Making Deductive Database a Practical Technology: a step forward

G Kiernan, C de Maindreville, E Simon

INRIA, Rocquencourt
78153 Le Chesnay
France

Abstract Deductive databases provide a formal framework to study rule-based query languages that are extensions of first-order logic However, deductive database languages and their current implementations do not seem appropriate for improving the development of real applications or even sample of them Our goal is to make deductive database technology practical The design and implementation of the RDL1 system, presented in this paper, constitute a step toward this goal Our approach is based on the integration of a production rule language within a relational database system, the development of a rule-based programming environment and the support of system extensibility using Abstract Data Types We discuss important practical experience gained during the implementation of the system Also, comparisons with related work such as LDL, STARBURST and POSTGRES are given

1. Introduction

Rule-based programming has been successful in at least two areas expert systems and active databases Rule-based systems like KEE [Kee85] or OPS5 [Brownston85] are shells to develop large expert system applications An important problem limiting their use in industrial applications is that they are disconnected from traditional application support, in particular database systems Coupling an expert system with a database system (e g , KEE connection) does not relax this limitation for two major reasons The same data have a different representation in an expert system program and in a database application program Hence, consistency of the data shared between the two systems is hard to manage The database system is simply a storage system with its interface restricted to "store", "load" and "call-sql" commands Second, loosely-coupled systems have severe performance limitations Most often, the rule-based language of the expert system shell is too general (it is a general purpose programming language) and lacks a well-defined semantics, thereby making program optimization a difficult problem

Research on active databases [Dayal88, McCarthy89, Stonebraker88] have shown the benefits of making a rule-based system a central component of an active database system However, rule-based systems as designed for expert system shells are inadequate for active DBMSs [Dayal88], mainly because they rely solely on main memory structures (RETE-like structures) and they cannot handle asynchronous updates These reasons have motivated the design of ad-hoc rule-based systems integrated with active database systems

Comparatively, research on deductive databases has resulted in a comprehensive theoretical framework to study rule-based query languages The Datalog query language is a good representative of logic-based query languages Much effort has been devoted to the processing and optimization of Datalog programs (see [Ullman89] for a survey)

Deductive database languages and their implementations do not seem appropriate for improving the development of real applications or even samples of them Regularly, the question "is there any real deductive database application?" is the basis of a panel session The area is often considered more paper-oriented (theoretical) than system-oriented (practical) Two reasons can explain this First, the Datalog-like languages

do not provide complete programming language capabilities control and meta-control structures (i e , expressive power), main memory variables (e g , user environment variables), procedure calls or side-effects, input/output interaction with the user They are also more query-oriented, database updates are hard to support within rule programs The second reason stems from the way relational database system are built In a relational database system, abstract layers range from evaluating high-level relational calculus expressions to lower-level operations All the operations are performed on the data at the lowest levels and the final result is finally returned to the interface process In comparison, many expert system shells are built on LISP environments Therefore, all the power of LISP is available to the rule programmer who can directly reference LISP functions within rules

We believe in the challenge of "using deductive database technology to build efficient systems that improve application programmer productivity" The development of the LDL system [Tsur86], at MCC provides a substantial contribution in this direction The LDL language [Naqvi89], is a Datalog-like language extended with powerful constructs such as negation, updates, control structures (cut, choice operator), and a data model including complex terms such as sets, functors and externals (e g , C procedures) [Chimenti89] The general approach is to define a declarative semantics (actually, a fixpoint semantics) for a logic programming language close to Prolog Efficient compilation and optimization of LDL is thus possible

Our approach is in the spirit of the LDL system However, it has some important differences First, our kernel language is a production rule language, called RDL1, [Maindreville88a] RDL1 is formally defined as an extension of Datalog with multiple literals in heads of rules and negative or positive literals in bodies and heads of rules [Abiteboul89] A negative literal in a head is interpreted as deletion of facts Second, the RDL1 system is integrated within a relational database system One consequence is that all the data manipulated in the rule language can also be queried or updated using an extended version of SQL Another consequence is that all the standard database features are supported by the RDL1 system A last difference with the LDL system is that we have an extensible architecture with Abstract Data Types (ADT) [Kiernan89]

Other related work can be found in [Delcambre88, Widom89, Stonebraker88b, Stonebraker89] In [Delcambre88], the use of database technology to support OPS-like languages is investigated As a result, their proposed language, RPL, is very similar to our language The Starburst project emphasizes the extension of a database system with a production rule-based facility, [Widom89] Their rule language expresses triggers in a way similar to [Dayal88] This work is therefore complementary to ours Finally, the POSTGRES project, [Stonebraker88b], proposes the extension of a relational database system with rules The limitations of this rule language are analyzed in [Stonebraker89] no support for view processing, problems for controlling rule activation, etc A new language, called PRSII, is proposed to alleviate these limitations This language is very close to the one described in [Widom89]

This paper presents the design and implementation of the RDL1 system with emphasis on practical aspects The system includes many programming features usually found in expert system shells for which we have provided a formal semantics and an efficient implementation The paper is organized as follows Section 2 presents the data model and the languages supported by the RDL1 system Section 3 presents the overall system architecture This architecture is an extension of an existing relational system Section 4 describes the techniques used to implement expert system shell features in the database framework Also, lessons learned from the implementation of the system are given Section 5 concludes the paper and points out open issues

2. Data Model and Languages

The two input languages of the system are RDL1, and an extended version of SQL The latter one is described in [Kiernan89] First, we briefly describe the underlying data model of these two languages which is the relational data model with ADTs, in the spirit of [Stonebraker88, Osborn87 Wilms88] Then, the rule language is presented

2.1. Abstract Data Types

The support of ADTs provides a rich typing capability for relational database systems An ADT is a type described by its operational semantics [Guttag77], i e , by a set of operations which can be performed on the instances of that type For example, the type Stack is described by the push, pop and top operations and not by the data structure that implements a stack

The ADT capability is supported by User-Defined Data Types (UDT) and User-Defined Functions (UDF) UDTs generalize the notion of domain in the relational model Thus, a domain is defined either as a basic data type (real, integer, boolean, string) or as a user-defined data type built from basic data types and type constructors that depend on the UDT implementation language Previously defined UDTs can be used in turn to build new data types From the languages point of view (extended SQL and RDL1), an ADT is viewed as a set of (user-defined) functions, that operate on instances of the defined type

The UDT/UDF implementation language supported by the system is LISP From the user point of view, LISP creates and manipulates complex hierarchical structures required by applications An interpreted environment protects the DBMS from abnormal termination in case of programming errors LISP as a general support for extensible programming is discussed in Section 4

2.1.1. User-defined Data Types

A UDT is specified using functional notation The name of the data type is also the name of a Boolean function that evaluates to True if its parameter qualifies as an instance of the defined type UDTs can be recursively defined using basic data types and existing UDT Furthermore, UDT are described in a ISA hierarchy capturing the usual notion of type inheritance Thus, UDT operators can be inherited along the hierarchy The set of basic types together with UDTs form the set of domains available to the programmer

The specification of a new type requires the use of the CREATE DOMAIN primitive It specifies the name of a super-type if it is a specialization of another type The type specification is the code used to check whether a datum is an instance of the type A UDT type specification is a LISP function that returns either a True or a Nil value Its syntax is

```
CREATE DOMAIN <super-type name> <type name>
        AS <type specification>
```

For instance, one could define a new "polygon" domain as a specialization of "list of points"

```
CREATE DOMAIN list-of-points polygon
        AS <type specification>
```

Then, a new relation called "map" can be created with a reference to this domain

```
CREATE TABLE map(mapid integer, contour polygon,    )
```

2.1.2. User-defined functions

User-defined functions (UDF) are built from a library of primitive functions These include standard LISP functions and window management functions (a C package) to implement graphics and user interaction The specification of a new function over UDT requires the use of the CREATE FUNCTION primitive It specifies the name of the function, the type of its arguments, and the type of its result The syntax of the command is

```
CREATE FUNCTION <function-name> (par1 T1,    parn Tn)
OF TYPE <type-name> AS <function-body>
```

where T1, , Tn are the respective types of the parameters par_1, , par_n The functions are inherited from a type to its subtypes A UDF can also be redefined on a subtype (overloading) For instance, the surface function defined on polygon can be redefined for triangle The selection of the code corresponding to a function name is done according to the type of all the arguments of the function

For example, the surface operator for polygons can be defined and stored as follows It accepts one argument of type polygon (or a specialization of this type) and returns a real number as result

```
CREATE FUNCTION surface (x polygon) OF TYPE real AS
```

Our approach is centered around one special purpose programming language for ADT programming One disadvantage of this approach is that the user has to learn it One advantage is that the implementation takes care of passing parameters to and from the DBMS and the language constructs The approach to ADT presented in [Wilms88] for the Starburst system allows ADTs to be implemented in any programming language This is an attractive feature However, for each new ADT and for each programming language that will be used to manipulate the ADT, two special purpose conversion routines, called IN and OUT, must be supplied to convert the ADT from its internal DBMS representation to that of the programming language and conversely

2.2. The RDL1 language

The kernel of the RDL1 language is first presented This kernel has been initially described in [Maindreville88a] Theoretical aspects such as expressiveness, complexity and fundamental properties (functionality, loop-freeness) of the language formalized as a Datalog extension, have been studied in [Simon88, Abiteboul89] The programming constructs provided in the language are discussed in the last subsection

2.2.1 The Kernel Language

An RDL1 program is composed of a set of if-then rules The IF part of a rule (also called condition part) is a tuple relational calculus expression The THEN part of a rule (also called action part) is a set of *actions* that are either insertions, deletions or updates of tuples in a relation A range restricted condition imposes that all the variables that appear in the action part also appear positively in the condition part of the rule

The semantics adopted for the language mixes non-deterministic and deterministic aspects Non-determinism is in the arbitrary choice of a rule to fire at each step of program execution Determinism is in set-oriented rules as opposed to instance-oriented rules A rule is *firable* if

its condition part evaluates to True in the current database state for a particular instantiation of the free variables of the condition, and if firing it modifies the current state *Firing* a rule consists in modifying the current database state using the action part of the rule More precisely, the firing operation is done as follows First, a relational query that corresponds to the condition part of the rule is run This query returns a result relation whose schema is given by the set of free variables of the condition part Thus, a rule is firable whenever the query returns a non-empty result Second, with each action (an insert, a delete, or an update) is built a relation, obtained by a projection over the query result on the arguments of the action The system enforces that the relations associated with actions are pairwise disjoint sets by eliminating the intersections This guarantees that the order of actions in the action part is irrelevant This firing process is clearly deterministic

The execution of a program describes a state transition diagram over database instances A database instance is reached from a current one by firing a rule chosen at random among the set of firable rules in the current instance A program execution terminates when no more rules are firable

2.2.2 Programming Constructs

In order to match real application needs, the kernel language has been extended to include several programming constructs These are main memory variables, externals, and procedural control The key idea is that each of them can be redefined in terms of the kernel language, thereby benefiting from a formal semantics Thus, a programming construct is often a syntactic sugaring that can be easily detected by the rule compiler for ad-hoc optimization Furthermore, rule programs are encapsulated into rule modules to help structuring programs

Rule module
The *rule module* is the compilation unit for the RDL1 system A rule module is also associated with an access right mechanism that protect rules from other users It is created with the CREATE MODULE <module name> command Its output interface is a set of deduced (i e, intentional) relations computed by the rule program These relations are declared using the "OUPUT <relation name>, [<relation schema>] " statement A module takes as input a set of base (i e, extensional) or deduced relations The output relations can be queried similarly to the base relations Deduced relations that are not declared as output relations implicitely are *temporary* relations They store intermediate results in the module The schema of a temporary relation can be either infered by the language analyzer or explicitely given by the user In the latter case, a temporary relation is declared using the "TEMPORARY <relation name>, [<relation schema>] " statement The internal structure of a module is composed of three parts a variable section, a rule section and a procedural control section A module creation terminates with the END key-word

The following example shows a module declaration with one output relation named *trip* which has three attributes The third attribute is defined over a UDT

```
CREATE MODULE dispatching,
OUTPUT trip(driver string, truck string,
itinerary path),
 END,
```

The notion of module is very similar to the popular notion of *rule set* in rule-based expert system shells Rule sets are also used in the new rule-based language, PRSII [Stonebraker89]

Rule and control section
The *rule section* is made of a sequence of rules, each one preceded by a statement "<rule name> " The section terminates with the "END RULES," statement The *control section* consists of a string that describes an explicit ordering between rules Basic symbols in this string are rule names Two particular symbols BLOC(string) and SEQ(string) can be recursively used to build a control string A string is evaluated from the inner-most paranthesized part to the outer-most one The string BLOC(string1) means that string1 must be executed up to saturation without any ordering consideration of the symbols contained in string1 On the contrary, SEQ(string1) means that string1 must be evaluated up to saturation using a total ordering of the symbols in string1 from left to right Thus, a control string can either describe a sequential, a stratified or a non deterministic execution, or a combination of these [Maindreville 88b] This control can always be simulated with the kernel language [Abiteboul89]

The following is a control string declaration over the rules r1, , r5

```
SEQ (SEQ (r1 r2 BLOC (r3 r4)) r5)
```

It enforces a computation of the form $(((r1^{\sigma}\ r2)^{\sigma}\ (r3^{*}\ r4^{*})^{\sigma})\ r5)^{\sigma}$ using standard notation for context-free grammars The notation $()^{\sigma}$ stands for "fire up to saturation" the content of () [Maindreville88b]

Local and environment variables
Main memory structured variables can be used in the rule section They are useful to pass parameters between rules or to communicate with the user environment There are two kinds of variables *local* variables defined in the *variable section* of a module, and *environment* variables that are part of the rule evaluator

Each local variable ranges over a domain of values which is a basic type or a UDT It is declared using the statement "VAR <type name> <variable name>", optionally followed by the statement " = <default value>",

which initializes it A local variable is used like a constant in the condition part of a rule Its value can be modified in the action part of a rule "<variable name> = value" To simulate local variables in the kernel language, each variable corresponds to a relation with a single attribute which contains one tuple since a local variable is monovalued

The following is an example of a variable declaration and manipulation in a rule section

```
VAR integer amount  = 10,000,
r1   IF trip(x) and cost(x itinerary) >= amount
THEN + expensivetrip(x),
END RULES,
```

Environment variables capture parameters of the user environment For instance, consider the size or location of windows used for interacting with the user These variables are made visible from rule programs to the programmer They can be explicitely modified by rules during the program execution The only difference with local variables is that they are predefined and initialized by the rule evaluator Environment variables are identified by a name, a built-in type name, a value, and can only be manipulated through a set of predefined functions

Externals

Routines written in LISP (also called *externals*) can be called from the action part of rules The code of an external may modify the user environment For instance, it may display text in a window However, it will never modify the database state

An external is declared using the statement "DO <routine name>(<arguments>)" immediately after the THEN clause in the action part of a rule The routine is executed if and only if the rule is fired The following is an example of external used in a rule

```
r1   IF trip(x) and cost(x itinerary)<4,000
        THEN + goodtrip(x)
              DO display(x itinerary),
```

Externals can be simulated in the kernel language The above rule is simulated as follows First, a relation of schema CALL (external string) is created Then, rule r1 is equivalent to the rules

```
r0   IF goodtrip(x) THEN old_good(x),
r1   r1 without the DO statement
r2   IF goodtrip(x) and not old_good(x)
        THEN + CALL(display(x itinerary)),
r3   IF CALL(x) THEN  - CALL(x),
```

Rule r0 saves the old state of goodtrip Rule r2 is firable whenever r1 has been fired Firing r2 requires evaluating the function *display* contained in its action part, thereby side-effecting the user environment The last rule enforces the fact that the CALL relation is always empty Thus, the rule containing the call statement is always firable since it will modify the database state (a tuple with a Nil value, for instance, will be inserted) The necessary control for ordering the above sequence of rules is available

A rule example We show how the "Mrs Manners" problem can be solved using the RDL1 program below We assume that two relations are given The first one, *guest(name, sex, hobby)*, gives for each guest, his/her name, sex and hobbies The second one, *nextseat(seat1,seat2)*, specifies the adjacent seats A seat is identified by an integer For instance, a possible input is nextseat(1,2), nextseat(2,3), Guests must be seated so that neighbours are of different sex and share a hobby (To simplify, assume that the number of seats equals the number of guests) The *nextguest(guest1,guest2)* relation (indicating who sits next to whom) is computed using the first rule Guests are all considered for the first seat using the second rule The third rule recursively expands a graph of all possible solutions for seating guests The actual solutions are produced by the fourth rule The temporary relation *current(value)* holds the number of the current seat where somebody has to be seated The value *firstseat* holds the number of a given initial seat Relation*seating(seat1,name1, name2, seat2,t-map)* tells who sits where and next to whom, and records a table map in the *tmap* attribute of type list of names (a UDT) The two functions *cons* and *length* are standard LISP functions inherited by the UDT Finally, the variable *started* is used to control that the initialization of the seating relation is performed once The rules are as follows

```
CREATE MODULE MISS-MANNERS,
OUTPUT
seating(seat1 integer, name1 string, name2 string, seat2
integer, tmap list_of_names),
TEMPORARY
current(value integer),
nextguest(guest1 string, guest2 string),
VAR  boolean started  = false,

r0   if true thenonce + current(firstseat),

r1    if guest(x)  and  guest(y)  and  x sex<>y sex  and
x hobby=y hobby
then + nextguest(x name, y name),

r2   if   guest(x) and nextguest(y) and x name=y name1 and
nextseat(z)  and current(t)  and \z seat1=t value  and  not
started
then + started =true, + current(z seat2)- current(t)
+seating(z seat1,x name,x name,z seat1,cons(x name, nil)),

r3   if   seating(x) and nextguest(y) and x name2=y name1
and current(t) and nextseat(z) and z seat1=t value and
t value<>firstseat and not member(y name2, x tmap)
```

```
then+ current(z name2), - current(t) +seating(x seat2,
x name2, y name2, z seat1, cons(x tmap, y name2)),

r4 if nextseat(x) and current(t) and t value=x seat2
and t value=firstseat and seating(y)
and length(y tmap) <x seat1
then - seating(y),
END RULES,
SEQ (BLOC(r0 r1 r2 r3) r4),
END
```

3. System Architecture

3.1. Functional Architecture

The functional architecture is divided into three layers the programming environment, the RDL1 compiler and deductive system, and the relational data storage system Figure 2 shows the architecture

The environment [Kiernan90] is built over the X Window manager The editors are modified versions of the standard X window editors These are the rule program editor and the ADT editor ADT source code is directly executable by the run-time system Thus, no compilation is necessary before loading the code into working memory The code is simply read in by the run-time environment

A rule program is first processed by the *RDL1 compiler* which produces a graph-based internal structure called Production Compilation Network (PCN) [Maindreville88b] The *rule evaluator* runs rule programs (in PCN form), invoked when an SQL query refers to a deduced relation, or when a *run* command is issued A logical optimization phase precedes the evaluation phase The optimizer performs a source-to-source transformation of a PCN The rule evaluator issues extended relational operations to the *Relational Data Storage System* SQL commands are parsed and processed by the Deductive System It performs the traditional functions of a relational database system such as data definition, semantic data control and query processing

The *Relational Data Storage System* is a low level relational DBMS which is part of the SABRINA relational system [Valduriez89] It has been extended with a module that performs the evaluation of the abstract data types captured by our data model The shaded part on the Figure 2 represents the traditional code of a relational database system The white part represents the code developed to make the RDL1 system

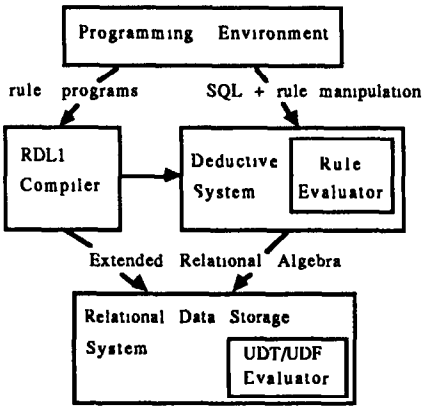

Figure 2 RDL1 system architecture

3.2. Process Architecture

The process architecture is shown in Figure 3 The distinction between the end-user interface processes and the database server process is reconsidered Each end-user has one process which comprises all the modules of the architecture portrayed in Figure 2 Users working simultaneously on the same database create their process on the same machine The XServer takes care of rerouting I/O to the user's workstation All user processes on a same machine share a common memory block in the UNIX shared memory This memory space is used to store cache memory, a catalog (data access manager), schemas (data dictionary manager), and transaction timestamps (concurrency control manager) Concurrent accesses to the shared memory are controlled using semaphores managed by the corresponding modules of the DBMS Since there is one process per user, users are not blocked while one process is waiting for disk accesses

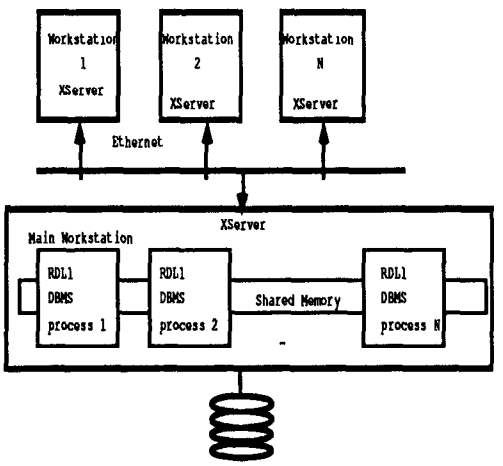

Figure 3 Process architecture

User accesses are functionally issued from three levels the programming environment, the rule evaluator, and

the UDT/UDF evaluator Since there is a single process, all I/O are issued and bound to the same process This provides a coherent environment for debugging, tracing and running rule programs with user interaction issued from all levels

4. Implementation

One of the key implementation choices was the design of a common module of code for implementing the rule evaluator, and for evaluating UDTs and UDFs This common module is essentially a LISP interpreter tailored for our specific needs, and integrated with the relational data storage system One advantage is that all the environment variables managed by the rule evaluator are shared with the UDT/UDF evaluator Furthermore, since the environment variables, the ADT library, and the rule evaluator are managed by the same module, it provides an integrated platform for implementing the programming environment A second advantage of this approach is the extensibility of the rule evaluator and the programming environment

4.1 Choice of LISP

As a basis for extensibility, LISP provides the features of a shell together with the features of a programming language LISP programs can be called interactively by the user in a sequence of command-answer cycles, where a command can be a simple expression or a complex expression involving all the power of LISP This feature is useful to build an interactive programming environment Futhermore, it enables a high level implementation of system extensibility By system extensibility, we refer to the ability of modifying the system interface using itself For instance, the addition of new ADTs in the data dictionary modifies the interface offered by the query language We shall see that the same mechanism (i e , addition of new functions in the data dictionary) also allows to modify the interface of the rule evaluator

Our approach to system extensibility can be compared to that of the Starburst project [Haas89] Query processing in Starburst is made extensible by using the same programming language as the DBMS implementation language The user may access the structure of a query manipulated by the DBMS, called a Query Graph Model (QGM), modify it, and return it to the DBMS The advantages of this approach are the simplicity of linking a user program with the DBMS code, efficiency and generality offered by the QGM structure Howewer, the interface for extensibility, QGM, presents implementation details at the level of a database system implementor This may not be easy to exploit Since the user may perform any kind of modification on a QGM structure, this structure can be corrupted Therefore, bad programs can be run with the DBMS code and cause a DBMS crash In our approach, LISP provides a higher level of abstraction For instance, LISP objects include relations, queries and other specialized objects These objects are presented as abstract entities thereby hiding many of the more complicated details The interpreter detects user programming errors and avoids running the DBMS into errors

4.2. The Rule Evaluator

This section first presents the general evaluation mechanism of a rule program Then, it describes how the rule evaluator extensibility is implemented Finally, the implementation of environment variables is given

Evaluation of rule programs .
The execution phase is a cycle consisting of three actions match, select, and fire The interpreter finds all the relevant rules whose conditions *match* the current database state The rule evaluator generates a query corresponding to the condition part of the rule and the relational query optimizer processes it A temporary relation is returned by the relational data storage system If it is not empty the rule is said *relevant* Among the set of relevant rules, the *conflict set* is built with all the firable rules (i e , the ones that can change the database state) To determine it, the rule evaluator computes the effect of each relevant rule This requires processing union (or difference) operations between the temporary relation associated with the rule and the relations that appear in its action part Finally, the rule evaluator *selects* one rule in the conflict set and *fires* it

An optimization issue is choosing the next rules that should be considered for relevance at any step of the execution cycle This determines the number of rules examined concurrently in the conflict set A strategy for such a choice is called the conflict resolution strategy In our system, the conflict resolution strategy is based on firing by *stepwise saturation* and *chaining* It induces an implicit partial ordering among the rules of the conflict set This ordering is defined as follows If two rules r_1 and r_2 are in the conflict set, then $r_1 \leq r_2$ iff there is a path in the PCN going from r_1 to r_2 If $r_1 \leq r_2$ and $r_2 \leq r_1$ then $r_1 \equiv r_2$ This means that the contents of a relation must reach a stable state before considering an other rule The control string specified in a rule program is consumed by the rule evaluator It adds supplementary ordering constraints between rules From an optimization point of view, this strategy minimizes the number of relational queries generated to execute the program For instance, it leads to fire a non recursive rule only once Indeed, a rule is first tested for relevance according to the partial order If the test fails, a next rule is chosen in the hierarchy Similarly, if a rule is relevant, one tests if it is firable If not, another rule is chosen

Rule evaluator extensibility :
The rule evaluator is a LISP program that evaluates a rule program compiled into a PCN format A set of primitive operations corresponds to basic operations over the PCN and its elements which are places, arc labels, transitions and relations For example, relations, transitions and places are accessible as LISP objects At the rule evaluator level, operations performed over relations are `union, intersection, merge, difference, remove-doubles, tuple-count` Calls to these LISP primitive operations are calls to the relational database storage system operators Results of the operations are returned as LISP objects

The functions that implement the rule evaluator are stored in the data dictionary Some of these functions are made visible to the end-user through the programming environment Examples of these are the functions that determine the strategy of the evaluator for processing a rule program, the functions that evaluate and fire a rule, or the functions that visualize intermediate changes to the database made by one or more rule They are easily modifiable through the ADT editor This enables the application program implementor to adjust the strategy of the rule evaluator according to the application needs Also, these functions allow debugging rule programs

Main memory variables :
Main memory variables may appear in the condition part or in the action part of a rule Consider the following rule action

```
var1 .= function (<arguments>)
```

where the function arguments cannot contain relational calculus expressions (e g , an attribute value of some relation tuple) Such an expression is evaluated in main memory by the LISP interpreter Now, consider the following expression occuring in the condition part of a rule

```
x att = var1
```

Where x is a variable ranging over a relation, att is an attribute name of that relation and var1 is a variable name This expression cannot be evaluated in main memory by the LISP interpreter because it belongs to a relational calculus expression It must be evaluated by the module which performs relational operations Thus, the expression is compiled into the form <attribute relop constant> whereby var1 is replaced by its value and assigned to the constant Then, the DBMS is able to process the relational selection

4.3. The UDT/UDF Evaluator

User-defined functions are processed by a LISP interpreter The processing of LISP expressions must be reconsidered in a database context The main interface to the ADT evaluator is

```
Procedure ApplyUDT (opname text,
arglist list) result,
  begin
   lispargs  = convert (arglist) ,
   ApplyUDT  = APPLY(opname, lispargs) ,
 end,
```

This simplified procedure receives two parameters which are respectively the name of the UDF to evaluate and the argument list The argument list is converted into a convenient representation for the LISP interpreter Then, the function is applied to this list using the standard LISP APPLY function which returns the result of the operation

The primitive LISP functions are implemented in the same language as the DBMS The graphics and window primitives are implemented in C over the X window interface

A simplification made with respect to usual LISP interpreters is to allow local variables only in UDF functions and to disallow property lists These are supported in the rule evaluator programming environment Therefore, only function code persists between ADT evaluation cycles Since the interpreter is essentially an ADT method evaluator and no user environment needs to be maintained between evaluation cycles, this restriction is not cumbersome to the ADT programmer These two restrictions greatly simplify the task of identifying garbage Function code is marked as non garbage and thus stays valid for the lifetime of the transaction All memory consumed during a cycle can be reused during the next cycle The memory allocation procedure has been altered to allocate old memory fragments until those fragments have been deleted At which point, basic system calls are run to allocate new memory Indices provide quick access to memory elements So, garbage collection does not hinder performance in this context

4.3 Lessons learned

During the implementation of the RDL1 system, we lgained valuable experience concerning the RDL1 language, the programming environment and system performance

- In the first specification of RDL1 [Maindreville88], the semantics of the language were purely non-deterministic (instance-oriented rules) a rule is firable for each combination of tuples satisfying its condition part Such a semantics, usually implemented in production rules systems [Brownston85], revealed to be very difficult to implement in a relational database system based on set-oriented query processing [Regnier89] Therefore, we

decided to implement another semantics where rules are set-oriented When a rule is fired, its action part is applied for all answer tuples to the relational query associated with the condition part of the rule Hence, rules can easily be mapped into relational queries Similar observations seemed to guide the design of a set-oriented rule language in Starburst [Widom89] and Postgres [Stonebraker89] Howewer, set-oriented rules are less expressive than instance-oriented rules This may pose problems in practical use For instance, one cannot solve the "Mrs Manners" problem of Section 2 by randomly returning only one configuration for the table map

• The presence of deletions in rules makes program termination difficult to control A rule may insert tuples in a relation that are deleted by another rule If the insertion rule is firable again then the program may enter an infinite loop A solution is to use special "control predicates" that inhibit a rule to be fired more than once with the same tuples Main memory variables may be useful as control predicates This is the case in the rule example of Section 2 with the *started* variable

• Finally, it appeared necessary to separate static control information from the rules This is for performance reasons because control information could be compiled and consumed by the rule evaluator in an efficient way Programs become more readable and control is easier to change

Three crucial points for optimization have been isolated while experimenting with the system managing temporary relations, detecting halting conditions and storing rules

• Firing a transition implies recomputing the conflict set because the database state has changed This task is the major time-consuming part of the execution cycle Thus, it deserves special effort for query optimization Two main tasks are involved in the recomputation recomputing the temporary relation associated with a rule, and computing the effect of the rule This can be time-consuming for non monotonic programs because (i) a rule for which the test of relevance failed at one step may become firable, and (ii) a rule that was fired may become firable again This means that a similar task can be repeated on consecutive cycles We proposed a first algorithm [Regnier89] which maintains some information accross the execution cycles, in order to optimize this task

• Another problem is the efficient storage of temporary relations in main memory in order to perform (i) a fast evaluation of the temporary relation associated with a rule, and (ii) fast unions and differences involved in the firing of a rule Standard relational DBMS do not treat temporary relations as first class citizens Indeed, most DBMS store temporary relations as sequential files The first measurements we performed on our prototype exhibit that improvements could be obtained by using indexing schemes for temporary relations Nevertheless, assuming that temporary relations can be indexed in main memory, determining the most appropriate way of indexing them is a very complex task The PCN model [Maindreville88b] needs to be extended to capture physical information such as sorted relations, cardinality of relations, indexing schemes A first step in this direction is to make the rule evaluator knowing which temporary relations have already been sorted This allows to optimizing union operations

• A good storage structure for rule programs is needed It should allow an efficient access to rules pertinent to a query The time ratio spent for accessing rules during query processing can be important A special algorithm based on a double hashing of the join attributes was designed to reduce this overhead [Cheiney89]

5. Conclusion and Open Issues

This paper presented and motivated the architecture of the RDL1 system The approach is centered around the integration of a production rule language with a relational database system, the development of a rule-based programming environment, and the support of system extensibility using ADT facilities The programming environment offers a better structuration of rule programs, the use of several programming constructs usually found in programming languages and expert system shells, flexible I/O facilities for displaying results and obtaining user input All these features are critical for making deductive database a practical technology, i e , supporting large rule-based programs A version of the RDL1 system, operational on UNIX workstations, was demonstrated at the SIGMOD'89 Conference

The implementation of this system provided rich feedback on the rule language semantics, the system performance bottlenecks and the programming environment In particular, constructive criticisms were given by users who developed an "Intelligent Training System (ITS)" as an RDL1 application, in the framework of the ISIDE Esprit project The ITS application is intended to teach maintenance personal to diagnose and repair helicopter failures Many difficulties were due to the lack of a programming environment and programming constructs in the rule language

Based on this experience, we started a new project with the objective of developing a more powerful rule-based system for large databases The design choices made for this system are to offer both instance-oriented and set-oriented rules in the language, to develop a compiler/optimizer besides the rule interpretor better suited for prototyping, and to extend the programming

environment with triggers and integrity constraints expressed as rules This last functionality is required by the ITS application

Acknowledgements

We are grateful to Patrick Valduriez for the fruitful discussions we had with him and his invaluable comments to improve the presentation of the paper We also wish to thank Max Jean-Noel, Nathalie Lefebvre, Dominique Pastre and Hervé Stora for their participation in the implementation effort of the RDL1 system

References

[Abiteboul89] S Abiteboul, E Simon "Fundamental Properties of Deterministic and non-deterministic Extensions of Datalog", to appear in *Journal of Theoretical Computer Science*

[Brownston85] L Brownston, R Farrell, E Kant, N Martin "Programming Expert Systems in OPS5 An Introduction to Rule-Based Programming", book, *Addison-Wesley Ed* , 1985

[Cheiney89] J P Cheiney, C de Maindreville "Relational Storage and Efficient Retrieval of Rules in a Deductive DBMS", *Proc of Int Conf on Data Engineering*, Los Angeles, Feb 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

[Guttag77] Guttag J V et al , "The Design of Data Type Specifications", *Current Trends in Programming Methodology, Vol IV* Data Structuring, Raymond T Yeh, ed , 1977

[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

[Kee85] Intellicorp "KEE Software Development System User's Manual", *Intellicorp Mountain View*, 1985

[Kiernan89] 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", *INRIA Research Report N° 1153, Jan 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

[Maindreville88b] C de Maindreville, E Simon "Modelling non-deterministic Queries and Updates in a Deductive Database" *Proc of Int Conf on VLDB*, Los Angeles, Aug 1988

[McCarthy89] D McCarthy, U Dayal "The Architecture of an Active Database Management System", *Proc of ACM SIGMOD Int Conf* , Portland, June 1989

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

[Osborn86] S L Osborn and T E Heaven "The Design of a Relational Database System with Abstract Data Types for Domains", *ACM TODS*, Vol 11, N3, 1986

[Regnier89] M Régnier, E Simon "Efficient Evaluation of Production Rules in a DBMS", *Proc of Advances in Databases*, edited by INRIA, Geneva, Sept 1989

[Stonebraker88] M Stonebraker "Inclusion of New Types in Relational Database Systems", *Proc of Int Conf on Data Engineering*,, Los Angeles, Feb 1988

[Stonebraker88b] M Stonebraker et al "The POSTGRES Rules System", *IEEE Transactions on Software Engineering*,, Vol 14, N° 7,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

[Tsur86] S Tsur, C Zaniolo "LDL a Logic Based Data Language", *Proc of Int Conf on VLDB*, Kyoto, Aug 1986

[Ullman88] J D Ullman "Principles of Database and Knowlege-Base Systems", book, *Computer Science Press, Vol 1 and Vol 2*, 1989

[Valduriez89] P Valduriez, G Gardarin "Analysis and Comparison of Relational Database Systems", book, *Addison Wesley*, 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

[Wilms88] P F Wilms et al "Incorporating data types in an extendible database architecture", *IBM Research Report, RJ 6405*, Almaden Res Center, Aug 1988 Also in *Proc of Int Conf on Improving Data Responsivness and Usability*, June 1988