ON THE USE OF ACTIVE DATABASE CONCEPTS TO SUPPORT VERY LARGE PRODUCTION SYSTEMS

By

RANDALL J. BLANCO–MORA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1991

ACKNOWLEDGEMENTS

I would first like to thank the chairman of my committee, Dr. Sharma Chakravarthy, for his guidance and support throughout this project.

I would also like to thank Dr. Manuel Bermudez for his valuable comments and his support in this thesis and during my entire stay in the University of Florida. Thanks are extended to him not only as an excellent professor, but as a great friend.

Thanks also go to Dr. Richard Smith for his comments and his guidance in this thesis.

I would also like to mention Dr. Richard Newman–Wolfe, Sharon Grant, Jeffie Woodham, Denise Bouton, John Weckesser, Javier Cordova, Marlene Hughes, and Ligia Bermudez for their support during my graduate studies.

Finally, thanks go to my family and friends for the moral support they have always given me.

Gracias a todos y que Dios los bendiga. (Thanks to all and God bless you.)

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ON THE USE OF ACTIVE DATABASE CONCEPTS TO SUPPORT
VERY LARGE PRODUCTION SYSTEMS

By

Randall J. Blanco–Mora

May 1991

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Sciences

Expert Systems are extensively used for applications that require heuristic-based algorithms using domain specific knowledge. OPS5 is a representative production rule system that implements the match–recognize–act cycle. In this thesis, we describe how an OPS5 system can be translated into a program on an Active Database System. This translation makes it possible to combine the features of production rule systems with the benefits of database systems. This approach will enable the support of very large production systems in a database environment. It will also allow multiple production systems to share a common database. Some benefits of sharing the database include concurrency control, and the verification of consistency of shared data.

CHAPTER 1
INTRODUCTION

The need for capturing the *semantics* of real–life applications as precisely as possible has led to the development of a variety of mechanisms for knowledge representation. The fields of Artificial Intelligence (AI) and Database Management Systems (DBMSs) have addressed this problem with different perspectives.

Artificial Intelligence technology has developed many different knowledge–representation techniques, including semantic nets, logic rules [Colm73] [Cloc81], situation–action rules [Forgy77], procedures, and frames [Mins75]. AI systems also include inference engines, or interpreters, to process that knowledge.

In AI we also have Production Systems (PSs), that use IF–THEN rules as the basic structure to store knowledge. These IF–THEN rules are called production rules, and they include a specific action(s) whenever a given condition(s) becomes true. Most PSs have been implemented as main memory systems — i.e., all rules and data reside in main memory. This limits the size of the set of rules, or *rule base*, as well as the data accessed and modified by an application program.

One of the most popular PSs is called OPS5 [Forgy77]. In OPS5, rules are compiled into a discrimination network, called an RETE network, which is maintained and executed in main memory. In OPS5, as in other production system applications, there is no sharing of data. Each application has its own set of rules and associated data and is executed independently. As a result, shared data must be replicated and maintained separately. Finally, changes to the rule set requires recompilation of the entire set of rules.

DBMSs have traditionally provided a different set of mechanisms for capturing the semantics of an application. In the DBMS field there are several data models oriented towards capturing the *extensional knowledge* rather than the *intensional knowledge*. Extensional knowledge refers to the *stored data* (specific facts about individual objects and their relationships) and the *metadata* (schemas that describe general facts about classes of objects). Intensional knowledge is a *collection of constraints* for deriving new data from the extensional knowledge.

In existing DBMSs the closest concept to a production rule is a *view definition* in the query language, which is usually of the *first–order* — i.e., incapable of enbodying recursion.

Rules provide a uniform mechanism for capturing extensional and intensional knowledge. Hence, smooth incorporation of rules, as a knowledge–representation mechanism to augment the data model, provides a powerful integrated knowledge model that will be capable of capturing the semantics of a wider class of applications.

Incorporating rules into the knowledge model and treating them as first–class objects — in contrast to metadata — provides a modular declarative way of specifying knowledge. Rules provide an elegant way of encoding knowledge about the application environment and a mechanism for specifying exceptions by associating different rules with different objects. Rules also simplify the specification of an algorithm by separating its static portion (which is data independent) from its dynamic portion (which is likely to change). Rule–based query optimizers [Frey87] [Grae87] are concrete examples of that separation.

Modeling the behavior of a production rule system in a DBMS leads to a number of benefits. There is no need to restrict the size of the rule base as efficient secondary storage access is used for accessing data as well as rules. Sharing of both rules and data is possible as well as better consistency maintenance. Finaly, optimization of

rules can be accomplished by incorporating well–known techniques that are already used in database systems.

Active Databases (ADBs) [Chakr89] [Ston87] [Wido90] combine the advantages of PS with advanced database techniques by providing *triggers*. A trigger is a rule–like structure of the following general form: *Event–Condition–Action*. The condition is evaluated when a specified event occurs, and the action is executed if the condition evaluates to true.

Our work addresses the problem of translating a given OPS5 program, which specifies a collection of production rules that operate upon non–persistent data, into a set of relations and triggers in an ADB, which operates upon persistent data.

This thesis is structured as follows. Chapter 2 gives the basic concepts of Production Systems (PSs) and Active Databases (ADBs). In addition, a description of OPS5 and the RETE match algorithm is given. Chapter 3 summarizes some related work regarding the combination of PS and ADB techniques. In Chapter 4 and Chapter 5 we give a method of translating an OPS5 program into an ADB environment. Chapter 6 presents the conclusions and possible extensions for this translation.

CHAPTER 2
BACKGROUND

Data handling and management is a vital issue for many computer systems, particularly in applications in which the data is very specific and well defined.

As the data becomes more specific, it requires more constraints in order to be accurate. Applications in which well defined data is crucial can be found in military control systems, medical diagnostic systems, airport traffic control systems, and so forth [Chak90]. For instance, in an airport traffic control system, variables — such as runway conditions, lighting, weather conditions, and the number of planes leaving and arriving at the airport — should be checked. The decisions of the airport controller should be based on these conditions and other factors that may be checked using some active devices that are monitored by a computer. Such devices should handle data that changes for each situation, as well as restrictions on that data. Both data and restrictions must be evaluated before taking any action in the main control.

These data oriented applications not only manage data, but also manage the restrictions or constraints upon the data. They should provide tools that allow the modification of data and its restrictions in an efficient way.

The fields of Artificial Intelligence and Databases have been addressing the problems of the implementation of such systems. The integration of both fields to produce a functional integrated environment has been a goal for many people [Kers86] [Kers87].

2.1 Production Systems — an Artificial Intelligence Solution

Research on *production systems* (PS) originated in the artificial intelligence and logic programming communities. Production systems are the basis of many *expert systems* [McDe81] [Grie84]. The growth of these systems has made necessary the search for more efficient ways to execute production system programs.

A production system [Rych76] contains two elements:

- a set of *production rules*, that form the *production memory*, and
- a set of *facts*, that form the *working memory*.

Each production rule has two sections, the *left–hand side* (LHS) and the *right–hand side* (RHS). The LHS contains a set of *conditional elements* (CEs) — pattern elements — that are matched against the working memory. A conditional element is a *positive conditional element* if it tests the *existence* of working elements satisfying its condition. A conditional element is a *negated conditional element* if it tests the *non–existence* of working elements satisfying its condition. The RHS contains a set of *actions* that will be executed when the working memory contains a set of facts that matches all the conditional elements given in the LHS.

The operation of a production system can be summarized in the following cycle:

- Match. For each rule, compare the LHS against the current working memory. Each subset of working memory elements satisfying a rule's LHS is called an *instantiation*. An instantiation indicates which production rule(s) is (are) satisfied by it. All current instantiations form the *conflict set*.
- Select. From the conflict set, choose a subset of instantiations according to some predefined criteria.

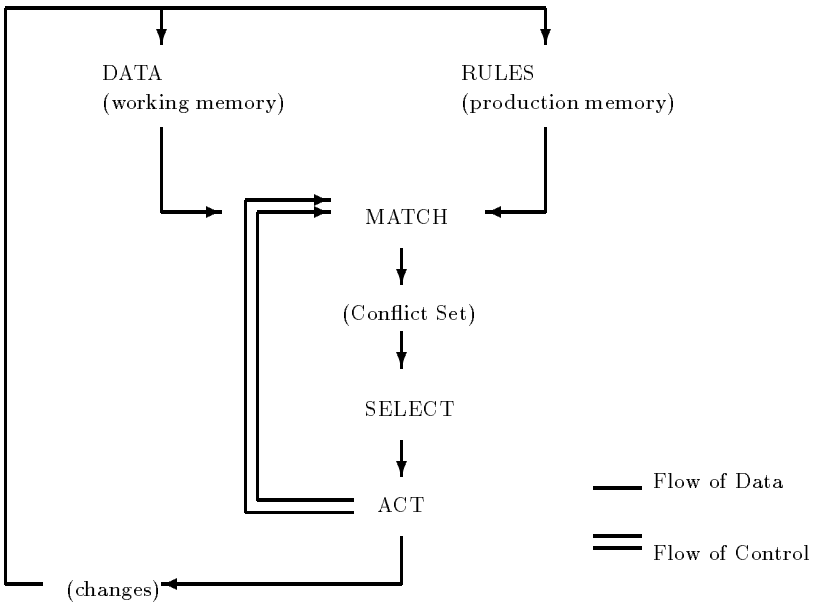

Figure 2.1. Architecture of a Production System

- Act. Execute the actions in the RHS of the rules indicated by the selected instantiations.

The architecture of a production system is shown in figure 2.1.

Some of the conflict–resolution criteria used in the Select phase are: [Barr81]

- choose the *first* rule that matches the working memory,
- choose the *highest priority* rule, according to some priority scheme,
- choose the *most specific* rule,
- choose the rule that refers to the element *most recently* added to the working memory,
- choose a *new* rule,
- choose an *arbitrary* rule.

The development of algorithms that match speciffic patterns (LHSs of rules) against data (working memory elements) has been under research in the Production Systems field. The performance of PSs depends on the complexity and degree of efficiency of such algorithms.

Some methods have been identified in order to gain efficiency in the algorithms of production systems: [McDem78] [Mira87]

- *Condition Membership* provides knowledge about the possible satisfaction of each condition element. Associated with each condition element in the production system is a running count indicating the number of working memory elements partially matching the condition element. A match algorithm that uses condition membership may ignore those rules that are *not active*. A rule is *active* when each of its positive condition elements is partially satisfied.

- *Memory Support* provides knowledge about which working memory elements partially satisfy each condition element. An indexing scheme indicates precisely which subset of working memory partially matches each condition element.

- *Condition Relationship* provides knowledge about the interaction of condition elements within a rule, and the partial satisfaction of rules.

- *Conflict Set Support* is explicitly retained across production system cycles. By doing so, it is possible to limit the search for new instantiations to those instantiations that contain newly asserted working memory elements.

Some of the advantages of the PS as a model of computation are the following: [Barr81] [Brow85]

- *modularity*, a rule is a modular element itself;

- *uniformity*, knowledge must be coded into a rule form;
- *naturalness*, human experts explain their expertise in rule terms;
- *expressibility*, production rules express basic symbol–processing acts;
- *simplicity of control*, a simple inference engine is needed to control the execution;
- *modifiability*, because data and production rules are stored in a separate, nearly independent units, production rules can be added with very few side effects; and
- *parallelism*, portions of the systems often can execute independently.

Still, there are some limitations in PS that need to be resolved:

- Production systems do not perform well with large number of rules.
- Production systems do not share data with other production systems.
- Production systems are main–memory based systems.
- Production systems compile their sets of rules before execution, preventing a rule from changing without re–compilation (i.e., no incremental changes to a rule can be made during execution).

One of the most popular production system programming languages is called OPS5 [Forg79]. Our next section describes this programming language.

2.1.1 OPS5 — a Production System Programming Language

OPS5 is a programming language designed to develop production rule systems. Its flexibility and modularity have been factors that made OPS5 a standard for the development of production rule systems applications.

OPS5 uses the Match–Select–Act cycle described earlier for production systems.

An OPS5 program has two sections: the *declaration section* — which contains LITERALIZE and VECTOR–ATTRIBUTE declaratives — and the *production section* — which contains P (for "production") declaratives.

The following description of these OPS5 declaratives illustrates the way an algorithm is processed in an OPS5 program. This description is not the OPS5 Grammar syntax, but it is based upon it.

2.1.2 The LITERALIZE and the VECTOR–ATTRIBUTE declaratives

The LITERALIZE declarative defines a *class* — or a relation — to hold working memory elements — data. The attributes are optional. The declarative has no attributes when it is defining a class that will contain *single facts* — or *single data*.

The general form is

(literalize relation [attribute1 ... attributeN])

The VECTOR–ATTRIBUTE declarative specifies the attributes that will have multiple–values. OPS5 allows only one vector–attribute for each class.

Its general form is

(vector–attribute attribute1 [attribute2 ... attributeN])

For example, the following declaratives create two classes called KID and ITEM, with the vector–attributes *contents* and *friends*, respectively:

```
(literalize KID
      name
      age
      country
      friends
)
```

```
(literalize ITEM
      address
      label
      type
      contents
)

(vector–attribute contents friends)
```

2.1.3 The P declarative

The P declarative defines a production rule. It is the most complex declarative in its structure as well as in its semantics. This declarative can be summarized as follows:

```
(p production
    conditional element1 /* Must be a Positive Conditional Element */
    conditional element2
            ⋮
    conditional elementN
 →
    action1
    action2
       ⋮
    actionM
)
```

LHS — a Set of Conditional Elements

A *conditional element* has one of three structures:

1. [–] (relation [↑attribute1 value–var1
                  ↑attribute2 value–var2
                           ⋮
                  ↑attributeN value–varN])

2. [–] ([(] relation [↑attribute1 value–var1
   ↑attribute2 value–var2
   ⋮
   ↑attributeN value–varN])
   [element–variable)]

3. [–] [(element–variable] (relation [↑attribute1 value–var1
   ↑attribute2 value–var2
   ⋮
   ↑attributeN value–varN])
   [)]

Notice that the symbol ↑ is the OPS5 symbol that distinguishes attributes from values, and the OPS5 symbol → separates the LHS from the RHS.

A *value–var* is either a *value* — such as HI, 5, ADAM — or a *var*iable — such as <x>, <kid>, <student>.

An *element–variable* is like a label that will refer to the working memory elements that match the conditional element where it is defined.

For example, in the following production rule,

```
(p 1:: refering to American kids and toys
   ((Kid   ↑name <name>   ↑country USA)   <usakid>)
   ((Item   ↑type TOY)   <toy>)
 →
   action
)
```

there are two element–variables — *usakid* and *toy*. The element–variable *usakid* refers to all the working memory elements in the class KID with the value USA in their attribute *country*. The element–variable *toy* refers to all the working memory elements in the class ITEM with the value TOY in their attribute *type*.

RHS — a Set of Actions

In our algorithm, we cover three OPS5 actions: MAKE, REMOVE, and MODIFY.

The MAKE action inserts a new working memory element into the working memory. The REMOVE action deletes a working memory element from the working memory. And finally, the MODIFY action updates a working memory element from the working memory.

In that context, an *action* has one of the five following structures:

1. (make relation [↑attribute1 value–var1
   ↑attribute2 value–var2
   ⋮
   ↑attributeN value–varN])
2. (remove N) /* where N is any integer */
3. (remove element–variable)
4. (modify N [↑attribute1 value–var1
   ↑attribute2 value–var2
   ⋮
   ↑attributeN value–varN])
5. (modify element–variable [↑attribute1 value–var1
   ↑attribute2 value–var2
   ⋮
   ↑attributeN value–varN])

2.1.4 Performance in OPS5

OPS5 uses the RETE algorithm, described in the next section, in order to match the LHSs of the production rules against the working memory. This makes OPS5 a main–memory based system, and therefore there is a limitation on the number of production rules that it is able to manage. This limitation restricts its scope of operation to small production systems, i.e. OPS5 is unable to handle very large database applications.

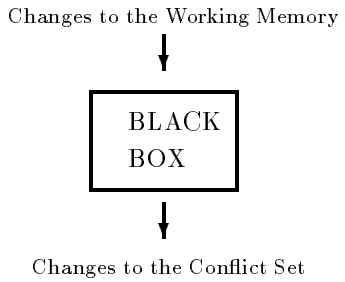

Figure 2.2. RETE viewed as a black box

While OPS5 has been a good solution for small applications, there is no hope for data intensive–very large production systems.

2.1.5 The RETE Algorithm — a Match Algorithm

The RETE match algorithm was developed by Charles Forgy [Forg79] [Forg82] and is the most commonly used match algorithm for implementing production systems.

The algorithm can be viewed as a black box that has one input and one output (see figure 2.2). The box receives the changes that are made to the working memory, and determines the changes that must be made in the conflict set to maintain its consistency.

The RETE algorithm compiles the left–hand sides of the production rules into a discrimination network. Each left–hand side tests different kinds of features of the working memory elements. These features can be divided in two classes: the *intra–element features*, and the *inter–element features*. [Forg82]

The intra–element features refer to features involving *only one* working memory element. For example, the conditional element of the following rule tests only for intra–element features:

```
(p looking for CIS–students in Databases
    (Student    ↑name <name>   ↑area DB    ↑major CIS)
 →
    action
)
```

The inter–element features refer to features involving more than one working memory element. Every time a variable appears in *more than one* conditional element, there is a test for an inter–element feature. For example, the two conditional elements of the following rule are used to test an inter–element feature:

```
(p warning to working students with more than 9 credit hours
    ((Student    ↑name <name>   ↑phone <phone>   ↑credit {> 9})   <student>)
    (Employee    ↑name <name>)
 →
    action
)
```

RETE's discrimination network contains four types of nodes:

- The *select node*. This node tests for intra–element features in the working memory elements. It has one input and one or more output nodes — the number of output nodes depends on the number of conditional elements in the production rules testing the same feature.

- The *α–memory node*. This node contains all the working memory elements selected with a select node. For each selection node, there is an α–memory node.

- The *join node*. This node tests for inter–element features in the working memory elements. It has two input nodes and one or more output nodes. The two input nodes come from the two conditional elements testing for the same inter–element feature. If there is an inter–element feature being tested in n

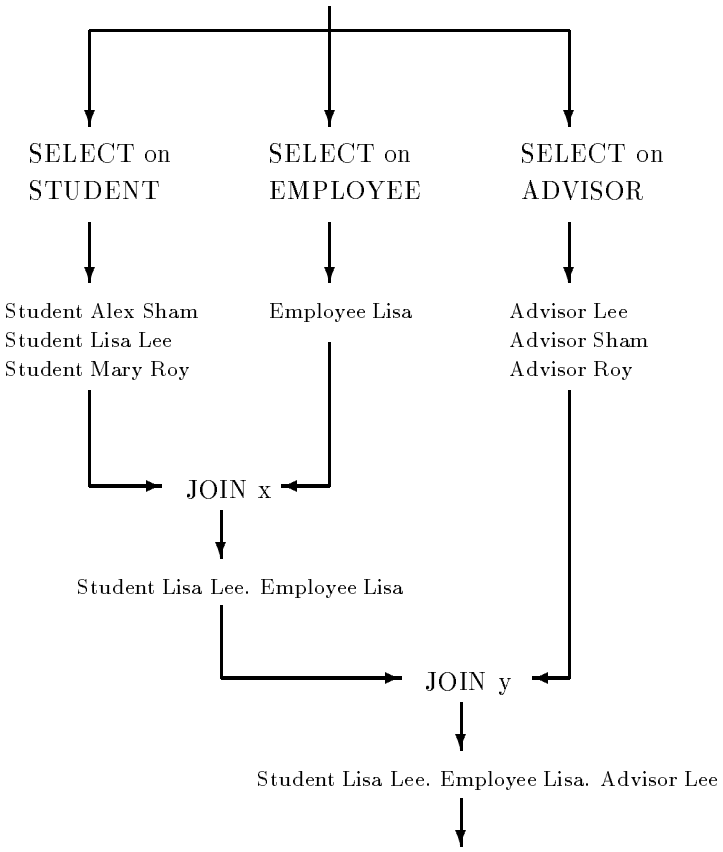

Figure 2.3. RETE Match Network for the rule *Example*

different conditional elements, then there will be $n-1$ join nodes to compute the result of that feature. The number of output nodes depends on the number of conditional elements in the production rules testing the same feature.

- The *β–node*. This node contains the working memory elements that are combined in order to compute the result of an inter–element feature.

The following rule — called *Example* — tests for intra, and inter–element features. The rule executes the action *action–whatever* to *all the working–students with an advisor*:

```
(p Example
    (Student    ↑name <x>   ↑advisor <y>)
    (Employee   ↑name <x>)
    (Advisor    ↑name <y>)
 →
    action–whatever
)
```

Assume that the working memory contains the following:

```
(Student    ↑name ALEX    ↑advisor SHAM)
(Student    ↑name LISA    ↑advisor LEE)
(Student    ↑name MARY    ↑advisor ROY)
(Advisor    ↑name LEE)
(Advisor    ↑name SHAM)
(Advisor    ↑name ROY)
(Employee   ↑name LISA)
```

The network produced by RETE for this rule is shown in figure 2.3. In general, the RETE network has two sections:

- a *selection network*, which selects and stores the working memory elements that match with the conditional elements, and
- a *join network*, which joins the outputs of the selection network keeping the bindings consistent — the results of these joins are also stored.

One advantage of the index used in the RETE algorithm — the network itself — is that it avoids an iteration over the set of production rules.

One of the major disadvantages of the RETE algorithm is that it is highly sequential, which makes the algorithm difficult to parallelize. Also, RETE is a main–memory based algorithm, which means that the global working memory is not persistent after a session. But probably, the major disadvantage of RETE is that it mixes data and production rules, which makes impossible the concept of sharing data.

2.2 Active Databases — a Database Solution

The experience of Artificial Intelligence has clearly shown the value of rule–based programming tools [Hans89]. Database researchers have also recognized the potential value of rule–based systems [Eswa76] [Bune79].

Research on database rule systems has focused primarily on two areas: [Hans89]

- deductive databases (e.g. making PROLOG work over a large database), and
- the addition of rule–based programming features to database systems.

The goal of the first area is essentially to couple a Database Management System (DBMS) with a deductive system — such as PROLOG — so recursive retrieval of data can be processed efficiently over large databases.

The goal of the second area is to construct *active database systems.*

The Need for Active Databases

Traditional DBMS are *passive*, i.e., the execution of queries are based upon the request of a user or an application program. But there are applications — such as process control, power generation/distribution networks, battle management, and so forth [Chak90] — which require time–constrained applications. For these systems, the traditional approach is not suitable.

In a time–constrained application, it is important to monitor conditions defined on states of the database, and when a condition occurs, to execute specific actions, most likely subject to some timing constraints [Chakr89].

Some time–constrained applications have been solved using passive databases. There are two solutions: [Chak90]

- Write an application that *periodically queries* the passive database to determine if the situation being monitored has occurred. However, if the application

queries too slowly, there is a risk of missing the monitored condition. If it queries too frequently, it is possible to flood the database with queries that return an empty answer.

- Augment each program that update the database to check the monitored condition. Unfortunately, software modularity is compromised: any modification to the monitored condition will require the modification of all the programs that updates the database.

An Active Database (ADB) will provide *timely response* and *modularity*, since it contains special *constant query–like* entities called *triggers*.

Triggers — the active element in the ADB

A trigger is the active element in an active database. It is like a production rule in a production system, but it is oriented towards database concepts.

In an active database, the working memory classes are stored in relations. Every condition testing for data becomes a query to the database. The element–variable concept of production systems, becomes now like an *alias* to the relation from which it is defined.

A trigger is like a constant query to the database. It checks for certain *database event(s)* in order to evaluate a given *condition(s)*. When the event is *signaled*, the condition is checked. If the condition is *fullfilled* — i.e., the queries return non–empty results — then the *database operations* or actions are executed.

There are many ways to describe a trigger. Here, we present an example of the trigger structure. We feel that it is the most standard structure and that it covers the structures presented in other systems [Chakr89] [Hans89] [Wido90].

The general structure is as follows:

(TRIGGER *trigger–name*
EVENT: *event–list*
CONDITION: *condition*
ACTION: *action–list*
)

The *trigger–name* is an identifier to the trigger itself.

The *event–list* is a logical condition of database events. The logical operators permitted for the event–list are AND and OR. In our algorithm, we allow three types of database events: INSERT, UPDATE, and DELETE.

The *condition* is a logical condition of database queries. The logical operators permitted for the condition are AND and OR.

The *action–list* is a list of database operations. Our algorithm allows three database operations: INSERT, UPDATE, and DELETE.

For example, the following trigger,

```
(trigger p1:: refering to American kids and toys
 EVENT:
   UPDATE(Kid) OR
   UPDATE(Item)
 CONDITION:
   EXISTS (SELECT INTO temp1 *
           FROM Kid
           WHERE ((Kid.country=USA)))
   AND
   EXISTS (SELECT INTO temp2 *
           FROM Item
           WHERE ((Item.type=TOY)))
   ACTION:
     action
)
```

refers to the American kids and the toy items. Notice that the conditional elements are queries to the database. In this trigger, whenever operations UPDATE(Kid) or UPDATE(Item) are signaled, the condition is checked. If *all* the

queries return non–empty results, then the action is executed — i.e., the "triggering" takes place.

In an active database, there is no match algorithm. The checking of the conditions against the data is performed using query optimization and evaluation techniques. This *query engine* is a part of the active database itself.

The query engine constantly checks for the signals of events or the fulfilment of conditions in the pre–defined database triggers.

One advantage of using active databases is that they provide a clear separation between data and control. The set of triggers is an independent entity of the database. It is not mixed with the data stored in the database.

CHAPTER 3
RELATED WORK

3.1 ARIEL

ARIEL is a Database Management System (DBMS) that has a built–in production rule system, which is being implemented with the EXODUS [Care86] database tool package. It uses an extended database–oriented RETE algorithm to match data and production rules [Hans89].

Since ARIEL is implemented with the EXODUS database tool package, properties, such as concurrency control, crash recovery, access methods, and query optimization services, are available in ARIEL [Care86] [Grae87].

The production rules in ARIEL are processed using the *recognize–act cycle* which is also used in OPS5 [Forg81].

The conflict set resolution strategy for ARIEL is a variation of the LEX strategy used in OPS5. ARIEL picks a rule using the following criteria:
after each of the steps, shown below, if there is only one rule still being considered, that rule is scheduled for execution. Otherwise, the set of rules still under consideration is passed to the next step:

- select the rule(s) with the highest priority,
- select the rule(s) most recently awakened,
- select the rule(s) whose condition is the most selective — the selectivity is estimated by the query optimizer at the time the rule is compiled,
- if more than one rule remains, select one arbitrarily.

In ARIEL, there are two alterations made to the RETE algorithm in order to improve its performance: [Hans89]

- The entire network — except for contents of the α–memory and β–memory nodes — is a main–memory data structure. The number of rules in the system is expected to be small enough so that the network will fit in main memory. Rules are a form of intensional data (*schema*), as opposed to extensional data (*contents*).

- A top–level index structure is used after the root node to limit the number of select nodes that must be checked against each token as much as possible.

The rules in ARIEL have the following general form:

DEFINE RULE *rule–name*
[PRIORITY *priority–value*]
[ON *event*]
[IF *condition*]
THEN *action.*

The *rule–name* should be unique for each production rule. The *priority–value* controls the order of production rule execution. The ON clause specifies an event that will trigger the production rule. The following types of events can be specified:

APPEND [TO]*relation–name*,
DELETE [FROM]*relation–name*,
REPLACE [TO] *relation–name* [(*attribute–list*)],
RETRIEVE [FROM]*relation–name* [(*attribute–list*)],
TIME = *time–list*, and
EVERY [*n*]*time–unit*
    [STARTING *time–value*]
    [ENDING *time–value*]

A *time–value* is of the form YY:MM:DD:HH:MM:SS using a 24 hour clock. A *time–list* is a comma–separated list of one or more *time–values.*

The *condition* after the IF clause has the form

qualification [FROM *from–list*].

The *qualification* is like a Where clause in a regular database query. The THEN part of the rule contains the action to be executed when the rule fires. ARIEL allows a *compound command* which is a DO ...END block of actions.

ON conditions respond to events and occurrences of a particular kind of database command; IF conditions match patterns in the data. The negated condition is possible in ARIEL with the use of the logical function NOT. A clause of the form

NOT *condition*

can be included in a logical expression in either a rule condition or a qualification clause in a query.

In ARIEL, it is possible to define variables in the condition clause. These variables are global throughout the rule, i.e. tuples in the rule action already satisfy the qualifications of the rule condition. In this way, ARIEL binds the condition and action of a production rule.

ARIEL either maintains a materialized version of the data matching the IF condition, or runs a query just after the ON event occurs to find the matching data [Hans89].

3.2 MOBY

MOBY is a distributed architecture derived from OPS5 [Forg79] designed to support the development of expert database systems in a rule based language. Its primary task is to exploit the potential for concurrency in a *rule–based context.*

MOBY uses a local area network (LAN), which consists of a control unit (CU) connected to a set of processing elements (PEs). The CU is responsible for the control of the recognize–act cycle, including synchronizing and handling communication

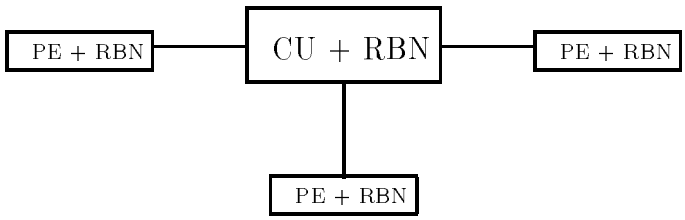

Figure 3.1. LAN structure used in MOBY

between the PEs. Each PE has a local primary and secondary memory. The primary memory of each PE is initialized with a copy of the RETE network which encodes the rulebase (RBN). Portions of the database are placed in the α–memory and β–memory of each PE [Bein87]. Figure 3.1 shows this structure.

Conceptually, there are two types of nodes: the *logical* and the *actual* node. A logical node contains a whole database relation. An actual node contains a horizontal fragment of a relation. Thus, for each logical node, the number of actual nodes is the same as the number of PEs. The objective in partitioning is to balance the processing load. It is possible to have an actual node with no data. This happens either because not much data exists in the logical node, or because the horizontal partition did not work well.

The CU of the network should ensure proper synchronization between actions and it also determines how to partition relations across the PEs. When a new joined — or selected — instance is inserted, the CU determines which PEs should get a copy of the instance and then sends a message to those PEs. Each PE receives the instance and maps the logical node — contained in the message — with an actual node in the machine. When this mapping is done, the instance is joined with existing instances on the node.

MOBY's capabilities include the simultaneous execution of several joins at different parts of the LAN. In the process of performing one incremental join, each newly joined tuple may be incrementally joined with the results from other previous incremental joins [Bein87].

MOBY's performance depends on the horizontal partitioning strategy. In order to have an even distribution of data and a maximum degree of concurrency across the PEs, it is important to choose the right horizontal partition.

3.3 DIPS

Data Intensive Production System (DIPS) uses data structures implemented with database relations to store rule definitions. A matching algorithm runs over these relations — called CONDs — to check when the conditional elements of the production rules are satisfied. The match process creates some partial match data that is stored in the COND database relations. Links between COND relations exist when there is a *join* between the conditional elements of the production rule.

In the match process only a single search over the COND relations is necessary to obtain the instantiated rules for a given cycle. This feature makes the match faster in DIPS than in RETE. Another advantage of the DIPS approach is that the match process is easily parallelizable. The propagation of changes can be performed in parallel to all COND relations, in contrast with the RETE method which is highly sequential [Sell89].

A locking protocol is used in DIPS to produce serializable execution schedules. For example, in a positive condition, two locks must be obtained before the execution of the RHS action(s):

- A READ LOCK is necessary for specific tuples that satisfy the LHS of the rule. This lock will prevent the deletion or the update of these tuples by other transactions.

- A WRITE LOCK must be obtained for specific tuples that are deleted or modified in the RHS of the rule.

In DIPS the matching is fast, but the maintenance is expensive. Also, DIPS consumes more space, but that is the trade off between matching time and space. One advantage of DIPS is its potential for parallelism.

3.4 HiPAC

The High Performance Active DBMS (HiPAC) project involved the implementation of an active database for applications requiring time–constrained data management and processing. HiPAC is an object–oriented DBMS, so all data manipulation is expressed as operations (functions) on objects (entities).

In HiPAC, rules are treated as objects. Thus, rules can be created, modified, or deleted in the same way that other objects are. Also, rules are subject to the same transaction semantics as other data objects: a transaction must obtain a *read lock* on a rule object in order to fire the rule, and a transaction must obtain a *write lock* on a rule object in order to modify, delete, or deactivate the rule. Hence, a rule that is in the process of being fired by one transaction cannot be modified, deleted, or deactivated by another transaction [Chakr89].

There is a rule object class, that has five rule's operations: create, delete, enable, disable, and fire. HiPAC automatically invokes the operation *fire* for all the instances — rules — in this rule object class.

HiPAC's rules can be *related* to other objects and can also have specific *attributes*. These properties are used for grouping rules by context (e.g., all rules with the same

attribute, or all rules related to the same object), thus reducing the scope of any search for rules.

3.4.1 Rules in HiPAC

HiPAC's rules are defined by specifying the following: [Chakr89]

- Rule identifier. A unique entity identifier.
- Event. The event that causes the rule to be fired.
- Condition. The coupling mode (between the triggering transaction and the condition evaluation), and a collection of queries to be evaluated when the rule is fired.
- Action. The coupling mode (between the condition evaluation and the action execution), and an operation to be executed when the condition is satisfied.
- Timing constraints. Deadlines, priorities/urgencies, or value functions.
- Contingency plans. An alternative action to be executed in case the timing constraints cannot be met.
- Attributes. Additional properties of rules (may yield scalar values such as strings and integers, or complex entities that may themselves have attributes).

When the event of a rule has occurred (i.e., the event has been *signaled*), it *fires* the rule. The process of firing a rule includes: [Chakr89]

a. binding the formal arguments in the event definition to the actual arguments in the event signal,

b. evaluation of the rule's condition, and

c. execution of the rule's action (depending on whether the condition was satisfied and subject to whatever concurrency and timing constraints were defined).

For each firing, HiPAC creates a system data structure called a *firing entity*. This data structure contains the rule identifier, the event signal, the bindings, and other relevant system state information.

An *event* has a *signal* operation that is detected by an "event detector", or it is executed by either a user or an application program. The signal operation binds the formal arguments, specified for the event, with the actual arguments. Since the execution of database operations is not instantaneous, it is possible to define two events for each database operation: the *beginning* and *end* of an operation. A similar situation occurs with the transactions, so the same events can be defined for transactions: the *beginning* and *end* of a transaction. HiPAC supports an *abstract event*, which is detected and signaled by users or application programs. This is like an external event for HiPAC. Finally, it is possible to define *composite events*, which are built up using three constructors — disjunction, sequence, and closure.

A condition contains two sections: a coupling mode, and a collection of queries. The condition is satisfied if *all* the queries return non–empty answers. The coupling mode indicates when a condition should be evaluated. There are four possibilities: [Chakr89]

1. *Immediately when the triggering event is signalled*, in which case the execution of the triggering transactions is suspended until the condition — and possibly the action — is executed.

2. *In a deferred mode*, in which case the condition is evaluated at the end of the transaction before the triggering transaction commits.

3. *Detached but causally dependent*, meaning that the condition is evaluated in a separate transaction but after the triggering transaction has committed. If the triggering transaction aborts, then the condition is not evaluated.

4. *Detached but causally independent*, meaning that the condition is evaluated in a separate transaction and the scheduler is free to schedule this transaction independently of the triggering transaction.

An action contains two sections: a coupling mode, and an operation to be executed. The operation can include database operations (retrievals, updates, abstract event types, and other function invocations over arbitrary entity types), or a message to an external application or process. The coupling modes indicate when the operation should be executed. As in the condition coupling mode, there are four possibilites: [Chakr89]

1. *Immediately when the condition is signalled*, in which case the execution of the triggering transactions is suspended until the operation is executed.

2. *Deferred to the end of the transaction in which the condition is signaled.* Note that if the condition evaluation itself was deferred, then the immediate and deferred modes for the action are equivalent.

3. *Detached but causally dependent*, meaning that the operation is executed in a separate transaction but after the triggering transaction has committed. If the triggering transaction aborts, then the operation is not executed.

4. *Detached but causally independent*, meaning that the operation is executed in a separate transaction and the scheduler is free to schedule this operation independently of the triggering transaction.

CHAPTER 4
IMPLEMENTATION ISSUES PRIOR TO THE TRANSLATION

Before presenting the algorithm that translates an OPS5 program into a set of relations and triggers in an ADB it is necessary to introduce some definitions and to discuss some implementation issues.

This chapter is divided in two sections. Section 4.1 introduces a categorization of conditional elements. Section 4.2 gives the implementation details on vector–attributes and single facts.

4.1 Categorization of Conditional Elements

The categorization of conditional elements (CEs) is based upon two features:

a. whether or not the CE is a Positive or a Negated CE, and

b. whether or not the CE contains a *join*, i.e., a variable that appears in another CE at the same production rule.

The first feature refers to the *semantics* of the conditional element. A Positive CE tests for the *existence* of some tuples meeting the condition in a given relation. A Negated CE tests for the *non–existence* of tuples that meet the condition in a given relation.

For example, the following Positive CE,

(R1 ↑attribute1 BOX ↑attribute2 BLUE),

tests whether there exists a tuple in relation R1 with *attribute1* equals to BOX, and *attribute2* equal to BLUE. It is a Positive CE since it tests the existence of a BLUE BOX.

An example of a Negated CE is the following,

−(R1 ↑attribute1 BOX).

This Negated CE tests for the non–existence of a BOX in relation R1. Notice that both CEs cannot be satisfied at the same time.

The second feature of our categorization refers to the *degree of dependency* that a CE has within a rule. A CE is *independent* if it has no variables that appear in any other CE in the same production rule; and it is *dependent* otherwise.

With that in mind, we present examples of the basic four categories derived from these two features.

For the examples given below, assume the following OPS5 declaration:

```
(literalize Student
      name
      advisor
      area
      credit
      phone
)

(literalize Employee
      name
      employer
)

(literalize Advisor
      name
      area
      phone
)
```

```
(literalize Note
      to
      from
      phone
      message
)

(literalize Start
)
```

4.1.1 The Independent Positive CE — IPCE

In the IPCE most of the values attached to the attributes are constant values, i.e. the IPCE contains no *joins* with other CEs. When an IPCE contains a variable, it is being used to pass values to the RHS of the production rule.

In the following production rule the CE is an IPCE.

```
(p 2:: Refer any CIS student interested in EXS to Dr. Dell
   ((Student   ↑name <name>   ↑phone <phone>
      ↑area EXS   ↑advisor NA)   <student>)
 →
   (Modify <student>   ↑advisor DR. DELL)
   (Make Note   ↑to DR. DELL   ↑from CIS   ↑phone <phone>
      ↑message <name> INTERESTED IN EXS)
)
```

4.1.2 The Independent Negated CE — INCE

This CE is very similar to the IPCE, but it has the *"minus sign"*, meaning that it is checking for the non-existence of tuples meeting certain criteria.

As in the IPCE, it contains no variables that appears in any other CE. In this CE type, the existence of variables used to pass values to the RHS of the production rule makes no sense. Since it tests the non–existence of tuples, whenever the RHS is executed there are no tuples meeting the condition in the Negated CE.

In the following production rule, the first CE is an IPCE, and the other CEs are INCEs.

```
(p 3:: John is supported by the CIS Department
    (Student    ↑name JOHN)
   −(Employee    ↑name JOHN    ↑employer CIS)
   −(Note    ↑to CHAIRMAN    ↑from CIS–JOB    ↑message JOHN NEEDS A JOB)
  →
    (Make Note    ↑to CHAIRMAN    ↑from CIS–JOB    ↑message JOHN NEEDS A JOB)
)
```

4.1.3 The Dependent Positive CE — DPCE

This type of CE is a Positive CE that has a variable which is present in at least one other Positive CE. This means that there exists a *positive join* between at least two relations. The relations linked are those attached with the CEs that contain the common variable. The join is made through the attributes attached to each variable in each CE.

In the following production rule, both CEs are DPCEs. The join variable is *name*, and the join condition is (*Student.name* = *Employee.name*).

```
(p 4:: Warning to working students with more than 9 credit hours
    ((Student    ↑name <name>    ↑phone <phone>    ↑credit {> 9})    <student>)
    (Employee    ↑name <name>)
  →
    (Make Note    ↑to <name>    ↑from CIS    ↑phone <phone>
        ↑message PLEASE CONTACT CIS OFFICE — CREDIT OVER LOAD)
)
```

4.1.4 The Dependent Negated CE — DNCE

The criteria to determine if a CE is a DNCE is different depending whether or not the CE is a Positive or a Negated CE. If the CE is a Positive CE, then it is a DNCE if it contains at least one variable that appears in a Negated CE. If the CE is a Negated CE, then it is a DNCE if every one of its variables is contained in a Positive CE.

The following production rule reflects a restriction in the CIS Department for non–employed database students. The restriction states: *every non–employed database*

student with an advisor whose area is not Databases, must have a database faculty member on his/her committee.

In this rule all the CEs are DNCEs,

```
(p 6:: Strange restriction for non–employed DB students
    (Student    ↑area DB    ↑name <x>    ↑advisor <y>    ↑phone <phone>)
  –(Employee    ↑name <x>)
  –(Advisor    ↑name <y>    ↑area DB)
  –(Note    ↑to <x>
        ↑message YOUR COMMITTEE MUST HAVE A DB FACULTY MEMBER)
 →
    (Make Note    ↑to <x>    ↑from CIS    ↑phone <phone>
        ↑message YOUR COMMITTEE MUST HAVE A DB FACULTY MEMBER)
)
```

4.2 Implementation Details

This section describes how to implement vector–attributes and single facts. It also explains how to materialize the queries involving vector–attributes.

4.2.1 Implementation of Vector Attributes

A vector–attribute is an attribute capable of having more than one value.

For example,

```
(vector–attribute contents friends)
(literalize ITEM
      address
      label
      type
      contents
)

(literalize KID
      name
      age
      country
      friends
)
```

creates the relations ITEM and KID with the multiple–value attributes called *contents* and *friends*, respectively. The following MAKE statements assign values to these attributes:

```
(Make ITEM    ↑address TEXAS    ↑label DR. REED    ↑type BOX
              ↑contents BOOK PENCIL LAMP CUP CLOCK)
(Make KID     ↑name JOE    ↑age 4    ↑country CANADA
              ↑friends MARK LISA TONY)
```

We present three ways to implement this type of attribute:

- having the vector–attribute as part of the relation,
- having one relation for all vector–attributes,
- having one relation per vector–attribute.

Having the Vector–Attribute as part of the Relation

This is the simplest implementation. It consists of storing all the values of the vector–attribute in the relation where the vector–attribute is defined.

In this case, if there are n values to include in a vector–attribute, then there will be n tuples in the relation.

Using the example below, this approach gives:

ITEM	address	label	type	contents
	TEXAS	DR. REED	BOX	BOOK
	TEXAS	DR. REED	BOX	PENCIL
	TEXAS	DR. REED	BOX	LAMP
	TEXAS	DR. REED	BOX	CUP
	TEXAS	DR. REED	BOX	CLOCK

KID	name	age	country	friends
	JOE	4	CANADA	MARK
	JOE	4	CANADA	LISA
	JOE	4	CANADA	TONY

With this approach there is repetition of data, and consequently waste of memory space. One advantage of this approach is that it reduces the overhead in handling the multiple–value attributes.

Having One Relation for all Vector–Attributes

Another possibility is to define a *unique relation* to store the values for all the vector–attributes in the database.

In order to link these values with the corresponding vector–attribute — and the relation it belongs to — it is necessary to introduce *keys*. A key is an identifier that is unique for each *tuple* in the relation. A tuple is one "row" of the relation.

This approach includes a scheme to generate unique keys for each tuple in the main relations. The approach of having one relation for all vector–attributes generates the following:

ITEM	address	label	type	contents
	TEXAS	DR. REED	BOX	ITEM_2

KID	name	age	country	friends
	JOE	4	CANADA	KID_4

VECTOR	key	value
	ITEM_2	BOOK
	ITEM_2	PENCIL
	ITEM_2	LAMP
	ITEM_2	CUP
	ITEM_2	CLOCK
	KID_4	MARK
	KID_4	LISA
	KID_4	TONY

This solution reduces the space wasted in unnecessary data. However, the approach of combining all the vector–attribute values in one relation is not effective when there are many relations with vector–attributes in their structure.

Also, the key to link the values to the vector–attribute should include the relation where the vector–attribute is defined. This increase the key size.

To solve these problems a third approach is presented.

Having One Relation per Vector–Attribute

This approach creates a new relation for each vector–attribute. The values are linked with a key associated at the relation where the vector–attribute was originally defined.

In the example, this approach yields:

ITEM	address	label	type	contents
	TEXAS	DR. REED	BOX	2

KID	name	age	country	friends
	JOE	4	CANADA	4

VECTOR_ITEM	key	value
	2	BOOK
	2	PENCIL
	2	LAMP
	2	CUP
	2	CLOCK

VECTOR_KID	key	value
	4	MARK
	4	LISA
	4	TONY

We consider this approach to be the best implementation. It reduces the space being utilized to store data, and gives flexibility to handle the multiple values of the vector–attributes.

There is some overhead required to handle different relations and to generate different keys for each of those relations. However, this overhead is a reasonable trade off to reduce the space being wasted in the other approaches.

Notice that in this approach — and also in the previous approach — if a CE involves a vector–attribute, then an extra *join* is added to the query covering that particular CE. The join is made using the key of the vector–attribute. The join is needed to link the values to the rest of the tuple data.

4.2.2 Single Facts

A single fact is a fact with no attributes. It is implemented using a relation with a single attribute — the attribute *dummy*. This attribute is used only as a reference to store data; its name has no semantics.

The only value that *dummy* can take is TRUE, meaning that the single fact exists in the working memory.

The only condition to test in a single fact is whether or not it is present in the working memory. Therefore, any query related with a single fact has a null statement in the WHERE clause, and an asterisk (*) in the SELECT clause.

CHAPTER 5
AN APPROACH TO AN OPS5 TO ADB TRANSLATION

In this chapter, we present an approach to transforming an OPS5 program into a set of relations and triggers in an ADB environment.

The chapter is divided into six sections. Section 5.1 is a general description of the algorithm that performs the translation. Section 5.2 gives an example of an OPS5 program to be the input of the algorithm. Section 5.3 gives the output generated by the algorithm for the input given in Section 5.2. Section 5.4 and Section 5.5 explain Phase 1 and Phase 2 of the algorithm. Finally, section 5.6 gives the algorithm itself.

5.1 General Description of the Algorithm

The algorithm takes as input an OPS5 program, and produces the set of relations and triggers necessary to produce, in an ADB, the same execution model used in the OPS5 program.

There are advantages of having an OPS5 execution model in an ADB. There is a clear separation of *data* and *control*. The data is stored in database relations. The control is stored in triggers. Furthermore, this separation makes possible the execution of different sets of production rules on the same data, which makes possible the concept of shared data among different production systems. Another advantage is that in the ADB the data is persistent, instead of the non–persistent data used in the OPS5 program.

In this translation, a working memory element becomes a "row" — or a *tuple* — in a database relation. In a sense, it is valid to say that a relation is a categorization of working memory elements — those elements that share the same constructor type.

Database queries are used to represent the logical conditions of the production rules. In general, a query is a more specific categorization of working memory elements — those elements that meet the condition specified in the production rule. Each production rule in the OPS5 program is translated into a trigger in the ADB. That trigger contains the queries — OPS5 conditions — as database conditions to execute the actions — database operations. We assume the same conflict set resolution strategies for the OPS5 execution model as for the ADB, i.e., if the conflict set is the same, the order of execution should be the same.

The algorithm has two phases. The first phase defines the relations, and finds the existent dependencies among the production rules. Initial value attributes, terminal attributes, don't care attributes, and cycles are detected in this phase.

The definition of the database queries — and consequently the definition of the triggers — takes place during the second phase.

5.2 The Input — an OPS5 Program

Consider the following OPS5 program:

```
(literalize Student
        name
        advisor
        area
        credit
        phone
)

(literalize Employee
        name
        employer
)

(literalize Advisor
        name
        area
        phone
)

(literalize Note
        to
        from
        phone
        message
)
```

```
(literalize Start
)

(p 1:: Start
     ((Start)   <start>)
  -->
     (Make Student    ↑name JOHN    ↑advisor DR. FRYE    ↑area DB    ↑credit 9    ↑phone 371-4627)
     (Make Student    ↑name MARY    ↑advisor DR. LEE    ↑area DB    ↑credit 9    ↑phone 322-1472)
     (Make Student    ↑name RICHARD    ↑advisor NA    ↑area EXS    ↑credit 12    ↑phone 335-7167)
     (Make Employee    ↑name MARY    ↑employer CIS)
     (Make Advisor    ↑name DR. DELL    ↑area EXS    ↑phone 392-2791)
     (Make Advisor    ↑name DR. LEE    ↑area ARQ    ↑phone 392-2693)
     (Make Advisor    ↑name DR. FRYE    ↑area DB    ↑phone 392-2272)
     (Remove <start>)
)

(p 2:: Refer any CIS student interested in EXS to Dr. Dell
     ((Student    ↑name <name>    ↑phone <phone>    ↑area EXS    ↑advisor NA)    <student>)
  -->
     (Modify <student>    ↑advisor DR. DELL)
     (Make Note    ↑to DR. DELL    ↑from CIS    ↑phone <phone>
          ↑message <name> INTERESTED IN EXS)
)

(p 3:: John is supported by the CIS Department
     (Student    ↑name JOHN)
   -(Employee    ↑name JOHN    ↑employer CIS)
   -(Note    ↑to CHAIRMAN    ↑from CIS-JOB    ↑message JOHN NEEDS A JOB)
  -->
     (Make Note    ↑to CHAIRMAN    ↑from CIS-JOB    ↑message JOHN NEEDS A JOB)
)

(p 4:: Warning to working students with more than 9 credit hours
     ((Student    ↑name <name>    ↑phone <phone>    ↑credit {> 9})    <student>)
     (Employee    ↑name <name>)
  -->
     (Make Note    ↑to <name>    ↑from CIS    ↑phone <phone>
          ↑message PLEASE CONTACT CIS OFFICE — CREDIT OVER LOAD)
)

(p 5:: Mrs. Lester assigns TAs
     ((Note    ↑to CHAIRMAN    ↑from CIS-JOB)    <note>)
  -->
     (Modify <note>    ↑to MRS. LESTER)
)

(p 6:: Strange restriction for non-employed DB students
     (Student    ↑area DB    ↑name <x>    ↑advisor <y>    ↑phone <phone>)
   -(Employee    ↑name <x>)
   -(Advisor    ↑name <y>    ↑area DB)
   -(Note    ↑to <x>
          ↑message YOUR COMMITTEE MUST HAVE A DB FACULTY MEMBER)
  -->
     (Make Note    ↑to <x>    ↑from CIS    ↑phone <phone>
          ↑message YOUR COMMITTEE MUST HAVE A DB FACULTY MEMBER)
)

(Make Start)
```

This program contains six production rules — production rule 1 is used as an initialization rule.

The first working memory element to be created is *Start*; its creation sends rule 1 to the conflict set. Once rule 1 is executed it sends rule 2 and rule 3 to the conflict set. Then, the program has two options, either the program fires rule 2 or it fires rule 3.

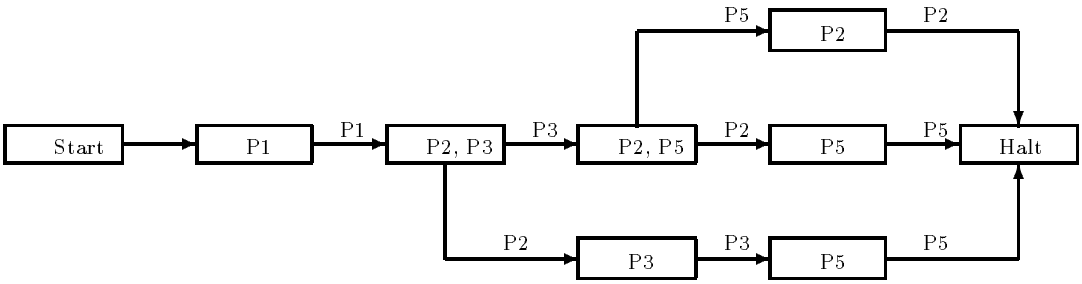

Figure 5.1. Possible ways of executing rules.

If rule 2 is fired, then the conflict set will contain only one rule — rule 3. Since that entails no conflict, rule 3 is fired. This action sends rule 5 to the conflict set, rule 5 is executed, and the conflict set becomes empty, causing a HALT state.

On the other hand, if rule 3 is fired instead of rule 2, then the conflict set will contain two rules: rule 2 and rule 5. After the execution of both rules — in either order — the program will fall into a HALT state.

Figure 5.1 shows the different ways in which the rules can be executed by OPS5. The order of execution depends solely upon the implementation of the conflict set resolution strategies. When the program reaches the halt state, it contains the following working memory elements:

(Student ↑name JOHN ↑advisor DR. FRYE ↑area DB
↑credit 9 ↑phone 371-4627)
(Student ↑name MARY ↑advisor DR. LEE ↑area DB
↑credit 9 ↑phone 322-1472)
(Student ↑name RICHARD ↑advisor DR. DELL ↑area EXS
↑credit 12 ↑phone 335-7167)
(Employee ↑name MARY ↑employer CIS)
(Advisor ↑name DR. DELL ↑area EXS ↑phone 392-2791)
(Advisor ↑name DR. LEE ↑area ARQ ↑phone 392-2693)
(Advisor ↑name DR. FRYE ↑area DB ↑phone 392-2272)
(Note ↑to DR. DELL ↑from CIS ↑phone 335-7167
↑message RICHARD INTERESTED IN EXS)
(Note ↑to MRS. LESTER ↑from CIS–JOB
↑message JOHN NEEDS A JOB)

5.3 The Output — an ADB Definition for the OPS5 Program

Given the OPS5 program shown above, the algorithm generates the following relations and triggers:

```
CREATE (Student,
       name, advisor, area, credit, phone)
CREATE (Employee,
       name, employer)
CREATE (Advisor,
       name, area, phone)
CREATE (Note,
       to, from, phone, message)
CREATE (Start,
       dummy)

(trigger 1::Start
 EVENT:
   INSERT(Start)
 CONDITION:
   EXISTS (SELECT INTO Temp1 *
          FROM Start start1
          WHERE())
 ACTION:
   INSERT INTO Student VALUES
     (JOHN, DR. FRYE, DB, 9, 371-4627)
   INSERT INTO Student VALUES
     (MARY, DR. LEE, DB, 9, 322-1472)
   INSERT INTO Student VALUES
     (RICHARD, DR. DELL, EXS, 12, 335-7167)
   INSERT INTO Employee VALUES
     (MARY, CIS)
   INSERT INTO Advisor VALUES
     (DR. DELL, EXS, 392-2791)
   INSERT INTO Advisor VALUES
     (DR. LEE, ARQ, 392-2693)
   INSERT INTO Advisor VALUES
     (DR. FRYE, DB, 392-2272)
   DELETE FROM Start
         WHERE()
)

(trigger 2::Refer any CIS student interested in EXS to Dr. Dell
 EVENT:
   INSERT(Student) OR
   UPDATE(Student)
 CONDITION:
   EXISTS (SELECT INTO temp1 *
          FROM Student student1
          WHERE ((student1.area=EXS) AND (student1.advisor=NA)))
 ACTION:
   UPDATE Student
         SET  Student.advisor=DR. DELL
         WHERE ((student.area=EXS) AND (student.advisor=NA)))
   INSERT INTO Note VALUES
     (DR. DELL, CIS, (SELECT temp1.phone FROM temp1),
      (SELECT temp1.name FROM temp1) INTERESTED IN EXS)
)

(trigger 3::John is supported by the CIS Department
 EVENT:
   INSERT(Student) OR
   INSERT(Employee) OR
   INSERT(Note) OR
   UPDATE(Note)
 CONDITION:
   EXISTS (SELECT INTO temp1 *
          FROM Student student1
          WHERE (student1.name=JOHN))
   AND
   NOT EXISTS (SELECT *
             FROM Employee employee2
             WHERE ((employee2.name=JOHN) AND (employee2.employer=CIS)))
   AND
   NOT EXISTS (SELECT *
             FROM Note note3
             WHERE ((note3.to=CHAIRMAN) AND (note3.from=CIS-JOB) AND
                  (note3.message=JOHN NEEDS A JOB)))
 ACTION:
   INSERT INTO Note VALUES
     (CHAIRMAN, CIS-JOB, , JOHN NEEDS A JOB)
)
```

```
(trigger 4::Warning to working students with more than 9 credit hours
 EVENT:
   INSERT(Student) OR
   INSERT(Employee)
 CONDITION:
   EXISTS (SELECT INTO temp1 *
          FROM Student student1, Employee employee2
          WHERE ((student1.name=employee2.name) AND (student1.credit ¿ 9)))
 ACTION:
   INSERT INTO Note VALUES
     ((SELECT temp1.name FROM temp1), CIS,
      (SELECT temp1.phone FROM temp1),
      PLEASE CONTACT CIS OFFICE — CREDIT OVER LOAD)
)

(trigger 5::Mrs. Lester assigns TAs
 EVENT:
   INSERT(Note) OR
   UPDATE(Note)
 CONDITION:
   EXISTS (SELECT INTO temp1 *
          FROM Note note1
          WHERE ((note1.to=CHAIRMAN) AND (note1.from=CIS-JOB)))
 ACTION:
   UPDATE Note
     SET  Note.to=MRS. LESTER
     WHERE ((note.to=CHAIRMAN) AND (note.from=CIS-JOB))
)

(trigger 6::Strange restriction for non-employed DB students
 EVENT:
   INSERT(Student) OR
   INSERT(Employee) OR
   INSERT(Advisor) OR
   UPDATE(Student) OR
   INSERT(Note) OR
   UPDATE(Note)
 CONDITION:
   EXISTS ((SELECT INTO temp1 *
           FROM Student student1
           WHERE (student1.area=DB)
          )
          MINUS
          ((SELECT *student1
            FROM Student student1, Employee employee2
            WHERE (student1.name=employee2.name)
           )
           UNION
           (SELECT *student1
            FROM Student student1, Advisor advisor3
            WHERE ((student1.advisor=advisor3.name) AND
                  (advisor3.area=DB))
           )
           UNION
           (SELECT *student1
            FROM Student student1, Note note4
            WHERE ((student1.name=note4.to) AND
                  (note4.message=YOUR COMMITTEE MUST HAVE A DB
                   FACULTY MEMBER)))))
 ACTION:
   INSERT INTO Note VALUES
     ((SELECT temp1.name FROM temp1), CIS,
      (SELECT temp1.phone FROM temp1),
      YOUR COMMITTEE MUST HAVE A DB FACULTY MEMBER)
)

INSERT INTO Start VALUES
  (True)
```

There is a one–to–one relation between production rules and triggers. In the ADB, triggers are constantly being checked against the existing data — stored in the relations. The condition is evaluated when an event is *signaled*; if the condition is fulfilled, then the actions are executed. The working memory element *Start* is a

single fact, i.e., it has no attributes attached to it. With the OPS5 LITERALIZE declaratives the algorithm defines the following relations:

STUDENT	name	advisor	area	credit	phone

EMPLOYEE	name	employer

ADVISOR	name	area	phone

NOTE	to	from	phone	message

START	dummy

The insertion of a tuple in the relation *Start* causes the insertion of data into the other relations, and consequently the execution — or firing — of the triggers that modify the existing data. This "triggering" stops when there are no more triggers to fire. At that stage the database contains the following values:

STUDENT	name	advisor	area	credit	phone
	JOHN	DR. FRYE	DB	9	371-4627
	MARY	DR. LEE	DB	9	322-1472
	RICHARD	DR. DELL	EXS	12	335-7167

EMPLOYEE	name	employer
	MARY	CIS

ADVISOR	name	area	phone
	DR. DELL	EXS	392-2791
	DR. LEE	ARQ	392-2693
	DR. FRYE	DB	392-2272

NOTE	to	from	phone	message
	DR. DELL	CIS	335-7167	RICHARD ...
	MRS. LESTER	CIS–JOB		JOHN NEEDS A JOB

START	Dummy

Notice that this database produces the same result as the OPS5 program did; the only difference is that in this case the data is stored in a set of relations.

The algorithm that produced this result is divided in two phases. The next two sections describe in detail each phase.

5.4 Description of Phase 1

The first phase has three important goals to achieve:

a. create the relations to be used in the ADB,

b. create a table to store the vector–attributes, and

c. find dependencies among the production rules.

The first goal is achieved by processing each LITERALIZE declarative in the OPS5 program. A LITERALIZE declarative implies a new relation in the ADB.

For example, in the case of:

```
(literalize Employee
      name
      employer
)
```

VECTOR_TABLE	vector_name	relation

Figure 5.2. VECTOR_TABLE: used to store vector–attribute information

our algorithm generates the following:

```
CREATE (Employee,
    name, employer)
```

The second goal — create a table to store the vector–attributes — is achieved by processing the VECTOR–ATTRIBUTE declarative. Figure 5.2 shows the information stored for each vector–attribute.

The third goal — find dependencies between the production rules — is a more difficult goal to achieve. To obtain these dependencies all the productions rules in the OPS5 program are processed.

For each attribute in a relation the algorithm handles two sections: the LHS–section and the RHS–section. The LHS–section contains the list of the productions that *refer* to the attribute in their LHS. The RHS–section contains the list of productions that *act* on the attribute in their RHS.

At the LHS of the production rules the *reference* to an attribute can be either *positive*, as in

(Student ↑name JOHN),

or *negated*, as in

−(Employee ↑name JOHN ↑employer CIS).

In the RHS the same situation exists: an action can be *positive*, as in

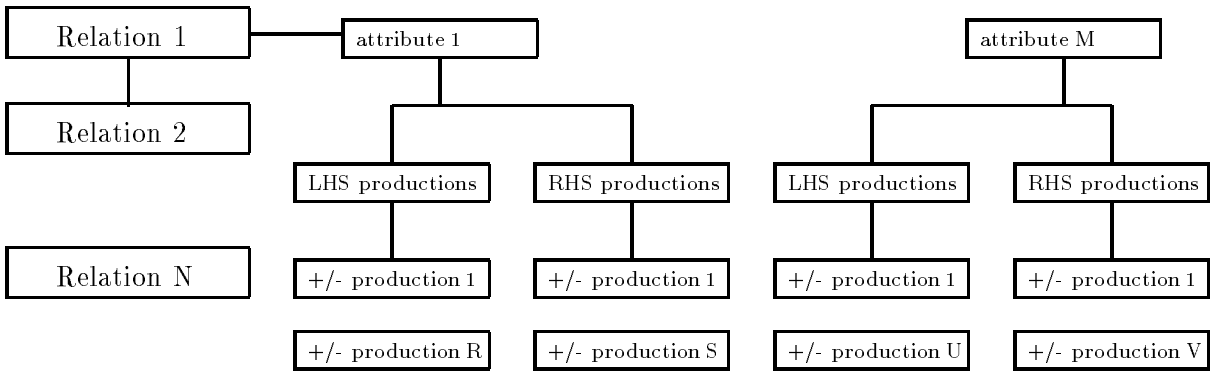

Figure 5.3. RELA_STRUC: used to store production rule dependencies

(Modify <student> ↑advisor DR. DELL), and

(Make Employee ↑MARY ↑employer CIS)

or *negated*, as in

(Remove <start>).

MAKE and MODIFY are the only two *positive actions*, while REMOVE is the only *negated action*. Also, notice that the actions MAKE and REMOVE involve *all* the attributes of a relation R, while the MODIFY action involves only those attributes being modified in R. Thus, along with the production name, a + or a − is stored, depending on whether the reference/action is positive or negated.

Figure 5.3 suggests a structure — called RELA_STRUC — to store these dependencies.

Some conclusions can be obtained, depending upon the values of the LHS–section and the RHS–section for a given attribute ATTR. These conclusions can be categorized in the following cases:

- the Dependency Case,
- the Initial Value Case,

- the Terminal Case, and
- the Don't Care Case.

5.4.1 The Dependency Case

LHS–section = $\{+A_1, +A_2 \ldots + A_n, -B_1, -B_2 \ldots - B_m\}$, and

RHS–section = $\{+X_1, +X_2 \ldots + X_p, -Y_1, -Y_2 \ldots - Y_q\}$.

Which means that the production rules $+X_1, +X_2 \ldots + X_p, -Y_1, -Y_2 \ldots - Y_q$ have the attribute ATTR at their RHS, and the production rules $+A_1, +A_2 \ldots + A_n, -B_1, -B_2 \ldots - B_m$ have the attribute ATTR at their LHS.

There is always a *dependency* between production rules of the RHS–section and production rules of the LHS–section. There are four dependency cases:

1. A positive action in the RHS with a positive reference in the LHS. If the positive reference matches with the positive action, then a new instantiation will enter the conflict set. With this dependency it is possible to augment the conflict set.

2. A positive action in the RHS with a negated reference in the LHS. Probably the positive action will disable a production rule that was in the conflict set. In this case no new instantiations enter the conflict set.

3. A negated action in the RHS with a positive reference in the LHS. Probably the negated action will disable a production rule that was in the conflict set. In this case no new instantiations enter the conflict set.

4. A negated action in the RHS with a negated reference in the LHS. If the negated reference matches the negated action, then a new instantiation will enter the conflict set. Thus, in this dependency it is possible to augment the conflict set.

Whenever a dependency exists we write $X_p \rightarrow A_n$, and we say that *production rule A_n depends on production rule X_p.*

For a given attribute ATTR, with the LHS–section and the RHS–section given above, there are $(n + m)(p + q)$ dependecies:

$$
\begin{array}{l}
X_1 \rightarrow A_1,\ X_1 \rightarrow A_2 \ldots X_1 \rightarrow A_n,\ X_1 \rightarrow B_1,\ X_1 \rightarrow B_2 \ldots X_1 \rightarrow B_m, \\
X_2 \rightarrow A_1,\ X_2 \rightarrow A_2 \ldots X_2 \rightarrow A_n,\ X_2 \rightarrow B_1,\ X_2 \rightarrow B_2 \ldots X_2 \rightarrow B_m, \\
\quad \vdots \\
X_p \rightarrow A_1,\ X_p \rightarrow A_2 \ldots X_p \rightarrow A_n,\ X_p \rightarrow B_1,\ X_p \rightarrow B_2 \ldots X_p \rightarrow B_m, \\
Y_1 \rightarrow A_1,\ Y_1 \rightarrow A_2 \ldots Y_1 \rightarrow A_n,\ Y_1 \rightarrow B_1,\ Y_1 \rightarrow B_2 \ldots Y_1 \rightarrow B_m, \\
Y_2 \rightarrow A_1,\ Y_2 \rightarrow A_2 \ldots Y_2 \rightarrow A_n,\ Y_2 \rightarrow B_1,\ Y_2 \rightarrow B_2 \ldots Y_2 \rightarrow B_m, \\
\quad \vdots \\
Y_q \rightarrow A_1,\ Y_q \rightarrow A_2 \ldots Y_q \rightarrow A_n\ Y_q \rightarrow B_1,\ Y_q \rightarrow B_2 \ldots Y_q \rightarrow B_m.
\end{array}
$$

Notice that with the dependency types 1 and 4 is possible to *augment* the conflict set. A dependency that belongs to any of these types is a *signed dependency.* The signed dependency exists when the two production rules involved in the dependency have the same *"sign"*. For a signed dependency we write $X_p \Rightarrow A_n$.

In the general case, there are $(np + mq)$ signed dependencies:

$$
\begin{array}{l}
X_1 \Rightarrow A_1,\ X_1 \Rightarrow A_2 \ldots X_1 \Rightarrow A_n, \\
X_2 \Rightarrow A_1,\ X_2 \Rightarrow A_2 \ldots X_2 \Rightarrow A_n, \\
\quad \vdots \\
X_p \Rightarrow A_1,\ X_p \Rightarrow A_2 \ldots X_p \Rightarrow A_n, \\
Y_1 \Rightarrow B_1,\ Y_1 \Rightarrow B_2 \ldots Y_1 \Rightarrow B_m, \\
Y_2 \Rightarrow B_1,\ Y_2 \Rightarrow B_2 \ldots Y_2 \Rightarrow B_m, \\
\quad \vdots \\
Y_q \Rightarrow B_1,\ Y_q \Rightarrow B_2 \ldots Y_q \Rightarrow B_m.
\end{array}
$$

The set of these signed dependencies is called the *signed dependency set of attribute ATTR*. The *signed dependency set of an action X* is the union of the signed dependency sets of the attributes involved in the action *X*.

Notice that transitivity can be applied to these dependencies, i.e. if P $\Rightarrow$Q, and Q $\Rightarrow$R, then P $\Rightarrow$R. But we write P $\rightarrow\ \rightarrow$R, to specify that the dependency is an *inferred dependecy*.

An OPS5 program contains a *cycle* when a production rule P executes an action X that causes a finite number of production rules to be executed, and the execution of those rules causes the original production rule P to execute action X again.

Formally, a cycle occurs when either of the following two situations arise:

- There is a production rule P that contains an action X such that P $\Rightarrow$P belongs to the signed dependency set of the action X. We called this a *direct cycle*.
- There is a production rule P that contains an action X such that P $\rightarrow\ \rightarrow$P can be inferred from the signed dependency set of X. We called this a *non–direct cycle*.

A cycle implies an endless loop in the execution model, and is a situation that needs to be avoided in any OPS5 program.

5.4.2 The Initial Value Case

LHS–section = $\{+A_1, +A_2 \ldots + A_n, -B_1, -B_2 \ldots - B_m\}$, and

RHS–section = $\{\}$,

or

LHS–section = $\{+A_1, +A_2 \ldots + A_n, -B_1, -B_2 \ldots - B_m\}$, and

RHS–section = $\{-Y_1, -Y_2 \ldots - Y_q\}$.

In any of these cases no production rule gives a value to the attribute ATTR in its RHS. However some rules refer to it in their LHS, which means that the attribute must have been initialized before the execution, i.e. an *initialization action* is expected to take place on attribute ATTR.

We call attribute ATTR the *initial value attribute*. In the OPS5 program given above, the attribute *dummy* is an initial value attribute.

5.4.3 The Terminal Case

LHS–section = {}, and

RHS–section = $\{+X_1, +X_2 \ldots + X_p, -Y_1, -Y_2 \ldots - Y_q\}$.

In this case no production rule refers to attribute ATTR in its LHS. However there are some production rules acting on it. Thus, the modifications being made at the production rules on the RHS–section will produce no effect on the execution of the OPS5 program.

We call the attribute ATTR a *terminal attribute* because no matter what value it takes it will not produce a change in the program's behavior.

5.4.4 The Don't Care Case

LHS–section = {}, and

RHS–section = {}.

In this case the attribute ATTR appears in no production rule. This means that we can erase it from the relation and the execution model will not be affected. We call the attribute ATTR a *don't care attribute*.

5.4.5 The Dependencies stored at RELA_STRUC in our example

The structure — RELA_STRUC — for the OPS5 program given above is the following:

RELATION	ATTRIBUTE	LHS--SECTION	RHS--SECTION
STUDENT	name	+p2, +p3, +p4, +p6	+p1
	advisor	+p2, +p6	+p1, +p2
	area	+p2, +p6	+p1
	credit	+p4	+p1
	phone	+p2, +p4, +p6	+p1
EMPLOYEE	name	-p3, +p4, +p6	+p1
	employer	-p3	+p1
ADVISOR	name	-p6	+p1
	area	-p6	+p1
	phone		+p1
NOTE	to	-p3, +p5, -p6	+p2, +p3, +p4, +p5, +p6
	from	-p3, +p5	+p2, +p3, +p4, +p6
	phone		+p2, +p4, +p6
	message	-p3, -p6	+p2, +p3, +p4, +p6
START	dummy	+p1	-p1

In this table the attribute START.dummy is an *initial value attribute*, and the attribute ADVISOR.phone is a *terminal attribute*. There are no *don't care attributes*. There is only one cycle: p5 $\Rightarrow$p5.

5.5 Description of Phase 2

In phase 2, the database triggers are generated. In the algorithm each production rule is processed at a time. For each production rule the process is divided in two stages:

- The first stage generates all the queries to cover all the CEs contained in a production rule.

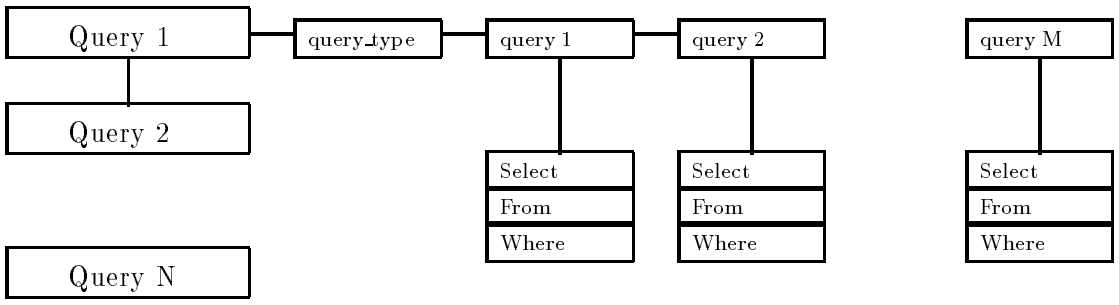

Figure 5.4. QUERY_STRUC: used to store queries

- The second stage generates the triggers using the queries generated in stage 1.

5.5.1 Generation of Queries

For each production rule the algorithm generates the set of queries to cover all its CEs. In figure 5.4 we present the structure needed to store these queries. We have four query types:

- the Independent Positive Query,
- the Independent Negated Query,
- the Dependent Positive Query, and
- the Dependent Negated Query.

The Independent Positive Query Type

This query contains only one IPCE. For example, if the IPCE is

(RA ↑att1 X1 ↑att2 X2 ... ↑attN XN),

the following will be generated:

```
EXISTS (SELECT INTO temp1 *
        FROM RA R1
        WHERE (R1.att1=X1) AND (R1.att2=X2) ... (R1.attN=XN))
```

The Independent Negated Query Type

This query contains only one INCE. For example, if the INCE is

−(RA ↑att1 X1 ↑att2 X2 ... ↑attN XN),

the following query will be generated:

```
NOT EXISTS (SELECT *
            FROM RA R1
            WHERE (R1.att1=X1) AND (R1.att2=X2) ... (R1.attN=XN))
```

The Dependent Positive Query Type

This query contains at least two Positive CEs sharing at least one variable. For example, if the two CEs are

(RA ↑atta <var> ↑att1 X1 ↑att2 X2 ... ↑attN XN), and

(RB ↑attb <var> ↑att1 Y1 ↑att2 Y2 ... ↑attM YM),

the following query will be generated:

```
EXISTS (SELECT INTO temp1 *
        FROM RA R1, RB R2
        WHERE (R1.atta=R2.attb) AND
              (R1.att1=X1) AND (R1.att2=X2) ... (R1.attN=XN) AND
              (R2.att1=Y1) AND (R2.att2=Y2) ... (R2.attM=YM))
```

The Dependent Negated Query Type

This query contains at least two CEs sharing at least one variable; one CE is a Positive CE, and the other CE is a Negated CE. All the variables of all the Negated CEs must be present in at least one of the Positive CEs. For example, if there are three CEs as follows,

(RA ↑atta <var1> ↑attb <var2> ↑att1 X1 ↑att2 X2 ... ↑attN XN), and

−(RB ↑attc <var1> ↑att1 Y1 ↑att2 Y2 ... ↑attM YM), and

−(RC ↑attd <var1> ↑atte <var2> ↑att1 Z1 ↑att2 Z2 ... ↑attP ZP),

the following query will be generated:

```
EXISTS ((SELECT INTO temp1 *
         FROM RA R1
         WHERE (R1.att1=X1) AND (R1.att2=X2) ... (R1.attN=XN)
        )
        MINUS
        ((SELECT *RA
          FROM RA R1, RB R2
          WHERE (R1.atta=R2.attc) AND
                (R1.att1=X1) AND (R1.att2=X2) ... (R1.attN=XN) AND
                (R2.att1=Y1) AND (R2.att2=Y2) ... (R2.attM=YM)
         )
         UNION
         (SELECT *RA
          FROM RA R1, RC R3
          WHERE (R1.atta=R3.attd) AND
                (R1.attb=R3.atte) AND
                (R1.att1=X1) AND (R1.att2=X2) ... (R1.attN=XN) AND
                (R3.att1=Z1) AND (R3.att2=Z2) ... (R3.attP=ZP))))
```

This is a compound query with what is known as a *driver relation* and two *negated relations*. The driver relation is RA, because it is the relation attached to the Positive CE. The negated relations are RB and RC, because they are the relations attached to the two Negated CEs. The final query is made using the following scheme:

(RA MINUS (Π (RA ⋈ RB)

UNION

Π (RA ⋈ RC)))

In general, given n Positive CEs and m Negated CEs we have:

(R1 condition)
(R2 condition)
⋮
(RN condition)
−(S1 condition)
−(S2 condition)
⋮
−(SM condition),

the driver relation is the augmented relation that results of joining the n Positive CEs. This join is possible because the CEs are DPCEs and therefore it is possible to join all of them.

The SELECT clauses of the joins between the driver relation and the negated relations select only those attributes of the driver relation. Thus, the production will generate the following query:

(Driver_relation MINUS (Π (Driver_relation ⋈ S1)
UNION
Π (Driver_relation ⋈ S2)
⋮
UNION
Π (Driver_relation ⋈ SM)))

where Driver_relation = R1 ⋈ R2 ⋈ ...⋈ RN.

5.5.2 Generation of Triggers

The generation of triggers is based upon the queries generated in the previous stage. The queries are used to create the CONDITION and the ACTION clauses of the trigger.

The CONDITION clause of a trigger is generated by "ANDing" all the queries generated for the production rule — stored in QUERY_STRUC. The condition is satisfied if *all* the queries return non-empty results.

The ACTION clause of a trigger is generated using the actions in the production rule. A *SELECT–FROM* statement is placed in the ACTION clause for any *variable* that appears in the action of the production rule. If an *element–variable* appears in the action of the production rule, then the action is executed upon the tuples referred by that *element–variable*.

The EVENT clause of the trigger is generated by using the structure RELA_STRUC, generated in Phase 1. Each action is processed at a time. For each attribute involved in an action, that action is included in the EVENT clause of the triggers related to the production rules in the LHS–section of that attribute. Therefore, EVENTs and ACTIONs clauses are generated in parallel. The EVENTs that are allowed are INSERT, UPDATE or DELETE. The EVENT clause creates a network–like structure of *alarms* — events — that will *wake up* selected triggers when certain events are signaled. Notice that the EVENT clause of any trigger is not defined until the end of the process of *all* the production rules.

5.6 Listing of the Algorithm

Besides RELA_STRUC, QUERY_STRUC and VECTOR_TABLE, the algorithm uses the following structures:

ELE–VAR_TABLE is used to store information about the *element–variables*.

ELE–VAR_TABLE	element_var	alias	query_id

VAR_TABLE is used to store information about the *variables*.

VAR_TABLE	variable	query_id	alias	attribute

Figure 5.5 gives a suggested structure — TRIGGER_STRUC — to store *triggers*.

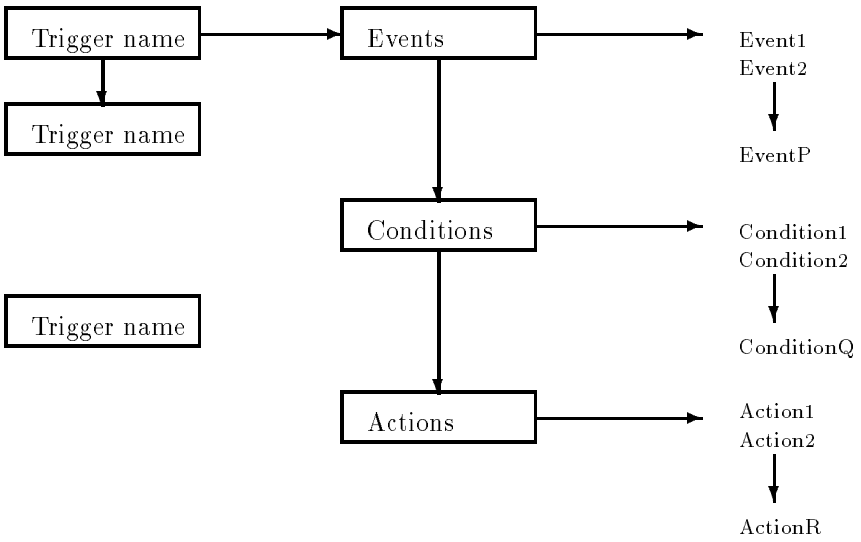

Figure 5.5. TRIGGER_STRUC: used to store triggers.

5.6.1 Phase 1

Pseudo Code of Phase 1

```
for each declarative DEC in the OPS program do
 case DEC of:
   literalize:
     create the relation;
   vector-attribute:
     create the extra-relation to handle it;
   p: begin
     for each CE in the production PROD do
       for each attribute referenced in the CE do
         store PROD at the LHS section of the attribute
       endfor
     endfor
     for each action ACT in PROD do
       for each attribute referenced in ACT do
         store PROD at the RHS section of the attribute
       endfor
     endfor
   end;
 endcase
endfor;
```

Code of Phase 1

```
\* Process of LITERALIZE, VECTOR-ATTRIBUTE or P declaratives *\
for each declarative in the OPS5 program do begin
  case declarative of:

  LITERALIZE: begin
    get the relation name --> RELA;
    if (there are no attributes attached to the relation)
    then create a relation RELA with the only attribute DUMMY
    else create a relation RELA with all the attributes being
         declared after the name of the relation;
  end;

  VECTOR-ATTRIBUTE: begin
    for each vector-attribute do
      add the attribute into the VECTOR_TABLE;
  end;

  P: begin
    ce_num = 0;
    ELE-VAR_TABLE = nil;
    get the production name --> PROD;

    \* Process the LHS of the production *\
    for each conditional element in the LHS of the production do begin
      ce_num = ce_num + 1;
      get the relation name --> RELA;
      if (there is an element-variable)
      then begin
        get the element-variable --> EL-VA;
        insert element-variable::relation_name in the ELE-VAR_TABLE;
      end;
      insert ce_num::relation_name in the ELE-VAR_TABLE;
      for each attribute ATTR in the conditional element do begin
        store the production PROD at the LHS-section of the
        attribute ATTR in the relation RELA at the RELA_STRUC;
      endfor;
    endfor; \* end of conditional elements in the LHS *\

    \* Process the RHS of the production *\
    for each action in the RHS of the production do begin
      if (action = MAKE) then get relation name --> RELA
      else begin
        if (there is an element-variable) then
          get the element-variable --> EL-VA;
        else begin
          get the ce_num --> EL-VA;
          get the relation name from the ELE-VAR_TABLE
          using EL-VA --> RELA;
        end;
```

```
      end;
      if (action = MODIFY) then begin
        for each attribute ATTR being modified do begin
          store the production PROD at the RHS-section of the
          attribute ATTR in the relation RELA at the RELA_STRUC;
        endfor
      end
      else begin
        for each attribute ATTR in the relation RELA do begin
          store the production PROD at the RHS--section of the
          attribute ATTR in the relation RELA at the RELA_STRUC;
        endfor
      end;
    endfor;

  end; \* P case *\

endcase; \* case covering LITERALIZE, VECTOR-ATTRIBUTE, and P *\

endfor; \* Declaratives of the OPS5 program *\
```

5.6.2 Phase 2

Pseudo Code of Phase 2

```
for each production PROD in the OPS program do
  for each CE in PROD do
    find the type of the CE;
    generate the query for the CE;
  endfor;
  if (trigger PROD does not exist)
  then generate trigger PROD;
  create the CONDITION of trigger PROD using the
  queries stored at QUERY_STRUC;
  for each action ACT at production PROD do
    for each attribute refered by the action ACT do
      for each production P at the LHS section of
      the attribute in RELA_STRUC do
        Pool:= Pool + P
      endfor
    endfor;
    remove repetitions from Pool;
    for each production P at Pool do
      if (trigger P does not exist)
      then generate trigger P;
      append the action ACT as an EVENT of trigger P
    endfor;
    append the action ACT as an ACTION of trigger PROD
endfor;
```

Code of Phase 2

```
\* Generation of Queries --- Process of the LHS *\
for each PRODUCTION in the OPS5 program do begin

  ce_num = 0;
  initialize ELE-VAR_TABLE;
  initialize QUERY_STRUC;
  initialize VAR_TABLE;
  initialize VECTOR_TABLE;
  initialize QQ; \* query_type, query1 ... queryN *\
  initialize QQUERY; \* Select, From, Where *\
  variables = false;
  query = false;

  get the production name --> PROD;
  for each conditional element (CE) in the production do begin
    ce_num = ce_num + 1;
    if (CE is negated) then begin
      \* Negated CE *\
      get the relation name --> RELA;
      append at QQUERY.From the relation RELA using
      the alias with the ce_num;
      for each attribute in the CE do begin
        if (the value/var is a value) then begin
          generate the condition COND from the attribute
          and the value/var;
          append COND to the QQUERY.Where;
        end
        else begin
          variables = true;
          if (value/var does not appear in the VAR_TABLE)
          then begin
            writeln('ERROR in production', PROD);
            writeln('Negated CE refering to a variable');
            writeln('not present in any previous Positive CE');
          end
          else begin
            generate the condition COND from the attribute
            and the value/var;
            append COND to the QQUERY.Where;
            get the query attached to the variable in
            the VAR_TABLE --> QUE;
            if QUE.type = DP then QUE.type = DN;
            CURRENT = next available query of QUE;
            merge the queries QQUERY and CURRENT into CURRENT;
            for the rest of the attributes in the CE do begin
              if (the value/var is a value) then begin
                generate the condition COND from
                the attribute and the value/var;
                append COND to the CURRENT.Where;
```

```
          end
          else begin
            if (the variable does not appear in the VAR_TABLE)
            then begin
              writeln('ERROR in production', PROD);
              writeln('Negated CE refering to a variable');
              writeln('not present in any previous Positive CE');
            end
            else begin
              get the query attached to the variable
              in the VAR_TABLE --> QUE2;
              if (QUE <> QUE2) then begin
                merge the queries QUE.query1 and QUE2.query1
                into QUE.query1;
                for the rest of queries Q of QUE2 do begin
                  insert Q at the next available query of QUE;
                endfor;
                update all references of QUE2 to QUE in the
                ELE-VAR_TABLE and the VAR_TABLE;
                delete QUE2 from the QUERY_STRUC;
              end;
              generate the condition COND from the attribute
              and the value/var;
              append COND to the CURRENT.Where;
            end
          end
        endfor
      end
    end
  endfor
  if (not variables) then begin
    QQ.type = IN;
    QQ.query1 = QQUERY
    insert QQ as a new query into the QUERY_STRUC
    \* INCE case *\
  end
end
else begin
  \* Positive CE *\
  get the relation name --> RELA;
  append at QQUERY.From the relation RELA using
  the alias with the ce_num;
  if (there is an element-variable) then begin
    get the element-variable --> EL-VA;
    insert the element-variable::relation_name
    in the ELE-VAR_TABLE;
  end;
  insert the ce_num::relation_name in the ELE-VAR_TABLE;
  for each attribute in the CE do begin
    if (the value/var is a value) then begin
      generate the condition COND from the attribute
```

```
  and the value/var;
  append COND to the QQUERY.Where;
end
else begin
  variables = true;
  if (value/var does not appear in the VAR_TABLE) then begin
    insert value/var::RELA::attribute in the VAR_TABLE;
    generate the condition COND from the attribute
    and the value/var;
    append COND to the QQUERY.Where;
  end
  else begin
    query = true;
    generate the condition COND from the attribute
    and the value/var;
    append COND to the QQUERY.Where;
    get the query attached to the variable
    in the VAR_TABLE --> QUE;
    CURRENT = QUE.query1;
    merge the queries QQUERY and CURRENT into CURRENT;
    for each variable V in the VAR_TABLE with no query_id
    do begin
      V.query_id = QUE;
    end;
    Update ELE-VAR_TABLE and VAR_TABLE with QUE
    in the query_id;
    for the rest of the attributes in the CE do begin
      generate the condition COND from
      the attribute and the value/var;
      append COND to the CURRENT.Where;
      if (the value/var is a variable)
      then begin
        if (the variable does not appear in the VAR_TABLE)
        then begin
          insert value/var::RELA::attribute::QUE
          in the VAR_TABLE;
        end
        else begin
          get the query attached to the variable
          in the VAR_TABLE --> QUE2;
          if (QUE <> QUE2) then begin
            merge the queries QUE.query1 and QUE2.query1
            into QUE.query1;
            for the rest of queries Q of QUE2 do begin
              insert Q at the next available query of QUE;
            endfor;
            update all references of QUE2 to QUE in the
            ELE-VAR_TABLE and the VAR_TABLE;
            delete QUE2 from the QUERY_STRUC;
          end
        end
```

```
            end
          endfor
        end
      end
    endfor; \* attributes at Positive CE *\
    if (not variables) then begin
      QQ.type = IP;
      QQ.query1 = QQUERY;
      insert QQ as a new query into the QUERY_STRUC;
      Update ELE-VAR_TABLE and VAR_TABLE with the new query_id;
      \* IPCE case *\
    end
    else begin
      if (not query) then begin
        QQ.type = DP;
        QQ.query1 = QQUERY;
        insert QQ as a new query into the QUERY_STRUC --> QUE;
        Update ELE-VAR_TABLE and VAR_TABLE with the new query_id;
        \* DPCE case *\
      end
    end
  end; \* If checking either Positive CE or Negated CE *\
endfor; \* Conditional elements *\

\* Generation of Triggers --- Process of the RHS *\
look for trigger PROD at TRIGGER_STRUC;
if (trigger does not exist) then generate new trigger PROD;
append at the trigger PROD the CONDITION using
the queries at the QUERY_STRUC;

for each action ACT do begin
  if (ACT = MAKE) then get the relation name --> RELA
  else begin
    if (there is an element-variable) then
      get the element-variable --> EL-VA
    else get the ce_num --> EL-VA;
    get the relation_name from EL-VA at ELE-VAR_TABLE --> RELA;
  end;
  PROD_POOL = nil;
  if (ACT = MODIFY) then
    for each attribute being modified of the relation RELA do
    begin
      for each production P in the LHS-section of the attribute
          at the RELA_STRUC do begin
        PROD_POOL = PROD_POOL + P;
      endfor
    endfor
  else
    for each attribute of the relation RELA do begin
      for each production P in the LHS-section of the attribute
          at the RELA_STRUC do begin
```

```
          PROD_POOL = PROD_POOL + P;
        endfor
      endfor;
    remove repetitions from PROD_POOL;
    for each production P at PROD_POOL do begin
      look for trigger P at TRIGGER_STRUC;
      if (trigger does not exist) then generate new trigger --> P;
      case ACT of
      MAKE: begin
        append at P.EVENTS the INSERT(RELA) event;
      end;
      MODIFY: begin
        append at P.EVENTS the UPDATE(RELA) event;
      end;
      REMOVE: begin
        append at P.EVENTS the DELETE(RELA) event;
      end
      endcase;
      append at PROD.ACTIONS the action ACT (include any variable
      and/or conditional-element referenced on the action
      as a Select_From_Where statement)
    endfor;
  endfor; \* actions *\
endfor; \* productions *\
```

CHAPTER 6
CONCLUSIONS AND FURTHER RESEARCH

The algorithm provided in the previous chapter translates an OPS5 program into a set of relations and triggers in an ADB. This translation makes possible the combination of algorithms and techniques from PSs and DBMSs.

We have shown that triggers and production rules are equivalent. That is, assuming the same conflict set strategies, the execution model of an ADB is equivalent to the execution model of an OPS5 program.

Since an ADB stores data and triggers in secondary memory, the size limitation of the rule base is eliminated. Also, the triggers act upon persistent data, versus the non–persistent data in an OPS5 program. As a result, the "loading phase" required for any OPS5's application is no longer needed.

The separation of data and triggers in an ADB makes possible the concept of shared data for OPS5's applications. With this transalation it will be possible to have different sets of triggers acting upon the same data ("working memory elements"), i.e., data is shared among those sets of triggers.

The translation process gives some information about the behavior of the OPS5 program, information such as: dependencies, initial value attributes, terminal attributes, don't care attributes, and cycles. The information about cycles is used to improve the execution of the OPS5 application in the ADB.

Still there is some work to be done. The optimization of the query processing is a topic that needs to be addressed. We have seen that the most expensive queries are the *Dependent Negated Queries*. The optimization of these queries is necessary

since has been shown that about 30% of the rules in a PS contain negated conditions [Forg83].

More research is needed to find other types of dependency among production rules. The algorithm presented here includes the dependencies from the attribute's point of view. It is possible that a different categorization of production rules will give more information about the inter–relations between production rules in an OPS5 program.

Also, it is possible that adding more database operations as event types may optimize the "activation" of triggers, and will result in an improvement of the overall execution of the ADB system.

This translation assumes that all the events are at the "relation level", i.e. no attribute values are included in any event. It is possible that with events at an "attribute level" the dependencies between production rules will be defined with more accuracy.

REFERENCES

[Barr81] Barr, A., and E. Feigenbaum, The Handbook of Artificial Intelligence, Vol. 1, HeurisTech Press, Stanford, CA, 1981.

[Bein87] Bein, Jonathan, Roger King and Nabil Kamel, MOBY: An Architecture for Distributed Expert Database Systems, Proceedings of the 13th VLDB Conference, Los Angeles, CA, 1987.

[Brow85] Brownston, Lee, Robert Farrell, Elaine Kant, and Nancy Martin, Programming Expert Systems in OPS5 — an Introduction to Rule–Based Programming, Addison–Wesley Publishing Company, Inc., Reading, MA, 1985

[Bune79] Buneman, O. P., and E. K. Clemons, Efficient Monitoring Relational Databases, ACM Transactions on Database Systems, 4(3):368–382, September 1979.

[Care86] Carey M., D. DeWitt, D. Frank, et al., The Architecture of the EXODUS Extensible DBMS, Proceedings of the International Workshop on Object–Oriented Database Systems, September 1986.

[Chak90] Chakravarthy, S., Overview of HiPAC:A Research Project on Active Time–Constrained Database Management, UF–CIS Technical Report TR–90–18, University of Florida, Gainesville, FL, May 1990.

[Chakr89] Chakravarthy, S., Barbara Blaustein, Alejandro Buchmann, Michael Carey, Umeshwar Dayal, David Goldhrisch, Meichun Hsu, Rajiv Jauhari, Rivka Ladin, Miron Livny, Dennis McCarthy, Richard McKee, and Arnon Rosenthal, HiPAC: a Research Project in Active Time–Constrained Database Management, Final Technical Report, Xerox Advanced Information Technology, Cambridge, MA, July 1989.

[Cloc81] Clocksin, William, and Christopher Mellish, Programming in PROLOG, Halliday Litho, West Hanover, MA, 1981.

[Colm73] Colmerauer, A., H. Kanoui, R. Pasero, and P. Roussel, Un Systeme de Comunication Homme–Machine en Francais, Research Report CRI 72–18, Groupe Intelligence Artificielle, Universite Aix–Marseille II, Paris, France, June 1973.

[Eswa76] Eswaran, K. P., Specifications, Implementations and Interactions of a Trigger Subsytem in an Integrated Database System, Technical Report, IBM Research Laboratory, San Jose, CA, 1976

[Forg79] Forgy, C., "On the Efficient Implementation of Production Systems," Department of Computer Sciences, Carnegie–Mellon University, Pittsburgh, PA, Ph. D. Thesis, February 1979.

[Forg81] Forgy, C., OPS5 User's Manual, Technical Report CMU–81–135, Carnegie–Mellon University, Pittsburgh, PA, July 1981.

[Forg82] Forgy, C., RETE: a Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem, Artificial Intelligence 19: 17–37, 1982.

[Forg83] Forgy C., and Anoop Gupta, Measurements on Production Systems, Technical Report CMU–CS–83–167, Carnegie–Mellon University, Pittsburgh, PA, December 1983.

[Forgy77] Forgy, Charles, and J. McDermott, OPS — A Domain–Independent Production System Language, Proceedings of Fifth International Conference on Artificial Intelligence, Cambridge, MA, 1977.

[Frey87] Freytag, J. C., A Rule–Based View of Query Optimization, Proceedings of the ACM–Sigmod Conference on Management of Data, San Francisco, CA, 1987.

[Grae87] Graefe G., and D. DeWitt, The EXODUS Optimizer Generator, Proceedings of the 1987 ACM–SIGMOD Conference on Management of Data, May 1987.

[Grie84] Griesmer, J. H., S. H. Hong, M. Karnaugh, J. L. Kastner, M. I. Schor, R. L. Ennis, D. A. Klein, K. R. Milliken, and H. M. Van Woerkom, YES/MVS: A Continuous Real Time Expert System, Proceedings of the National Conference on Artificial Intelligence, AAAI, 1984.

[Hans89] Hanson, E., An Initial Report on the Design of ARIEL: A DBMS with an Integrated Production Rule System, Sigmod Record, Vol. 18, No. 3, September 1989.

[Kers86] Kershberg, L., Editor, Expert Database Systems: Proceedings from the First International Workshop, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1986.

[Kers87] Kershberg, L., Editor, Expert Database Systems: Proceedings from the First International Workshop, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[McDe81] McDemott, J., R1: The Formative Years, AI Magazine, 2:21–29, 1981.

[McDem78] McDemott, J., A. Newell, and J. Moore, The Efficiency of Certain Production System Implementations, Pattern–Directed Inference Systems, Academic Press, 1978.

[Mins75] Minsky, M., A Framework for Representing Knowledge, The Psychology of Computer Vision, P. Wilson Editor, McGraw Hill: New York, NY, 1975.

[Mira87] Miranker, Daniel P., TREAT: a Better Match Algorithm for AI Production Systems, Technical Report AI TR87–58, July 1987.

[Rych76] Rychener, M., "Production Systems as a Programming Language for Artificial Intelligence," Department of Computer Sciences, Carnegie–Mellon University, Pittsburgh, PA, Ph. D. Thesis, 1976.

[Sell89] Sellis, Timos, Chih–Chen Lin, and Louiqa Raschid, Data Intensive Production Systems: The DIPS Approach, Sigmod Record, Vol. 18, No. 3, September 1989.

[Ston87] Stonebraker, Michael, Eric Hanson, and Chin–Heng Hong, The Design of the PostGres Rules System, Proceedings of the Third International Conference on Data Engineering, IEEE, Los Angeles, CA, February 1987.

[Wido90] Widom, Jennifer, and Sheldon J. Finkelstein, Set–Oriented Production Rules in Relational Databases, Proceedings of the 1990 ACM Sigmod International Conference on Management of Data, Atlantic City, NJ, June 1990.

BIOGRAPHICAL SKETCH

Randall J. Blanco–Mora was born in San José, Costa Rica, in 1966. He is the second son of José and Dora Blanco. He completed his elementary studies at public schools in Coronado, San José, Costa Rica.

In 1983, he began studies at the Computer Science Department at the University of Costa Rica, in San José, Costa Rica. In 1986, he enrolled in the Department of Mathematics and Computer Sciences of Mississippi College, Mississippi. In May 1987, he obtained the bachelor's degrees in computer science and mathematics from Mississippi College.

In January 1989, he began master's degree studies in the Department of Computer and Information Sciences at the University of Florida. His interests include expert systems, active databases, local area networks, and database design.