A Self-Controlling Interpreter for the Relational Production Language

Lois M. L. Delcambre and James N. Etheredge

The Center for Advanced Computer Studies
University of Southwestern Louisiana
P.O. Box 44330
Lafayette, Louisiana 70504 USA
(318) 231-6284

Abstract

The Relational Production Language (RPL) solves the paradigm mismatch between expert systems and database systems by relying on the relational data model as the underlying formalism for an expert system. The result is a formally-defined production system language with immediate access to conventional databases. Working memory is modeled as a relational database and rules consist of a relational query on the left hand side (LHS) and database updates on the right hand side (RHS). This paper reports on the design of the RPL 1.0 prototype. The prototype directly executes RPL programs and capitalizes on the inherent advantages of the relational approach, particularly for intra-rule and inter-rule parallelism. By using a self-describing approach for representing the interpreter data structures, the interpreter is a self-controlling system that allows conflict resolution, error handling and a wide spectrum of software metrics to be explicitly specified using RPL meta-rules.

1. Introduction

The Relational Production Language, RPL, is a system that establishes the relational data model as the underlying formalism for a production system. Working memory is modeled as a relational database and production rules consist of relationally complete queries on the left hand side (LHS) and database update operations on the right hand side (RHS). RPL eliminates the problem of two disparate computational paradigms for a knowledge base management system, and provides a formal framework for the investigation of the internal workings of a production system. RPL has been shown to be strictly more expressive than OPS5 [Delcambre87a]. The direct manipulation of an entire set of instantiations supports intra-rule parallelism and the notion of serializable execution of transactions provides for inter-rule parallelism. Finally, query optimization/query processing techniques are applicable to the pattern matching capability required for the RPL interpreter. The use of a relational language for the purposes of forward chaining was suggested by Stonebraker [Stonebraker86] in the context of the Ingres database management system.

A series of RPL prototypes are currently under development and this paper reports on the first in the series, RPL 1.0. The primary motivation for this prototype is to demonstrate the feasibility of the approach. RPL 1.0 relies on SunUNIFY [Sun86], a relational DBMS, to process the schema DDL, LHS queries, and RHS actions that appear as part of the RPL program. Thus the prototype relies on relational query processing for pattern matching rather than the Rete algorithm [Forgy82] normally used for production systems. RPL 1.0 re-executes the LHS query whenever a base relation referenced by the query is modified. The 1.0 prototype implements the basic recognize/act cycle for a production system by determining the conflict set, performing conflict resolution, and firing the selected rule(s).

Implementation plans for RPL include a second prototype, RPL 2.0, that will support working memory with a main memory database system [Bitton86]. The performance of RPL 2.0 should compete with other memory-resident production systems and RPL 2.0 should retain the potential for parallelism inherent in the RPL language. The ultimate goal of this research is to develop RPL 3.0, a two-level database system. The RPL 3.0 architecture will simultaneously support conventional disk-resident databases and a RPL 2.0 system with automatic migration of data between the two levels. The design challenges for each of the RPL prototypes are presented in [Delcambre87b] and are discussed briefly in Section 6.

The contribution of this paper is the presentation of the design of the RPL 1.0 prototype based on a commercial-quality, relational DBMS. By extending the self-describing approach of a relational database [Mark85] to the internal workings of the production system, it is possible to express interpreter directives using RPL rules. RPL 1.0 is implemented as a two-level interpreter: the lower level supports the application and the upper level controls the inferencing process. Meta-level RPL rules are used to implement the conflict resolution strategy, perform program tracing and error handling functions, and accumulate software metric data. When the meta-level reasoning is complete, the instantiations of the user-supplied rules chosen during the conflict resolution process are fired at the lower level of the interpreter. The final contribution of this paper is the description of how the user-supplied schema and rules can be automatically modified to support the internal workings of the interpreter. This preprocessing simplifies both the RPL interpreter and the user-supplied RPL programs.

Throughout this paper, the reader is assumed to be familiar with both relational query processing and production system languages [Date86, Maier83, Brownston85]. The organization of the paper is as follows. The RPL language is introduced in Section 2. Section 3 presents the details of the RPL design and Section 4 discusses the self-controlling aspects of RPL. Section 5 describes the functions performed by the RPL preprocessor. Finally, an evaluation of the RPL 1.0 design is presented, along with conclusions and future work, in Section 6.

2. The RPL Language

A RPL program consists of three types of statements: data definition statements, database load statements, and RPL production rules. Each of these statement types is discussed in the paragraphs that follow. A sample RPL schema and a sample rule is presented in Figure 1. They are part of a university application where students have taken courses. The sample rule cleans up a student's transcript by eliminating courses that have been re-taken with an improved grade.

Data definition statements are SQL CREATE DDL statements which define the schema for the relational database. Database load statements are SQL INSERT operations used to load data into the relational database. The RPL approach allows a particular schema definition and even a particular database state to be used for multiple RPL programs.

RPL rules are in the form of standard production system if/then statements. The LHS conditions are specified as SQL [Date86, Ullman82] queries. The

```
CREATE Student (
   Stud-id = Int(9),
   Stud-name = Text(25))

CREATE Crs-taken (
   Stud-id = Int(9),
   Crs-id = Text(10),
   Sem-taken = Text(3),
   Grade = Int(1))

CREATE Course (
   Crs-id = Text(10),
   Sem-offered = Text(3),
   Enrollment = Int(3),
   Instructor = Text(25))

Eliminate-duplicates:
FOR ALL
 SELECT C.Stud-id, C.Crs-id
 FROM Crs-taken C, Crs-taken T
 WHERE C.Stud-id = T.Stud-id
   AND C.Crs-id = T.Crs-id
   AND C.Grade <= T.Grade
   AND C.Sem-taken < T.Sem-taken
DO
 DELETE Crs-taken
 WHERE Stud-id = C.Stud-id
   AND Crs-id = C.Crs-id;
END
```

Sample RPL DDL and Rule
Figure 1

answer that results from the LHS query can be modified by SQL update operations appearing on the RHS of RPL rules. Figure 2 contains a syntax diagram for RPL rules. For each rule, the user has the option of specifying a rule name and a priority number in parentheses. When priorities are specified for rules, static rule priority can be used as the conflict resolution strategy thereby allowing the user to explicitly control the sequence of rule firing during execution. The keyword *FOR* is followed by the quantifier and the LHS query. The quantifier controls the processing of the answer set when the rule fires. The keyword *DO* separates the LHS query from the RHS actions. The keyword *END* terminates the RPL rule.

Section 2.1 discusses the utility of the quantifier and the sort sequence in a RPL rule to support conflict resolution. This is followed by a discussion, in Section 2.2, of one major challenge presented by the RPL language, the explicit identification of an instantiation.

2.1. Rule Level Conflict Resolution

Because RPL relies on relational query processing to perform pattern matching, the manipulation of a set of instantiations during one firing of the rule can be considered. The ***instantiation set*** for a rule is the set of answer tuples that satisfy the LHS query. The RPL language relies on both the user-specified sort sequence

```
[rule-name] [(priority-number)] :
FOR quantifier
    LHS SQL query
DO
    RHS action 1;
    ...
    RHS action n;
END
```

Syntax Diagram of RPL rule
Figure 2

and rule quantifier to provide ***rule-level conflict resolution.*** The user thus controls how the instantiation set will be fired.

The quantifiers supported in RPL are *ALL*, *ONE*, and *FIRST*. The quantifier *ALL* specifies that all answer tuples retrieved by the LHS query are to be processed by the RHS actions of the rule during one firing (i.e., one cycle of the interpreter). The *ALL* quantifier thus supports intra-rule parallelism. The quantifier *FIRST* processes only the first answer tuple returned. However, the remainder of the instantiation set is left in the conflict set and may be processed during subsequent firings of the rule. This corresponds to the way OPS5 and other production systems work. The quantifier *ONE* processes the first instantiation and removes the remainder of the instantiation set from the conflict set. In this case, the unprocessed answer tuples do not fire in subsequent cycles of the interpreter and the LHS query serves as a one-time trigger for the RHS processing.

The next section discusses the problem of identification of the objects comprising an instantiation returned by a relational query.

2.2. The Identification of an Instantiation in RPL

The increased expressive power of the LHS of RPL rules over conventional production system languages (e.g., OPS5) complicates the explicit identification of the tuples that comprise an instantiation. Since the instantiation establishes the context for the execution of RHS updates, the explicit identification of tuples and/or values that comprise the instantiation is necessary.

The first issue to be addressed concerns data that participates in the query but does not appear in the final answer to the LHS query (i.e., does not appear in the attribute list of the outermost query). Consider the following query (on the LHS of a RPL rule) as specified by the user:

```
SELECT UNIQUE C.Instructor
FROM    Course C, Crs-taken T
WHERE  C.Crs-id = T.Crs-id
          AND T.Grade = 'A'
```

In this case, the answer to the query, and thus the instantiation, identifies only instructors of courses that have at least one enrolled student who received a grade of 'A'. The identification of the actual Crs-taken tuple (e.g. by T.Stud-id and T.Crs-id) is not of interest to the user. In terms of the RPL interpreter, this rule should re-fire only for a new, distinct course (when the first grade of 'A' is recorded for one of the enrollees) but should not re-fire when subsequent students receive a grade of 'A' for a course.

On the other hand, consider the following LHS query:

```
SELECT UNIQUE C.Instructor,
       C.Crs-id, T.Stud-id
FROM    Course C, Crs-taken T
WHERE  C.Crs-id = T.Crs-id
          AND T.Grade = 'A'
```

In this case, the identification of the student taking the course is an integral part of the instantiation. This rule should be fired for each Course/Crs-taken pair where the student receives a grade of 'A'. Thus, based on the users choice of attributes in the query, tuples from base relations where none of the attributes appear in the attribute list of the LHS query are not considered to be part of the instantiation.

The second issue concerns tuples from base relations that are explicitly modified or deleted by a RHS action. As discussed in [Delcambre87a], the RPL paradigm, as well as object-oriented systems in general, require that distinct objects in the database be explicitly identified by a permanent, internal identifier (i.e., a surrogate [Codd79]). Therefore, all user-defined relations are assumed to contain an object identifier, called a ***tuple-id*** that serves as a key for the relation. As a result, if a tuple from a base relation is modified or deleted by a RHS action of a rule then:

(1) the tuple-id must appear in the outermost attribute list for the LHS, and

(2) the RHS delete or modify action can identify the tuple to be manipulated based solely on tuple-id.

Note that this requires modifications to the user-supplied relations and rules and can be done automatically by the preprocessor discussed in Section 5 below.

There is one final issue to consider in the identification of an instantiation. The increased expressive power of RPL over OPS5 derives, in part, from the project operator from relational algebra with automatic elimination of duplicates. When the attribute list for a LHS query does not include the tuple-id or a key for a base relation, but does include some other attribute produced by the project operator, then the *value* of the attribute is of interest rather than the tuple-id. As an example, consider the second rule in Figure 1 above. There may be multiple Crs-taken tuples (at least one for each core course) that collectively result in one instantiation of the rule. The rule should be fired only once for each unique Stud-id, even if the student subsequently takes additional courses or repeats a core course.

Based on the use of tuple-ids to identify tuples from base relations and the use of values for attributes produced by the project operator, the precise identification of an instantiation for a rule in RPL is presented here. For an arbitrary LHS query, the instantiation is identified by:

(1) a potentially empty set of tuple-ids, one for each identifiable base relation tuple that appears in the attribute list of the user specified LHS; and

(2) a potentially empty set of attribute values, one for each attribute in the user-specified attribute list that is not part of any tuple identified by the set of tuple-ids listed in (1) above.

Note that both sets cannot be empty. An empty instantiation identification could only result from a null attribute list as specified by the user. For the rule presented in Figure 1, the identification of the instantiation is shown in Figure 3. The first rule requires the explicit identification of a Crs-taken tuple because of the delete operation in the RHS action.

Rule Name	Instantiation
Eliminate-duplicates	tuple-id for Crs-taken

Identification of the Instantiation
Figure 3

3. The Design of RPL

The paragraphs that follow describe the RPL interpreter and the flow of control within the RPL system. Section 3.1 discusses the data structures required by the RPL interpreter. They are presented in order to facilitate the description of the RPL 1.0 recognize/act cycle presented in detail in Section 3.2.

3.1. RPL Interpreter Data Structures

Figure 4 presents the data structures required by the RPL interpreter. These relations are provided in addition to the user-specified relations described in Section 2. The *Base* relation identifies base relations referenced on the LHS of a RPL rule. The *Rel* relation stores a flag used to indicate that a base relation has been updated. These relations are used by the interpreter to determine which rule instantiations in the conflict set have been invalidated due to the updating of a base relation on the previous interpreter cycle. These instantiations are removed from the conflict set.

The *Rule-lhs* relation contains the LHS query as well as an Instantiation-flag that indicates whether there is an instantiation for the rule currently in the conflict set. The *Rule-rhs* relation contains the RHS actions specified for each rule in the program. This relation is used by the interpreter when firing a rule. The Fire-group, Fire-group-data, and the CR-strategy relations are used for conflict resolution and are discussed in Section 3.2. Finally, the *Error* relation is used when an error occurs during the processing of a rule LHS or RHS action.

```
CREATE Base(
  Rule-id = Int(6),
  Relation-name = Text(25))
CREATE Rel(
  Relation-name = Text(25),
  Update-flag = Text(3))
CREATE Rule-lhs(
  Rule-id = Int(6),
  Rule-name = Text(25),
  Quantifier = Text(5),
  Instantiation-flag = Text(3),
  Query = Text(200))
CREATE Rule-rhs(
  Rule-id = Int(6),
  Rhs-action = Text(80))
CREATE CR-strategy (
  Goal = Text(10))
CREATE Fire-group (
  Group-id =Int(8),
  Rule-id = Int(8))
CREATE Fire-group-data (
  Group-id = Int(8),
  Rule-cnt = Int(8),
  Tuple-cnt = Int(8),
  Recency = Int(8))
CREATE Error(
  Firing-cycle = Int(8),
  Firing-number = Int(8),
  Rule-id = Int(10),
  Error-code = Text(80))
```

RPL Interpreter Data Structures
Figure 4

In addition to the data structures shown in Figure 4, the interpreter requires several relations for each rule in a RPL program. Based in part on the functioning of the Rete algorithm [Forgy82], production systems normally do not re-fire a rule for the same instantiation. In such systems, an object can be "refreshed" in order to allow it to fire again (as if it were a new object). This ability to control the firing of objects is normally supported through a *recency number*. In order to be upward compatible with such systems, RPL provides for a recency number to be associated with each object in working memory. Unlike production systems like OPS5, RPL distinguishes between the tuple-id which is a permanent part of the database and a recency number that is used to support conflict resolution.

Each object that participates in the instantiation set is identified by its recency number for the purpose of the history relation and is identified by its tuple-id for the purpose of the trace relation, as shown in Figure 5. Whenever a LHS query is executed, the instantiation is actually the answer for the LHS minus the appropriate history relation for the rule. Thus, the history relation is used solely to prevent the re-firing of rules for the same instantiation. The "refresh" capability is supported by giving an object a new, higher recency number. The ability to refresh an entire rule is also easily supported by deleting all tuples in the appropriate history relation.

The trace relation for a given rule includes the tuple-id and recency number for all objects explicitly identified in the instantiation and values for the remaining attributes. The trace relation also includes the interpreter cycle number and the rule firing number to identify the conflict resolution cycle and the instantiation that is firing, respectively.

```
CREATE Trace$$001(
  Firing-cycle = Int(8),
  Firing-number = Int(8),
  Crs-taken-tuple-id = Int(8),
  Crs-taken-tuple-recency = Int(8))
CREATE History$$001(
  Crs-taken-tuple-recency = Int(8))
```

Trace and History Relations for the Sample Rule
Figure 5

3.2. The RPL Interpreter

The heart of the RPL 1.0 interpreter is the recognize/act cycle. This is the mechanism by which RPL: (1) determines which rules are eligible to fire (determines which rules have instantiations), (2) chooses a set of independent rules to fire (performs conflict resolution), and (3) fires the rules in the group (executes the RHS actions of the rule using the tuples in the instantiation set). The RPL 1.0 recognize/act cycle is comprised of the five steps listed in Figure 6 below. The RPL interpreter involves two distinct levels of operation. There is the level normally associated with production system languages and also a meta-level to control the interpreter. The meta-level interpreter: (1) performs error handling, (2) collects software metrics, and (3) performs conflict resolution. Thus, Step 3 consists of the execution of the meta-level interpreter. In the remainder of this section, each step of the interpreter is described in more detail.

1). Remove invalid instantiations from the conflict set.
2). Generate new instantiations and add them to the conflict set.
3). Perform conflict resolution.
4). Fire each rule in the selected group.
5). Repeat the process starting at step 1.

Steps in the RPL 1.0 Recognize/Act Cycle
Figure 6

STEP 1: At the beginning of each cycle the interpreter removes all instantiations in the conflict set that reference base relations updated by rule firings of the previous cycle.

STEP 2: The interpreter then issues the LHS query for rules that address base relations updated during the previous cycle. Any non-empty answer set returned by a query constitutes an instantiation set for a rule and is added to the conflict set.

STEP 3: The first component of conflict resolution identifies sets of rules that can fire in parallel. Two rules X and Y are ***independent*** if the LHS of rule X references no base relation updated by a RHS action of rule Y. A ***firing group*** is a set of rules that are independent and they are recorded in the Fire-group relation. Firing rules within only one fire group ensures serializable execution. During the instantiation process of Step 2, the interpreter collects data concerning the attributes of the instantiation set such as tuple count and highest recency. This data, stored in the Fire-group-data relation, is used in conjunction with the user supplied conflict resolution strategy stored in the CR-strategy relation to choose one of the firing-groups for execution. Such meta-level rules are discussed in Section 4.

STEP 4: The RHS actions of each rule in the chosen firing-group are performed on each instantiation in the instantiation set for the rule. The number of answer tuples fired for each rule is controlled by the rule quantifier specified by the user. During firing, as each tuple of the instantiation set is processed, it is added to both the history and trace relations and removed from the instantiation set.

STEP 5: The recognize/act cycle represented by Steps 1 through 4 above is repeated until either Step 3 produces an empty conflict set or the user explicitly stops the system by issuing a "halt" command as a RHS action.

4. A Self-controlling Interpreter

In an effort to improve the performance of expert systems, there is strong interest in controlling the inferencing process [Cheng85, Friedman85]. Various aspects of this problem include: (1) controlling the conflict resolution strategy, (2) partitioning the knowledge (e.g. the rules) into logically related groups, and (3) meta-level reasoning. Within the database community, current standardization efforts [Mark85] are exploring the utility of a self-describing data model to support more powerful database systems as well as a more friendly user interface.

As described in Section 3.1 above, the information that describes the activity of the interpreter, including any errors that occur, is maintained in relations in working memory. This approach supports a self-controlling interpreter in a natural way. Specifically, the conflict resolution strategy, the error handling routines, the tracing and debugging routines, and requests for software metrics are all expressed as RPL rules. Thus, the expressive power of a relationally complete query language is available to control the interpreter. The use of object identifiers, as well as recency numbers, to uniquely identify tuples in working memory can be used in concert with a self-describing schema to provide the flexibility normally associated with a production system [Delcambre87a]. In this section, interpreter directives for conflict resolution, error handling, tracing, and software metrics are discussed in the paragraphs below.

4.1. Controlling the Conflict Resolution Process

The information relevant to the process of conflict resolution consists of both static and dynamic data. Static data such as the relations that are accessed or updated by a rule or the complexity of the LHS can be determined by the preprocessor. Dynamic information includes such things as the number of tuples appearing in the instantiation set for each rule in each cycle of the interpreter. When a recency tag is associated with the objects in working memory, the highest or average recency of tuples that participate in the answer set may be used by a conflict resolution strategy.

Figure 4 illustrates both static and dynamic data relevant to conflict resolution and Figure 7 shows two different conflict resolution strategies that differ only in their sort sequence. They use the ONE quantifier to select only one firing group. The first rule in Figure 7 fires the rule group with the maximum number of instantiations. The final rule performs conflict resolution based on the recency of the data involved in the instantiation set. This is similar to the conflict resolution strategy normally used in a production system. These conflict resolution rules are assigned a priority of 2 so that error handling and other meta-level rules will execute at a higher priority. The RPL meta-level interpreter employs a conflict resolution strategy based solely on static rule priority.

In the next section, the use of RPL rules to capture software metrics is discussed.

4.2. Tracking the Progress of the Program

By using the trace and history relations to store the firing history of the program, a wide variety of software metrics can be implemented as RPL rules. Such rules can be invoked during post-processing (when the program is finished) or at any point during the execution of the program. By placing stop points before or after the firing of individual rules, these software metrics can be displayed to the user upon request. Figure 8 presents an example of such a rule. The Average-firings-per-cycle rule calculates the average number of instantiations fired per rule per cycle since program execution began. The rule shown in Figure 8 calculates the metric for the user-supplied rule shown in Figure 1.

The final self-controlling aspect of RPL concerns the way in which errors are handled.

```
Quantity (2):
FOR ONE
  SELECT Group-id
    FROM Fire-group-data, CR-strategy
    WHERE Goal = 'Quantity'
      ORDER BY Tuple-cnt DESC
DO
  call Fire-instantiation (Group-id);
END
```

```
Recency (2):
FOR ONE
  SELECT Group-id
    FROM Fire-group-data, CR-strategy
    WHERE Goal = 'Recency'
      ORDER BY Recency DESC
DO
  call Fire-instantiation (Group-id);
END
```

RPL Conflict Resolution Rules
Figure 7

```
Average-firings-per-cycle (1):
FOR ONE  SELECT AVG(Count(*))
         FROM Trace$$001
         GROUP BY Firing-cycle
DO
  WRITE ('Rule 1', AVG(Count(*)));
END
```

A RPL Software Metrics Rule
Figure 8

4.3. Handling Errors

During the execution of a RPL rule (either LHS or RHS), errors may occur. Although some errors can be prevented by syntax checking performed by the preprocessor, it is still possible that a rule might attempt to insert an object that is already in the database or the DBMS might encounter some error. To simplify the processing of individual rules, all such errors are automatically recorded in an error relation by the RPL interpreter as shown in Figure 4. RPL rules can be used to specify error handling routines, based on the intent of the user and/or the severity of the error. Figure 9 shows two different error handling rules. The first halts the production system completely whenever an error has been encountered. The second error handling rule in Figure 9 simply prints out the error message and continues processing. These two rules represent two extremes. The use of RPL rules to specify error handling allows the user to control the interpreter as he wishes.

The rules presented throughout this section are merely suggestions for the types of interpreter directives that might be useful in an application. All of these rules are meta-level rules in that they reference or manipulate meta-data (i.e., data that describes the working memory and/or the rules) and they execute at the

```
Halt-on-error (1):
FOR ALL  SELECT (*) FROM Error
DO  Halt;
END
```

```
Print-error (1):
FOR ALL
  SELECT (*)
  FROM Error E, Trace$$001 T
  WHERE E.Firing-cycle = T.Firing-cycle
    AND E.Firing-number = T.Firing-number
DO
  WRITE('Error', E.Error-code);
END
```

RPL Error Handling Rules
Figure 9

meta-level. In general, RPL provides the capability to monitor the progress of the inferencing process through software metric rules and then dynamically alter the conflict resolution process. The result is a flexible, two-level, self-controlling inference system that can support sophisticated meta-level reasoning.

5. Automatic Modification of a User-Supplied RPL Program

In order for the RPL interpreter to work correctly and efficiently, the user supplied rules and schemes must be augmented to support the history processing, trace processing, etc. These modifications can be done automatically during the preprocessing step. The modification of both user-supplied rules and data structures is described briefly in this section. Figure 10 shows a user-supplied relation and Figure 11 shows the user-supplied rule from Figure 1 as they appear after modification by the RPL preprocessor. The modifications are shown in bold type. Schema modification consists of the addition of tuple-id and tuple-recency attributes to every user-supplied relation.

The first step in the processing of a user-supplied rule is the identification of the instantiation. Once the appropriate tuple-ids and values have been identified, the preprocessor automatically generates schema DDL for both the history and trace relations for the rule. Secondly, the tuple-id and tuple-recency attributes are automatically appended to the attribute list. The LHS query is also modified so that the appropriate history

```
CREATE Student (
  Student-tuple-id = Int(8),
  Student-tuple-recency = Int(8),
  Stud-id = Int(9),
  Stud-name = Text(25),
```

A Modified User-supplied Relation
Figure 10

relation is subtracted from the answer to the query to prevent the re-firing of instantiations. Next the RHS actions for delete and modify are transformed to use the appropriate tuple-id for the identification of the object to be updated. Finally, two additional RHS actions are added to the rule to insert a trace tuple and history tuple appropriately.

6. Evaluation, Conclusions, and Future Work

The RPL 1.0 interpreter has been implemented in C on a Sun3 workstation using Eric procedures to interact with a SunUNIFY DBMS. The prototype has been tested using several small programs. A non-trivial expert system for awarding financial aid for a graduate program is concurrently being developed in YAPS [Allen83], RPL, and Prolog. This will provide the framework for an in-depth evaluation of the RPL approach.

The RPL 1.0 prototype demonstrates the feasibility of the approach and provides an experimental vehicle for the execution of RPL programs. Current plans include performance evaluation of the RPL 1.0 prototype. For problems that can exploit the set-based processing of RPL, the performance may be competitive with memory-resident production systems. Additionally, RPL 1.0 is interesting in its own right because it supports investigation into both meta-level reasoning and parallelism in a production system.

```
Eliminate-duplicates:
FOR ALL
 SELECT C.Stud-id, C.Crs-id,
      , C.Crs-taken-tuple-id
      , C.Crs-taken-tuple-recency
 FROM Crs-taken C, Crs-taken T
 WHERE  C.Stud-id = T.Stud-id
   AND C.Crs-id = T.Crs-id
   AND C.Grade <= T.Grade
   AND C.Sem-taken < T.Sem-taken
   AND NOT <Crs-taken-tuple-recency> =
        SELECT *
        FROM Hist$$001
DO
 DELETE Crs-taken
 WHERE Crs-taken-tuple-id
        = C.Crs-taken-tuple-id;
 INSERT INTO
       Hist$$001(C.Crs-taken-tuple-recency);
 INSERT INTO
    Trace$$001(Firing-cycle, Firing-number,
          C.Crs-taken-tuple-id,
          C.Crs-taken-tuple-recency);
END
```

A Modified User-supplied Rule
Figure 11

In a larger sense, the definition of the RPL language provides the key to many fundamental problems that must be solved by a large-scale KBMS. RPL 2.0 will address the performance issues associated with a disk-resident DBMS by implementing RPL in conjunction with a main memory database system [Bitton86]. The challenge for RPL 2.0 is finding the appropriate blend of conventional production system pattern matching and relational query processing. Finally, RPL 3.0 will provide a two-level database system with an outer level comprised of conventional (relational) databases and an inner level consisting of a memory-resident RPL 2.0 interpreter. The challenge for RPL 3.0 is the automatic migration of data between the two levels.

References

[Allen83] Allen, E.M., "Yaps: Yet Another Production System", CS TR-1146, Maryland Artificial Intelligence Group. University of Maryland, Department of Computer Science, College Park, MD, Dec. 1983.

[Bitton86] Bitton, F., "The Effect of Large Main Memory on Database Systems", *in Proc. ACM SIGMOD '86 International Conference on Management of Data*, Zaniolo C. (ed.), Washington D.C., May 1986.

[Brodie86] Brodie, M.L. and Mylopoulos, J. (eds.), *On Knowledge Base Management Systems*, Springer-Verlag, New York, NY, 1986.

[Brownston85] Brownston, L., Farrell, R., Kant, E., and Martin, N., *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, MA, 1985.

[Cheng85] Cheng, Y., and Fu, K.S., "Conceptual Clustering in Knowledge Organization", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAM1/7, No. 5, Sept. 1985.

[Codd79] Codd, E.F., "Extending the Database Relational Model to Capture More Meaning", *ACM TODS*, Vol. 4, No. 4, Dec. 1979.

[Date86] Date, C.J., *An Introduction to Database Systems, Volume I, 4th edition*, Addison-Wesley, Reading, MA, 1986.

[Delcambre87a] Delcambre, L.M.L., and Etheredge, J., "The Relational Production Language: A Production Language for Relational Databases", in *Proc. of the 2nd Intl. Conf. on Expert Database Systems*, Tysons Corner, VA, April 1988.

[Delcambre87b] Delcambre, L.M.L., and Etheredge, J., "A Three Phase Prototype for the RPL Expert Database System", Working paper, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA., Oct. 1987.

[Forgy82] Forgy, C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", *Artificial Intelligence*, Vol. 19, No. 1, Sept. 1982.

[Friedman85] Friedman, L., "Controlling production firing: The FCL language", *in Proc. of the Ninth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1985.

[Maier83] Maier, D., *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

[Mark85] Mark, L. and Roussopoulos, N., "The New Database Architecture Framework - A Progress Report", *in INFORMATION SYSTEMS: Theoretical and Formal Aspects*, Sernadas, J., Bubenko, Jr., and Olive, A., (eds.), Elsevier Science Publishers B.V. (North-Holland), 1985.

[Stonebraker86] Stonebraker, M., "Triggers and Inference in Database Systems", in [Brodie86].

[Sun86] "SunUNIFY Overview", *in SUN Microsystems, Part No. 800-1566-10, Revision: A of 19*, Dec. 1986.

[Ullman82] Ullman, J.D., *Principles of Database Systems, 2nd edition*, Computer Science Press, Rockville, MD, 1982.