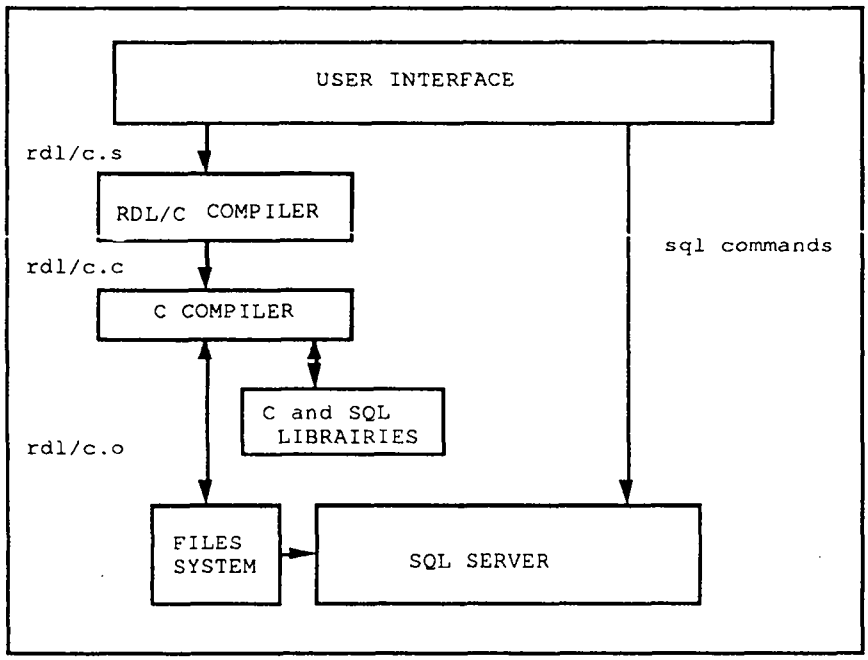

Figure 2 System Architecture.

The RDL/C compiler calls the C compiler to process the RDL/C output (along with run-time libraries) to produce an executable version of the rule program. Type checking is done at compile-time and at run-time. The correctness of expressions is ensured at compile-time. The types of terms appearing in expressions are domains. In our implementation, these domains include user-defined abstract data types [Kiernan89b].

3.3 The Programming Environment.

The user interface includes two sets of primitives: The SQL language, and a set of commands to edit, run compile and debug rule programs. There are two key compilation options. The first one generates a C/SQL procedure. The input interface to these procedures is given by the INPUT section declared in the module. The output interface is given by the OUTPUT section declared in the module. The second option generates C/SQL programs. The RDL/C compiler can be invoked with a debug option. The -d option allows using the symbolic debugger dbx to trace the execution of rule programs. The name of the RDL/C source file is stated on the RDL/C command line along with other object code files and libraries used in the compilation. The -p option includes performance measurement procedures in RDL/C applications. Performance results can be obtained after each application run. Programs produced by the RDL/C compiler can be run with the -t option to obtain run-time information such as the list of pertinent rules for a cycle; the rule chosen by the inference engine for firing; the number of tuples produced by a rule's firing; the SQL query produced by the inference engine, and error messages. RDL/C source and object code are simply managed by the UNIX standard editors and file system.

3.4 Implementation Issues

Some of the difficulties encountered in this implementation were with respect to SQL attribute naming conventions and deduced relation schemas. In the implementation of SQL we are using, names are generated for constant, arithmetic and ADT operator expressions. Furthermore, the attribute names resulting from UNION operations are not the names of the relations involved in the union but specially generated system names like COL1 and COL2. So attribute names cannot be hard coded into SQL statements. Schema information has to be obtained after each query to resolve attribute names in SQL statements. Moreover, names for temporary relations have to be generated for deduced relations when more then one rule concludes on the same relation. Attribute assignments in the RHS of rules have to respect precise schema matches because schemas cannot be given in the SQL projection clause. Rather, they are deduced from the elements found in the projection clause. However, if no ADT operators are used in the projection clause, type checking at compile-time maintains the correctness of deduced relation schemas for run-time execution. Another difficulty

is the lack of the DIFFERENCE operator in SQL. Deletion actions have to be implemented using the universal quantifier.

4. Performance

We study performance from two viewpoints: that of the compiler itself and that of the application program (including programmer productivity). The compiler performance is divided into two categories: the RDL/C translator from rule source into C code and the C compiler. Figure 3 shows the compiler performances with respect to the size of the program to be compiled. The size is given in number of lines. The graphs shows: *the total compilation time,* the time it takes to process the rule source and produce an executable program; *the RDL/C compilation time,* the time it takes to process rule source to produce C code; and *the C compilation time,* the time it takes the C compiler to process the C code to produce the executable program.

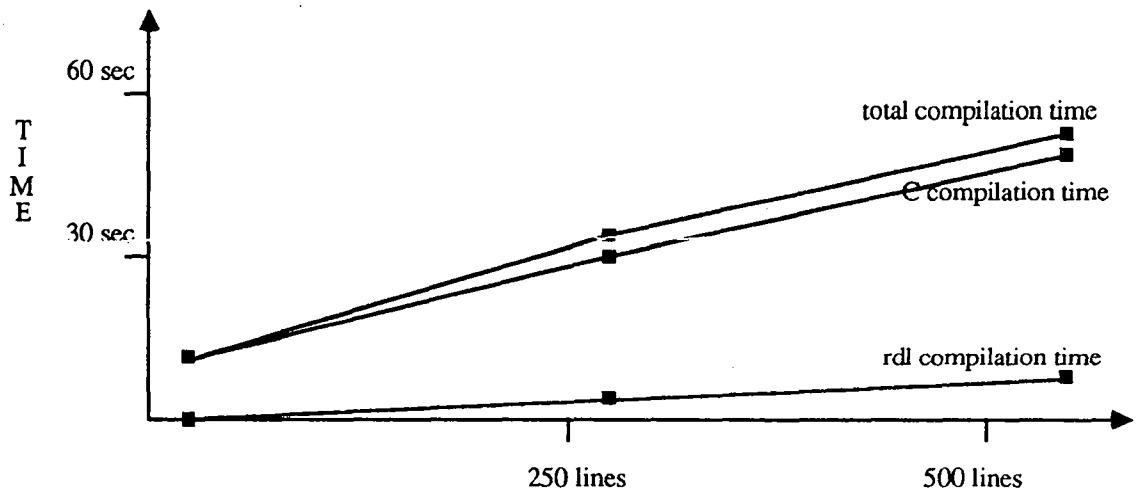

Figure 3: Compiler Performances.

Preliminary performance results show clearly that the C compilation phase is heavier than that of the RDL/C compilation phase. These results might be due to the fact that there is a 1 to 10 ratio between the number of source lines and the number of C code lines generated by the RDL/C phase. Better performances might be achieved by using a faster C compiler or by reducing the amount of C code produced by the RDL/C phase. However, the size of the run-time libraries which must be linked to the C program

remain constant. The loader is called by the C compiler and is included in the C compiler's performances in this analysis.

The overall good performances of the compiler are based on the fact that it can compile programs independently of the DBMS because relation schemas are given in the source program. The base relation schemas could be obtained from the DBMS but we felt that having them included in the source code adds to program readability.

Verifying the correctness of schemas at compile time is useless because schemas may evolve from one run to the other. Schemas are validated at each program run. This enables the detection of missing base relations or changes to schemas which may require program recompilation.

Figure 3 shows the performances of an RDL/C application with respect to the number of rules in a program. The performances of the application program are sensitive to the following points:

- *the DBMS connection and disconnection time;* the DBMS connection command issued at application start-up forks the application process to execute the corresponding DBMS process. The DBMS is a large process so time is necessary for it to be loaded into memory and for execution to begin. The connection and disconnection times are important but remain constant per application.
- *issuing queries to solve a rule;* each rule that is fired results in several SQL statements being delivered to the DBMS. These statements select tuples which qualify the condition part of the rule, perform the union or difference between these tuples and those in the deduced relation. Temporary results are named issuing an SQL temporary relation naming command. Temporary results no longer needed are released by the SQL FREE command.
- *schema interrogation time and other activities;* before the inference process starts, the schemas for base relations are queried. Each query is a request issued to the DBMS. During the inference process, each SQL query is followed by a schema interrogation to obtain attribute names and to verify the correctness of deduced schemas. Other activities comprise the initialization of the data structures used by the inference engine and the inference process itself. Preliminary performance measurements show this time to be insignificant.

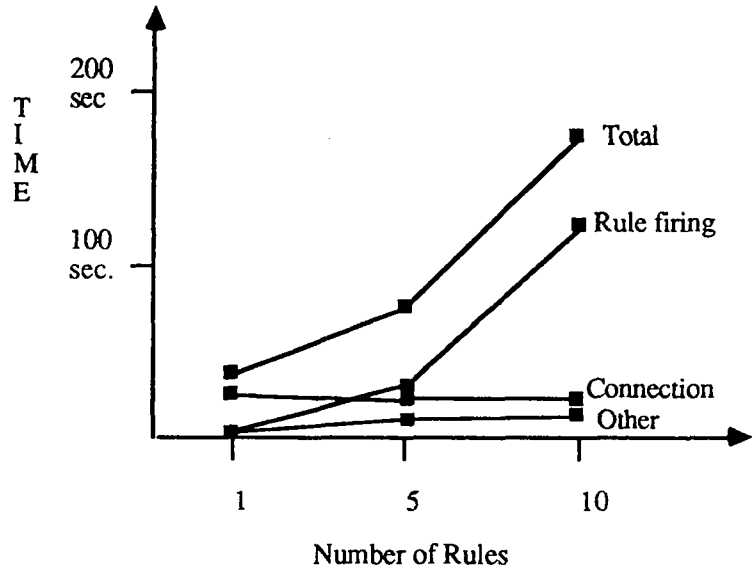

Figure 3 Performance measurements of RDL/C applications.

Figure 4 shows that the performances of RDL/C programs are identical to those of equivalent C-SQL application programs. The C programs take the

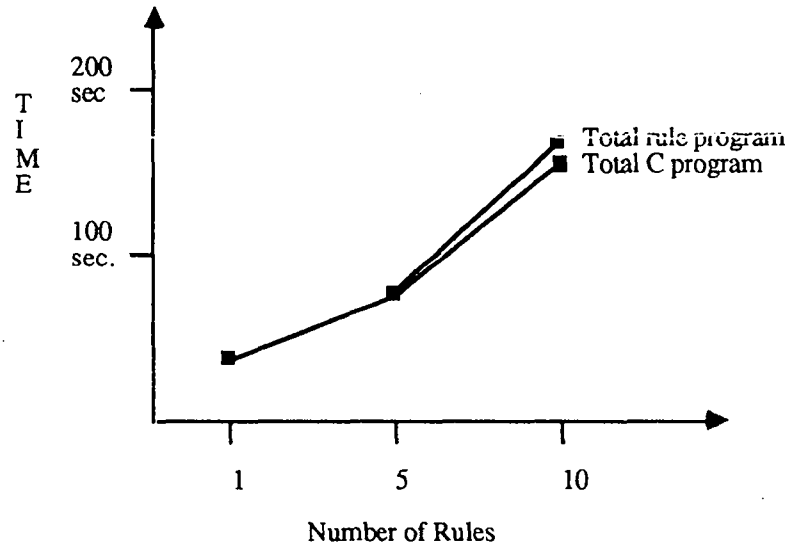

Figure 4 Performance comparisons between RDL/C programs and equivalent C-SQL programs.

same time as the RDL/C programs to establish connection with the DBMS. Furthermore, the C-SQL programmer uses the same SQL queries to run the application as those produced by the compiler from RDL/C rules. However, the RDL/C inference engine is more general than the control structure in the application program, so

initialization time is saved. Moreover, base and deduced schema validation was not implemented in the application programs so schema validation and interrogation time is saved.

The difficulty in implementing a rule based application in C-SQL is learning C and learning how a particular SQL implementation is embedded into C. It takes about three to four times as many lines of code to implement a rule based program in C-SQL as in RDL/C. Moreover, control strategies must be implemented in large application programs because the difficulties of rule selection cannot be hard coded by the programmer with short-cuts. Added difficulties are due to the fact that the application programmer has to manage temporary relations, and SQL attribute naming conventions.

These results show that rule based application programs written in RDL/C perform as well as equivalent C-SQL application programs and are easier to implement, test, debug and maintain.

6. Conclusion

This paper presented a rule database language compiler for SQL database servers. The RDL/C compiler accepts a production rule language extended with procedural constructs and produces C/SQL programs or procedures. The RDL/C language mixes declarative with procedural programming. This allows (i) a better organization of rule programs, (ii) the use of several programming constructs usually found in programming languages and expert system shells, and (iii) flexible I/O facilities for displaying results and obtaining user input. Applications requiring reasoning need such features. Standard DBMS applications can also be easily written in the RDL/C language. A first sample of a geographic application shows that the resulting code is shorter and easier to maintain than equivalent C-SQL programs and that performances are similar in both cases.

An implementation which, at first glance, appears quite similar to ours, can be found in D/KBMS [Ramnarayan88]. The similarity with our implementation is in the fact that C programs are generated to run rule programs. To solve an SQL query on a deduced relation, D/KBMS extracts information from the DBMS and generates a C program using this information. It is then compiled and run. The result relation that is generated is returned to the user as the query result. Performance analysis found in

the paper show results for (i) the time it takes for a program to be generated, compiled and run and (ii) the time it takes to run the program to obtain the query result.

The aim of RDL/C is quite different than that of D/KBMS. RDL/C is application-oriented rather than query-oriented. An RDL/C program results in a rule-based application program which can return more that one relation as program result. Actually, at the end of an RDL/C program execution, all deduced relations can be used by other modules or displayed using the language's report facility. The compiler is used to generate applications as are other standard language compilers.

Compared to the RDL1 integrated approach [Kiernan90], RDL/C must deal with the interaction between relational data and main memory C variables and has less flexibility because the programming and run-time environment of RDL1 is based on an interpreter and not a compiler. Furthermore, the integrated interpreter accesses database operators through internal procedure calls and not through SQL. This allows for a global optimization of relational algebra programs [Kiernan89]. In RDL/C, C variables can pass information to relational data in the form of constants which are materialized before the query is submitted to the DBMS. However, relational data cannot be passed to C variables as it is done with ADT operators in the RDL1 integrated approach. Moreover, the RDL1 programming environment shell provides easier access to run-time information in order to tune and debug rule programs. The main advantage of RDL/C over RDL1 is the portability of the compiler over different relational DBMSs through SQL. Furthermore, rules don't need to be stored and retrieved in some rule base. Everything is compiled into the application and ready to go. This includes all C procedures used in the rules. This also allows easy integration of rule based programs with C applications.

A running prototype of the RDL/C compiler implements all the features described in this paper. Preliminary performance measurements are encouraging. They are achieved by the fact that all the data manipulation is done by the DBMS.

References

[Abarbanel86] R. Abarbanel, M. Williams : " A Relational Representation for Knowledge Bases", *Technical Report, Intellicorp, Mountain View*, April 1986.

[Abiteboul89] S. Abiteboul, V. Vianu : " Fixpoint Extensions of First Order Logic and Datalog-like Languages", *Proc. of IEEE Int. Conf. on Logic in Computer Science*, 1989.

[Chimenti89] D. Chimenti, R. Gamboa, R. Krishnamurthy : "Towards an Open Architecture for LDL", *Proc. of Int. Conf. on VLDB*, Amsterdam, Aug. 1989.

[Dayal88] U. Dayal, et al. : " The HiPAC Project : Combining Active Databases and Timing Constraints", *ACM SIGMOD RECORD Vol. 17, N°1*, March 1988.

[Delcambre88] L.M.. Delcambre, J.N Etheredge : "The Relational Production Language : a Production Language for Relational Database". *Proc. of Int. Conf. on Expert Database System*, April 1988.

[Georgeff82] M.P. Georgeff: "*Procedural control in Production Systems*", Artificial Intelligence, (18) : 175-201, 1982.

[Haas89] L.M. Haas, J.C. Freytag, G.M. Lohman, H. Pirahesh : "Extensible Query Processing in Starburst" , *Proc. of ACM SIGMOD Int. Conf.*, Portland, June 1989.

[Kiernan89a] G. Kiernan, C. de Maindreville, E. Simon : "The Design and Implementation of an Extendible Deductive Database System", *ACM SIGMOD RECORD, Vol. 18, N° 3*, Sept. 1989.

[Kiernan89b] G. Kiernan et al.: "Managing Complex Objects in an Extendible Relational DBMS", *Proc. of Int. Conf. on VLDB*, Amsterdam, Aug. 1989.

[Kiernan90] G. Kiernan, C. de Maindreville, E. Simon : "Making Deductive Database a Practical Technology: a step forward", *to appear in Proc. of SIGMOD 90*, Atlantic City NJ., June. 1990.

[Maindreville88] C. de Maindreville, E. Simon : "A Production Rule Based Approach to Deductive Databases", *Proc. of Int. Conf. on Data Engineering*, Los Angeles, Feb. 1988.

[Naqvi89] S. Naqvi S. Tsur : "A language for Data and Knowledge Bases", book, *W.H. Freeman*, 1989.

[Ramnarayan88] R. Ramnarayan, H. Lu : "A Data/Knowledge Base Management Testbed and Experimental Results on Data/Knowledge Base Query and Update Processing", *Proc. of ACM SIGMOD Int. Conf.*, Chicago, 1988.

[Stonebraker88b] M. Stonebraker et al. : "The POSTGRES Rules System", *IEEE Transactions on Software Engineering*, July 1988.

[Stonebraker89] M. Stonebraker, M. Hearst, S. Potamianos : "A Commentary on the POSTGRES Rules System", *ACM SIGMOD RECORD, Vol. 18, N° 3*, Sept. 1989.

[Widom89] J. Widom, S. Finkelstein : "A Syntax and Semantics for Set Oriented Production Rules in Relational Databases, " *IBM Research Report, RJ 6880*, Almaden Research Center, June 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399